Small inversions for smaller inversions

Jean-François MONIN
Inversion, simple example

Even natural numbers

Inductive even : \forall 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.

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
Smaller inversion (part of the Braga method)

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
| 0 => 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 G, match G 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’.

\[ e : \text{even } n \]
\[ e' : \text{even } (S (S n)) \]

\[ S (\text{half } n e) = \text{half } (S (S n)) e' \]
Unicity of e

Again: induction on e and inversion on e’

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

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 $\equiv$ and $\leq$ in nat

Bounded natural numbers

Inductive $t : \text{nat} \rightarrow \text{Set} :=$
\[
| \text{FO} \{n\} : t (S \, n) \\
| \text{FS} \{n\} : t \, n \rightarrow t (S \, n).
\]

Failures for standard inversion.
Inversion technologies

<table>
<thead>
<tr>
<th>Standard tactic of Coq: fully automated [Cornes &amp; Terrasse, 1995; Murthy?]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improved over the years, very impressive black box</td>
</tr>
<tr>
<td>lack of control</td>
</tr>
<tr>
<td>big underlying terms</td>
</tr>
<tr>
<td>failures with dependent inductive types</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Small inversions: handcrafted [Monin 2010, Monin &amp; Shi 2013]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flexible approach with several variants</td>
</tr>
<tr>
<td>Developed for a big experiment with CompCert</td>
</tr>
<tr>
<td>Attempts towards automation (Braibant, Boutillier)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TYPE’2022</th>
</tr>
</thead>
<tbody>
<tr>
<td>Made clearer with auxiliary inductive types</td>
</tr>
<tr>
<td>Improvement needed for dependent types</td>
</tr>
</tbody>
</table>
Small inversions with auxiliary inductive types

Recipe

Given an inductive relation \( \text{rel} : \text{Tx} \rightarrow \text{Ty1} \rightarrow \ldots \text{Prop} \)
with “input” argument \( x : \text{Tx} \), define:

- For each input case (constructor \( C \)) in \( \text{Tx} \),
  an auxiliary inductive relation of type \( \text{Ty1} \rightarrow \ldots \text{Prop} \)
  by copy and paste of relevant telescopes of \( \text{rel} \)
- No recursion

- A dispatch function \( \text{rel\_disp} \) from \( x : \text{Tx} \) to \( \text{Ty1} \rightarrow \ldots \text{Prop} \)
  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 match \( \text{rel\_inv} R \) with...

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

**Recipe**

Given an inductive relation \( \text{rel} : T_x \rightarrow Ty_1 \rightarrow \ldots \text{Prop} \) with “input” argument \( x : T_x \), define:

- For each input case (constructor \( C \)) in \( T_x \), an *auxiliary inductive relation* of type \( Ty_1 \rightarrow \ldots \text{Prop} \) by copy and paste of relevant telescopes of \( \text{rel} \)
- No recursion

- A *dispatch function* \( \text{rel\_disp} \) from \( x : T_x \) to \( Ty_1 \rightarrow \ldots \text{Prop} \) 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 \( (C\ldots) \)
Explicit injectivity

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

Then rewrite the occurrences of \( R \) with \( \text{rel}_\text{back} (\text{rel}_\text{inv} R) \) before the pattern-matching on \( \text{rel}_\text{inv} R \).

Improvement: built-in injectivity

- In the previous recipe, *add a last argument of shape \( C \ldots \)*
- *Same code* for \( \text{rel}_\text{disp} \) and \( \text{rel}_\text{inv} \)
- *Bonus*: inline \( \text{rel}_\text{disp} \) in the statement of \( \text{rel}_\text{inv} \)
Basic small inversion on even [2021 talks]

Inductive even : \( \forall \ n, \ \text{Prop} := \)
  | Ev0 : \(\text{even} \ 0\)
  | Ev2 \(\ n : \ \text{even} \ n \ \rightarrow \ \text{even} \ (S \ (S \ n))\).

Inductive even0 : \text{Prop} := even0_Ev0 : even0.
Inductive even1 : \text{Prop} :=.
Inductive even2 \(\ n : \ \text{Prop} := \) even2_Ev2 : even \(\ n \ \rightarrow \ \text{even2} \ n\).

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

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

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

Inductive even0 : Prop := even0_Ev0 : even0.
Inductive even1 : Prop :=.
Inductive even2 n : Prop := even2_Ev2 : even n \rightarrow 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.
Improved small inversion on even with built-in injectivity

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 \texttt{nat} \to \texttt{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 recipe with built-in injectivity *)
Inductive \texttt{is\_dia0} : \texttt{diag} 0 0 \to \texttt{Prop} := \texttt{ii00} : \texttt{is\_dia0} \texttt{dia0}.
Inductive \texttt{is\_diaS} x y : \texttt{diag} (S x) (S y) \to \texttt{Prop} :=
  \texttt{iiSS} : \forall (d : \texttt{diag} x y), \texttt{is\_diaS} x y (\texttt{diaS} x y d).
Inductive \texttt{no\_diag} x y : \texttt{diag} x y \to \texttt{Prop} := .

Definition \texttt{diag\_inv} \{x y\} (d : \texttt{diag} x y) :
  match x, y return \texttt{diag} x y \to \texttt{Prop} with
  | 0, 0 => \texttt{is\_dia0}
  | S x, S y => \texttt{is\_diaS} x y
  | x, y => \texttt{no\_diag} x y
end d.

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

Inductive \( \text{diag} : \text{nat} \rightarrow \text{nat} \rightarrow \text{Prop} := \)
| \( \text{dia}0 : \text{diag} \ 0 \ 0 \)
| \( \text{diaS} \ x \ y : \text{diag} \ x \ y \rightarrow \text{diag} \ (S \ x) \ (S \ y). \)

(* small inversion : standard receipe with built-in injectivity *)
Inductive \( \text{is dia}0 : \text{diag} \ 0 \ 0 \rightarrow \text{Prop} := \) \( \text{ii}00 : \text{is dia}0 \ \text{dia}0. \)
Inductive \( \text{is dia}S \ x \ y : \text{diag} \ (S \ x) \ (S \ y) \rightarrow \text{Prop} := \)
\( \text{iiSS} : \forall \ (d : \text{diag} \ x \ y), \text{is dia}S \ x \ y \ (\text{diaS} \ x \ y \ d). \)
Inductive \( \text{no dia} x \ y : \text{diag} \ x \ y \rightarrow \text{Prop} := . \)
Definition \( \text{diag inv} \ \{x \ y\} \ (d : \text{diag} \ x \ y) : \)
match \( x, y \) return \( \text{diag} \ x \ y \rightarrow \text{Prop} \) with
| \( 0, 0 \) => \( \text{is dia}0 \)
| \( S \ x, S \ y \) => \( \text{is dia}S \ x \ y \)
| \( x, y \) => \( \text{no dia} x \ y \)
end \( d. \)
Proof. destruct \( d; \) constructor. Qed.
**Definition** \texttt{diag_refl} \{x\} : \texttt{diag} x x.

Proof. induction \texttt{x} as [ | \texttt{x} \texttt{IHx}]; constructor. apply \texttt{IHx}. Defined.

**Definition** \texttt{eqdiag} \{x y\} (e : x = y) : \texttt{diag} x y.

Proof. case \texttt{e}. apply \texttt{diag_refl}. Defined.

**Definition** \texttt{diag_back} \{x\} : \forall y, \texttt{diag} x y \rightarrow x = y.

Proof. induction \texttt{x}; destruct \texttt{y}; intro d; destruct (\texttt{diag_inv} \texttt{d});

[reflexivity | apply \texttt{f_equal}, (\texttt{IHx} _ d)]. Defined.

**Lemma** \texttt{diag_back_isrefl} \{x\} : \forall (d : \texttt{diag} x x), \texttt{eq_refl} = \texttt{diag_back} d.

Proof. induction \texttt{x} as [ | \texttt{x} \texttt{IHx}]; simpl; intro d; destruct (\texttt{diag_inv} \texttt{d});

[reflexivity | case (\texttt{IHx} \texttt{d}). cbn. reflexivity]. Qed.

**Lemma** \texttt{diag_mono} \{x y\} (e : x = y) : e = \texttt{diag_back} (\texttt{eqdiag} e).

Proof. destruct \texttt{e}; destruct \texttt{x} as [ | \texttt{x}]; simpl.
+ destruct (\texttt{diag_inv} \texttt{dia0}); reflexivity.
+ destruct (\texttt{diag_inv} (\texttt{diaS} x x \texttt{diag_refl})) as [\texttt{d}]. case (\texttt{diag_back_isrefl} \texttt{d}); reflexivity.

Qed.

**Corollary** UIP_{nat} (x: \texttt{nat}) (e : x = x) : \texttt{eq_refl} = e.

Proof. rewrite (\texttt{diag_mono} e). apply \texttt{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]; simplify; 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]; simplify.
    + 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.
Definition \texttt{diag_refl} \{x\} : \texttt{diag} x x.
Proof. induction \texttt{x} as [ \mid \texttt{x} \texttt{IHx}] ; constructor. apply \texttt{IHx}. Defined.

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

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

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

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

Corollary \texttt{UIP_nat} (x: nat) (e : x = x) : \texttt{eq_refl} = e.
Proof. rewrite (\texttt{diag_mono} e). apply \texttt{diag_back_isrefl}. Qed.
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\} : \(\forall\ (d : `diag` x x), `eq_refl` = `diag_back\ d\).
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 \( \leq \)?

Inversion performed “as if” \( \leq \) was defined as

\[
\begin{align*}
\text{Inductive le n : nat \to Prop :=} \\
\quad | \text{le_e_0 : n = 0} & \Rightarrow n \leq 0 \\
\quad | \text{le_e_S m : n = S m} & \Rightarrow n \leq S m \\
\quad | \text{le_S m : n \leq m} & \Rightarrow n \leq S m.
\end{align*}
\]

Definition \text{eq_le n m (e : n = m)} : n \leq m :=

\[
\begin{align*}
\text{match e with eq_refl \Rightarrow le_n n end.}
\end{align*}
\]

Inductive \text{le_0 [n]} : n \leq 0 \to Prop :=

\[
\begin{align*}
\text{| le_0_e : \forall e, le_0 (eq_le e)}.
\end{align*}
\]

Inductive \text{le_Sm [m n]} : n \leq S m \to Prop :=

\[
\begin{align*}
\text{| le_Sm_e : \forall e, le_Sm (eq_le e)} \\
\text{| le_Sm_S : \forall l, le_Sm (le_S n m l)}.
\end{align*}
\]

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

\[
\begin{align*}
\text{match m with} \\
\quad | 0 \Rightarrow le_0 \\
\quad | S \ m \Rightarrow @le_Sm m \\
\text{end n l.}
\end{align*}
\]
Equality is too easy, what about \( \leq \)?

Inversion performed “as if” \( \leq \) was defined as

```coq
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 [ | 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/