Smaller Inversions and Unleashed Recursion in Coq

The Braga Method

Jean-François MONIN

The Braga method is joint work with Dominique Larchey-Wendling
Coq material and references

Small inversions

http://home/jf/www/Proof/Small_inversions/2021/

The Braga method


Dominique Larchey-Wendling and Jean-François Monin.


Unleashed recursion

Write partially/non terminating functional programs in Coq
To be extracted in OCaml exactly as desired

Key ingredient : small inversions

- From V0 (2010-2013) to V1 (2017): V1 quite simple
- Less simple in recursive programs, issue solved with V2 (2018-2020)
- Beat Coq standard inversion (V0, V1) in case of dependent types including with the Braga method (V3, 2021)

Empty inductive types and looping forever

Kind of basic case
Basically, only **total** functions as programs

- **Termination certificate** needed at definition time
- But termination may depend on partial correctness (and conversely)
- Partially terminating functions make sense
- Extraction: partial functions allowed in target language
OCaml and Coq are

Functional programming languages

- Functions as ordinary values
- Recursion
- **Algebraic** types
- Static type-checking (and type inference)
- Polymorphism
Algebraic types

Construction
- cartesian products = juxtaposition of $n$ things
- sums (disjoint unions) = choice between $m$ cases distinguished by a unique name (constructor)

Ideal for tree-like structures
- Lists, binary (search) trees), etc.
- Abstract Syntax Trees
- Rule-based semantics
- Proof-trees

Analyzed by a central weapon
PATTERN MATCHING on constructors
Algebraic types

Construction
- cartesian products $=$ juxtaposition of $n$ things
- sums (disjoint unions) $=$
  choice between $m$ cases distinguished by a unique name (constructor)

Ideal for tree-like structures
- Lists, binary (search) trees, etc.
- Abstract Syntax Trees
- Rule-based semantics
- Proof-trees

Analyzed by a central weapon
PATTERN MATCHING on constructors
<table>
<thead>
<tr>
<th>In OCaml only</th>
</tr>
</thead>
<tbody>
<tr>
<td>primitive data types (int, char, etc.)</td>
</tr>
<tr>
<td>imperative features</td>
</tr>
<tr>
<td>exceptions – <em>inhabit any type</em></td>
</tr>
<tr>
<td>non terminating computations – <em>inhabit any type</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>In Coq only</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computations on types; types have a type (called a universe)</td>
</tr>
</tbody>
</table>
| **Dependent types** :  
  the *type* of an expression may depend on  
  the *value* of another expression  
  Example: lists having a *given length* |
| Algebraic types with *zero* cases (*empty* in the empty environment) |
| Special universe *Prop* dedicated to proof-trees |
Recursion in Coq (more generally: type theory)

Limited to structural recursion in order to ensure termination

Not that serious

- Not a theoretical issue (e.g., inductive definition of WF relations)
- In practice: various tricks
- Precisely the point of the Braga method
Explicit/implicit terms

OCaml and Coq: explicit definitions
Explicit code for data types and functions

Coq only: interactive mode
- step-by-step development of functions driven by types
- Using tactics
- interactive building... and interactive reading!
- Hidden code, visible specification
- Especially convenient when dealing with dependent types
Special universe for assertions: Prop

Dedicated to Coq proof-trees

- Proofs are native
- $t : P$ means that $t$ is a proof of $P$
- The “empty” type in $\text{Prop}$ is just False ($\bot$)
  Consistency forbids exceptions and non-terminating computations
- A proof of $P \Rightarrow Q$ is seen as a function (program) with
  a proof of $P$ in input and proof of $Q$ in output.
  Actual notation: $P \rightarrow Q$
- A proof of $\forall x : A, Q x$ is seen as a function (program) with
  a value $x : A$ in input and providing a proof of $Q x$.
- Remark $Q : A \rightarrow \text{Prop}$
  Predicates are just dependent types
Lemma elimination

Lemma \texttt{many\_primes} : \forall \ n: \text{nat}, \exists \ p: \text{nat}, \ n \leq p \land \text{prime} p.

Proof of the statement

Theorem \texttt{thm1} : some other statement

Proof using \texttt{(many\_primes 1960) and (many\_primes 24^3)}

Can be computed into

Theorem \texttt{thm1} : some other statement

Proof including specific proofs

of \exists p: \text{nat}, \ 1960 \leq p \land \text{prime} p

and \exists p: \text{nat}, \ 24^3 \leq p \land \text{prime} p.
Computations on proof trees

Provides meaning
to reasoning by case analysis

Not performed in practice
We don’t care

Excepted for reducing recursive functions
when the structurally decreasing argument is in Prop
Computations on proof trees

Provides meaning to reasoning by case analysis

Not performed in practice

We don’t care

Excepted for reducing recursive functions

when the structurally decreasing argument is in Prop
From OCaml to Coq and conversely: summary

Formal reasoning boils down to

data and computation
presented by proof trees

Coq provides a uniform framework dealing in the same way with

- typed programs
- proofs of properties
From OCaml to Coq and conversely: extraction (1/2)

Coq

Fixpoint minlist l : list A (n: non_empty l) :
    {y : A | mem y l ∧ ∀x, mem x l → x ≤ y} :=
match l with
| [] ⇒ something for this absurd case
| x :: l ⇒ code computing y and proofs
end

The proof tree needs not to be computed for computing the result y

OCaml

let rec minlist l : α list : α =
match l with
| [] -> assert false
| x :: l -> code computing y only
end
Coq

Fixpoint minlist l : list A (n: non_empty l) :
  {y : A | mem y l \(\land\) \(\forall\) x, mem x l \(\rightarrow\) x \(\leq\) y} :=
  match l with
  | [] => something for this absurd case
  | x :: l => code computing y and proofs
end

The proof tree needs not to be computed for computing the result y

OCaml

let rec minlist l : α list : α =
  match l with
  | [] -> assert false
  | x :: l -> code computing y only
end
Separation between
- “real” data (and functions on them)
- (logical) knowledge or reasoning about them

No information leakage between Prop and Type
- Statically ensured by a constraint on pattern-matching
- Some debatable exceptions

Terms in Prop can be erased
- From Coq to compilable functional languages (OCaml, Haskell,...)
- Aka dead-code elimination, “never executed asserts”
- An elegant way to provide correct-by-construction programs
The Braga method (first presented at Types’18, Braga)

In type theory (CIC++): only total functions
- Termination certificate (TC) needed at definition time
- Many possible types for the TC: any (recursive) inductive type
  Issues to be considered before writing the function itself

Studying partial correctness properties is useful
- before getting knowledge
- or even in order to get knowledge on termination
  Concrete example: first order unification
→ Egg and chicken problem

Partially terminating functions make sense
- WF relation are then a too strong requirement
- TC interpreted as a domain argument

Extraction: partial functions allowed in target language
LISP viz ML style

**LISP**

if \( l = [] \) then 0
else \( f1 (\text{head } l) + f2 (\text{tail } l) \)

**Proof obligations** to ensure that \text{head} and \text{tail} are called with a non-empty argument :-(

**ML**

match \( l \) with
| [] \rightarrow 0
| \( h::t \) \rightarrow f1 h + f2 t

Type checking does the job :)

---

J-F. Monin  
small inversions & recursion  
June 2021 17/71
LISP viz ML style

### LISP

```lisp
(if \(l = []\) then 0
else \(f1\ (head\ l) + f2\ (tail\ l)\)
```

**Proof obligations** to ensure that `head` and `tail` are called with a non-empty argument :(

### ML

```ml
match \(l\) with
| [] \(\rightarrow\) 0
| \(h :: t\) \(\rightarrow\) \(f1\ h + f2\ t\)
```

Type checking does the job :)
Easy dependent pattern matching

Generalization is mandatory

Definition deptyp n : Type :=
  match n with
  | O ⇒ bool
  | 1 ⇒ nat
  | _ ⇒ unit
end.

Definition fct1 n : deptyp n :=
  match n return deptyp n with
  | O ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt (* () in OCaml *)
end.

Definition fct2 n : deptyp (n*n) :=
  match n*n as n2 return deptyp n2 with
  | O ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt
end.
Easy dependent pattern matching

Definition \texttt{deptyp} \( n \) : \texttt{Type} :=
\begin{align*}
& \text{match } n \text{ with} \\
& \mid 0 \Rightarrow \texttt{bool} \\
& \mid 1 \Rightarrow \texttt{nat} \\
& \mid _{} \Rightarrow \texttt{unit} \\
& \text{end}.
\end{align*}

Definition \texttt{fct1} \( n \) : \texttt{deptyp} \( n \) :=
\begin{align*}
& \text{match } n \text{ return } \texttt{deptyp} \( n \) \text{ with} \\
& \mid 0 \Rightarrow \texttt{false} \\
& \mid 1 \Rightarrow 3 \\
& \mid _{} \Rightarrow \texttt{tt} (* () in \texttt{OCaml} *) \\
& \text{end}.
\end{align*}

Definition \texttt{fct2} \( n \) : \texttt{deptyp} \((n\times n)\) :=
\begin{align*}
& \text{match } n\times n \text{ as } n2 \text{ return } \texttt{deptyp} \( n2 \) \text{ with} \\
& \mid 0 \Rightarrow \texttt{false} \\
& \mid 1 \Rightarrow 3 \\
& \mid _{} \Rightarrow \texttt{tt} \\
& \text{end}.
\end{align*}

Generalization is mandatory
Easy dependent pattern matching

Generalization is mandatory

Definition \texttt{deptyp} \( n : \text{Type} := \)
\begin{align*}
\text{match } n \text{ with} \\
\mid 0 & \Rightarrow \text{bool} \\
\mid 1 & \Rightarrow \text{nat} \\
\mid _ & \Rightarrow \text{unit}
\end{align*}
end.

Definition \texttt{fct1} \( n : \text{deptyp} \ n := \)
\begin{align*}
\text{match } n \text{ return deptyp } n \text{ with} \\
\mid 0 & \Rightarrow \text{false} \\
\mid 1 & \Rightarrow 3 \\
\mid _ & \Rightarrow \text{tt} \quad (* \text{() in OCaml} *)
\end{align*}
end.

Definition \texttt{fct2} \( n : \text{deptyp} (n*n) := \)
\begin{align*}
\text{match } n*n \text{ as } n2 \text{ return deptyp } n2 \text{ with} \\
\mid 0 & \Rightarrow \text{false} \\
\mid 1 & \Rightarrow 3 \\
\mid _ & \Rightarrow \text{tt}
\end{align*}
end.
Definition \texttt{deptyp} \( n : \text{Type} \) := 
match \( n \) with 
| \( 0 \) \Rightarrow \text{bool} 
| \( 1 \) \Rightarrow \text{nat} 
| _ \Rightarrow \text{unit} 
end.

Definition \texttt{fct1} \( n : \text{deptyp} \ n := \)
match \( n \) return \( \text{deptyp} \ n \) with 
| \( 0 \) \Rightarrow \text{false} 
| \( 1 \) \Rightarrow 3 
| _ \Rightarrow \text{tt } (* () \text{ in OCaml } *) 
end.

Definition \texttt{fct2} \( n : \text{deptyp} \ (n*n) := \)
match \( n*n \) as \( n2 \) return \( \text{deptyp} \ n2 \) with 
| \( 0 \) \Rightarrow \text{false} 
| \( 1 \) \Rightarrow 3 
| _ \Rightarrow \text{tt} 
end.
Easy dependent pattern matching

Generalization is mandatory

Definition \textit{deptyp} \( n \) : Type :=
\[
\text{match } n \text{ with}
| 0 \Rightarrow \text{bool}
| 1 \Rightarrow \text{nat}
| _ \Rightarrow \text{unit}
\text{end}.
\]

Definition \textit{fct1} \( n \) : \textit{deptyp} \( n \) :=
\[
\text{match } n \text{ return } \textit{deptyp} \( n \) \text{ with}
| 0 \Rightarrow \text{false}
| 1 \Rightarrow 3
| _ \Rightarrow \text{tt} \quad (* () \text{ in OCaml } *)
\text{end}.
\]

Definition \textit{fct2} \( n \) : \textit{deptyp} \( n^2 \) :=
\[
\text{match } n^2 \text{ as } n2 \text{ return } \textit{deptyp} \( n2 \) \text{ with}
| 0 \Rightarrow \text{false}
| 1 \Rightarrow 3
| _ \Rightarrow \text{tt}
\text{end}.
\]
Trojan horse, general idea

Carry information (here: $G$) to be revealed after coming into the place. The type of $G$ (for guard) depends on the case.

Definition $\text{is\_cons } l : \text{Prop} :=$
\[
\text{match } l \text{ with } :: \Rightarrow \top \mid _- \Rightarrow \bot \text{ end}.
\]

Definition $\text{head } l : \text{is\_cons } l \rightarrow X :=$
\[
\text{match } l \text{ with } \\
\mid x :: t \Rightarrow \lambda G, x \\
\mid _- \Rightarrow \lambda G, \text{match } G \text{ with end end}
\]
LISP (terrible) style with embedded proofs

Definition `is_nil` (l : list X) : bool :=
  match l with
  | [] ⇒ true
  | _ :: _ ⇒ false
end.

Lemma nil_false : is_nil [] = false → ⊥.

Definition head (l : list X) : is_nil l = false → X :=
  match l with
  | x :: l ⇒ λ G, x
  | _ ⇒ λ G, match nil_false G with end
end.

Definition LISP_style l : nat :=
  (if is_nil l as b return (is_nil l = b → nat)
   then λ (pre : is_nil l = true), 0
   else λ (pre : is_nil l = false),
     f1 (head l pre) + f2 (tail l pre)
   ) eq_refl.
Usual universal realizer: exception

```ml
let univ : α = assert false
```

Another universal realizer: loop

```ml
let rec loop x = loop x
```
Universal realizer

Usual universal realizer: exception

```
let univ : α = assert false
```

Another universal realizer: loop

```
let rec loop x = loop x
```
Section sec_absurd.

Variable X : Type.
Variable f : ⊥.

(* An arbitrary inductive proposition *)
Definition P : Prop := ⊤.

Let Fixpoint loop (x:P) : X := loop (match f with end).

Hypothesis p : P.
Definition Floop_P : X := loop p.

End sec_absurd.
Loops with an absurd parameter in Coq

The same in 2 lines (with \(\top\) for \(P\))

Definition \texttt{Floop\textunderscore T} (X: Type) (f: \(\bot\)) : X :=

\((\text{fix loop (}_:-\top):=\text{ loop (match f with end)))\) \texttt{I}.

The same in 2 shorter lines (with \(\bot\) for \(P\))

Definition \texttt{Floop\textunderscore F} (X: Type) : \(\bot\) -> X :=

\texttt{fix loop f := loop (match f with end)}.

An additional concrete parameter for better extraction

Definition \texttt{Floop} (X: Type) : \(\bot\) -> X :=

\((\text{fix loop t (f:}\bot):=\text{ loop tt (match f with end)))\) \texttt{tt}.
The same in 2 lines (with $\top$ for $P$)

Definition $\text{Floop}_T (X : \text{Type}) (f : \bot) : X :=$
\hspace{1cm}
(fix loop (_ : $\top$) := loop (match f with end))) I.

The same in 2 shorter lines (with $\bot$ for $P$)

Definition $\text{Floop}_F (X : \text{Type}) : \bot \rightarrow X :=$
\hspace{1cm}
fix loop $f :=$ loop (match $f$ with end).

An additional concrete parameter for better extraction

Definition $\text{Floop} (X : \text{Type}) : \bot \rightarrow X :=$
\hspace{1cm}
(fix loop $t (f : \bot) :=$ loop $tt (\text{match } f \text{ with end}) \text{)) } tt.$
Loops with an absurd parameter in Coq

The same in 2 lines (with ⊤ for P)

Definition Floop_T (X: Type) (f: ⊥) : X :=
(fix loop (_:⊤) := loop (match f with end)) I.

The same in 2 shorter lines (with ⊥ for P)

Definition Floop_F (X: Type) : ⊥ -> X :=
fix loop f := loop (match f with end).

An additional concrete parameter for better extraction

Definition Floop (X: Type) : ⊥ -> X :=
(fix loop t (f:⊥) := loop tt (match f with end)) tt.
Definition **Fexc** \( \{X : \text{Type}\} \ (f : \bot) : X := \)

\[
\text{match } \text{Floop } \text{Empty} \text{set} \ f \ \text{with end}.
\]

\(\text{Empty} \text{set} = \) empty informative inductive type

\(\text{Floop } \text{Empty} \text{set} \ f\) has type \(\text{Empty} \text{set}\)

At Coq level, no leakage from Prop to Type

\[
\text{match } \text{whatever} \ \text{with end}
\]

extracted at OCaml level as assert false

\(\text{Floop } \text{(params)}\) considered as dead code \(\rightarrow \) canceled
Inductively defined semantics

Inductive eval : te -> val -> Prop :=
  | E_Const : forall n,
    eval (Te_const n) (Nval n)
  | E_Plus : forall t1 t2 n1 n2,
    eval t1 (Nval n1) ->
    eval t2 (Nval n2) ->
    eval (Te_plus t1 t2) (Nval (n1 + n2)).

Two goals

e : eval (Te_plus (Te_const 1) (Te_const 2)) v
v = Nval 3

3 = 5
Inversion

Purpose
Extract the information contained in a hypothesis $H$ of type $T$
- where $T$ is an inductive relation
- with some inductive arguments

Expectations
- Only relevant cases (constructors) for $T$ are kept
- In the remaining cases, decompose $H$ into its components

Essentially : (subtle) case analysis on $H$
- Simultaneous case analysis on $H$ and its arguments
- game on dependent pattern-matching
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)
- Made clearer with recent unpublished improvements
- Other improvements needed for the Braga method
A real example with CompCert C semantics (2013)

\( \text{H:eval_expr} \ (\text{Genv.globalenv prog_adc}) \ e \ m \ \text{RV} \)

\[
\begin{align*}
(\text{Ecall} \ (\text{Evalof} \ (\text{Evar} \ \text{copy_StatusRegister} \ T14) \ T14) \\
(\text{Econs} \\
(\text{Eaddrof} \\
(\text{Efield} \ (\text{Ederef} \ (\text{Evalof} \ (\text{Evar} \ \text{proc} \ T3) \ T3) \ T6) \\
\text{adc_compcert.cpsr} \ T7) \ T8) \\
(\text{Econs} \\
(\text{Ecall} \ (\text{Evalof} \ (\text{Evar} \ \text{spsr} \ T15) \ T15) \\
(\text{Econs} \ (\text{Evalof} \ (\text{Evar} \ \text{proc} \ T3) \ T3) \ \text{Enil}) \ T8) \\
\text{Enil})) \\
T12) \ t \ m' \ a' \\
\end{align*}
\]

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

\text{proc_state_related} \ m' \ e \ st'

\( \text{inv H. inv H4. inv H9. inv H5. inv H4. inv H5.} \)
\( \text{inv H15. inv H4. inv H5. inv H14. inv H4. inv H3.} \)
\( \text{inv H15. inv H5. inv H4. inv H5. inv H21. inv H13.} \)
\( \ldots \)
Practical issues with Coq standard inversion

- Behavior not easy to predict
  number of cases, number and type of components
- Many additional equalities to be rewritten
- Scripts depend on the versions of Coq
  (and of CompCert for the previous case study)
- Heavy machinery generating gigantic underlying proof terms
- Underlying reasoning somewhat mysterious
- Fails in situations with dependent types
e : eval (Te_div0 (Te_const 1)) \forall

3 = 5

pose (diag t :=
  match t with
    | Te_div0 (Te_const 1) => 3 = 5
    | _ => True
  end).
change (diag (Te_div0 (Te_const 1))).
destruct e; simpl; exact I.
A more modular variant

Definition inv_eval_1_div0 t v (e: eval t v) :=
  let diag t :=
    match t with
      | Te_div0 n => \forall X: Prop, X
      | _ => True
    end
  in match e in eval t v return diag t with
    | E_Const n => I
    | E_Plus _ _ n1 n2 H1 H2 => I
  end.

e : eval (Te_div0 (Te_const 1)) v

3 = 5

apply (inv_eval_1_div0 e).
Small Inversions V0: diagonalization function

- yields the premises of focused constructor
- independent from specific conclusion
- takes bindings into account

For constructor $E_{\text{Plus}}$:

```latex
\text{diag } t \ v := \text{match } t \ \text{with}
\begin{align*}
| \ Te_{\text{plus}} \ tc1 \ tc2 \Rightarrow & \forall X: \text{te} \to \text{Prop}, \\
& (\forall n1 \ n2, \text{eval } tc1 \ (Nval \ n1) \to \\
& \text{eval } tc2 \ (Nval \ n2) \to \\
& X \ (Nval \ (n1 + n2))) \to X \ v \\
| _ \Rightarrow \text{True}
\end{align*}
end
```

**NO ADDITIONAL EQUALITY**
Small Inversions V0: diagonalization function

- yields the premises of focused constructor
- independent from specific conclusion
- takes bindings into account

For constructor E_Plus:

\[
\text{diag } t \ v := \text{match } t \text{ with }
\]

\[
\mid \text{Te} \_ \text{plus } tc1 \ tc2 \Rightarrow
\]

\[
\quad \forall \ X: \text{te} \rightarrow \text{Prop},
\]

\[
\quad (\forall \ n1 \ n2, \text{eval } tc1 (\text{Nval } n1) \rightarrow
\quad \text{eval } tc2 (\text{Nval } n2) \rightarrow
\quad X (\text{Nval } (n1 + n2))) \rightarrow X \ v
\]

\[
\mid \ _ \Rightarrow \text{True}
\]

end

**No additional equality**
Small Inversions V0: diagonalization function

- yields the premises of focused constructor
- independent from specific conclusion
- takes bindings into account

For constructor E_PLUS:

diag t v := match t with
  | Te_plus tc1 tc2 =>
    ∀ X: te -> Prop,
    (∀ n1 n2, eval tc1 (Nval n1) ->
    eval tc2 (Nval n2) ->
    X (Nval (n1 + n2))) -> X v
  | _ => True
end

**No additional equality**
Small inversions V1, with auxiliary inductive types

Recipe

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

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

- A dispatch function \( \text{rel'} \) from \( x : T_x \) to \( T_y \rightarrow \ldots \ \text{Prop} \) by pattern matching on \( x \)

- A trivial proof \( \text{rel}_\text{rel'} : \text{rel} \implies \text{rel'} \)

Usage

- Given a hypothesis \( R : \text{rel} (C\ldots) \\ \text{expr} \ldots \) invoke a pattern matching on \( \text{rel}_\text{rel'} R \)
- Boils down to the relevant aux. inductive relation corresponding to (C\ldots)
Small inversions V1, 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 \emph{auxiliary inductive relation} of type \( \text{Ty1} \rightarrow \ldots \text{Prop} \) by copy and paste of relevant telescopes of \( \text{rel} \)
- No recursion

- A \emph{dispatch function} \( \text{rel}' \) from \( x : \text{Tx} \) to \( \text{Ty1} \rightarrow \ldots \text{Prop} \) by pattern matching on \( x \)

- A trivial proof \( \text{rel}_\text{rel} : \text{rel} \text{ implies } \text{rel}' \)

Usage

- Given a hypothesis \( R : \text{rel} (C\ldots) \text{ expr_1\ldots} \)
  invoke a \emph{pattern matching} on \( \text{rel}_\text{rel}' R \)

- Boils down to the relevant \emph{aux. inductive relation} corresponding to \( (C\ldots) \)
Small inversion V1, for dependent (data) types

Complement of receipe

When $R$ occurs as an argument in the goal (usually happens for dependent data types rather than relations), we need also the converse $rel'\_rel$ of $rel\_rel'$ (trivial as well), and a proof of $rel'\_rel\ (rel\_rel'\ R) = R$.

Then rewrite the occurrences of $R$ with $rel'\_rel\ (rel\_rel'\ R)$ before the pattern-matching on $rel\_rel'\ R$.

To be completed, or see script:
http://www-verimag.imag.fr/~monin/Proof/Small_inversions/2021/
**Inductive** \texttt{eval : te \rightarrow val \rightarrow Prop :=}

| \texttt{E_Const : \forall n,}
| \quad \texttt{eval (Te\_const n) (Nval n)}
| \texttt{E_Plus : \forall t1 t2 n1 n2,}
| \quad \texttt{eval t1 (Nval n1) \rightarrow eval t2 (Nval n2) \rightarrow eval (Te\_plus t1 t2) (Nval (n1 + n2)).}

**Inductive** \texttt{eval\_Const' n : val \rightarrow Prop :=}

| \texttt{E_Const' : eval\_Const' n (Nval n)}.

**Inductive** \texttt{eval\_Plus' t1 t2 : val \rightarrow Prop :=}

| \texttt{E_Plus' : \forall n1 n2,}
| \quad \texttt{eval t1 (Nval n1) \rightarrow eval t2 (Nval n2) \rightarrow eval\_Plus' t1 t2 (Nval (n1 + n2)).}

**Definition** \texttt{eval' : te \rightarrow val \rightarrow Prop := fun t =>}

\texttt{match t with}

| \texttt{Te\_const n \Rightarrow eval\_Const' n}
| \texttt{Te\_plus t1 t2 \Rightarrow eval\_Plus' t1 t2}

\texttt{end.}
**Inductive** \texttt{eval : te \rightarrow val \rightarrow Prop :=}

| E_Const : \forall n, \\ eval (Te\_const n) (Nval n) \\
| E_Plus : \forall t1 t2 n1 n2, \\ eval t1 (Nval n1) \rightarrow eval t2 (Nval n2) \rightarrow \\ eval (Te\_plus t1 t2) (Nval (n1 + n2)).

**Inductive** \texttt{eval\_Const' n : val \rightarrow Prop :=}

| E_Const' : eval\_Const' n (Nval n).

**Inductive** \texttt{eval\_Plus' t1 t2 : val \rightarrow Prop :=}

| E_Plus' : \forall n1 n2, \\ eval t1 (Nval n1) \rightarrow eval t2 (Nval n2) \rightarrow \\ eval\_Plus' t1 t2 (Nval (n1 + n2)).

**Definition** \texttt{eval' : te \rightarrow val \rightarrow Prop := fun t =>}

match t with 
| Te_const n => eval\_Const' n \\
| Te\_plus t1 t2 => eval\_Plus' t1 t2 
end.
Inductive eval : te → val → Prop :=
  | E_Const : ∀ n,
    eval (Te_const n) (Nval n)
  | E_Plus : ∀ t1 t2 n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval (Te_plus t1 t2) (Nval (n1 + n2)).

Inductive eval_Const' n : val → Prop :=
  | E_Const' : eval_Const' n (Nval n).

Inductive eval_Plus' t1 t2 : val → Prop :=
  | E_Plus' : ∀ n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval_Plus' t1 t2 (Nval (n1 + n2)).

Definition eval' : te → val → Prop := fun t =>
  match t with
  | Te_const n => eval_Const' n
  | Te_plus t1 t2 => eval_Plus' t1 t2
end.
**Inductive** \textit{eval} : \textit{te} → \textit{val} → \textit{Prop} :=

| E_Const : \forall n,
  eval (\textit{Te\_const} n) (\textit{Nval} n)
| E_Plus : \forall t1 t2 n1 n2,
  eval t1 (\textit{Nval} n1) → eval t2 (\textit{Nval} n2) →
  eval (\textit{Te\_plus} t1 t2) (\textit{Nval} (n1 + n2)).

**Inductive** \textit{eval\_Const'} \textit{n} : \textit{val} → \textit{Prop} :=

| E_Const' : \textit{eval\_Const'} \textit{n} (\textit{Nval} \textit{n}).

**Inductive** \textit{eval\_Plus'} \textit{t1} \textit{t2} : \textit{val} → \textit{Prop} :=

| E_Plus' : \forall n1 n2,
  eval t1 (\textit{Nval} n1) → eval t2 (\textit{Nval} n2) →
  eval\_Plus' t1 t2 (\textit{Nval} (n1 + n2)).

**Definition** \textit{eval'} : \textit{te} → \textit{val} → \textit{Prop} := \text{fun } t ⇒

match \textit{t} with
| \textit{Te\_const} n ⇒ \textit{eval\_Const'} \textit{n}
| \textit{Te\_plus} t1 t2 ⇒ \textit{eval\_Plus'} \textit{t1} \textit{t2}
end.
Definition \texttt{eval\_eval'} \{t v\} : \texttt{eval t v} \rightarrow \texttt{eval'} t v.
Proof. intro e; destruct e; constructor; assumption. Qed.

Definition \texttt{eval\_eval'}\_bavard \{t v\} : \texttt{eval t v} \rightarrow \texttt{eval'} t v := \lambda e, match e in \texttt{eval t v} return \texttt{eval'} t v with
\mid \texttt{E\_Const} n => \texttt{E\_Const'} n
\mid \texttt{E\_Plus} t1 t2 n1 n2 e1 e2 => \texttt{E\_Plus'} t1 t2 n1 n2 e1 e2 end.
Definition eval_eval' {t v} : eval t v → eval' t v.
Proof. intro e; destruct e; constructor; assumption. Qed.

Definition eval_eval'_bavard {t v} : eval t v → eval' t v := λ e, match e in eval t₀ v₀ return eval' t₀ v₀ with
| E_Const n => E_Const' n : (eval_Const' n) (Nval n)
| E_Plus t₁ t₂ n₁ n₂ e₁ e₂ => E_Plus' t₁ t₂ n₁ n₂ e₁ e₂ : (eval_Plus' t₁ t₂) (Nval (n₁ + n₂)) end.
Definition \texttt{eval_eval'} \{ t \ v \} : \texttt{eval} \ t \ v \ \rightarrow \ \texttt{eval'} \ t \ v.
Proof. intro \ e; destruct \ e; constructor; assumption. Qed.

Definition \texttt{eval_eval'_bavard} \{ t \ v \} : \texttt{eval} \ t \ v \ \rightarrow \ \texttt{eval'} \ t \ v := \lambda \ e, 
\quad \text{match} \ \ e \ \text{in} \ \texttt{eval} \ t_0 \ v_0 \ \text{return} \ \texttt{eval'} \ t_0 \ v_0 \ \text{with} \\
\quad \mid \ \texttt{E_Const} \ n \quad (* \ t_0 := \texttt{Te} \texttt{\_const} n, \ v_0 := \texttt{Nval} n *) \\
\quad \Rightarrow \ \texttt{E_Const'} \ n : (\texttt{eval} \texttt{\_Const'} n) \ (\texttt{Nval} n) \\
\quad \mid \ \texttt{E_Plus} \ t_1 \ t_2 \ n_1 \ n_2 \ e_1 \ e_2 \quad (* t_0 := \texttt{Te} \texttt{\_plus} t_1 \ t_2, \ v_0 := \texttt{Nval} (n_1+n_2) *) \\
\quad \Rightarrow \ \texttt{E_Plus'} \ t_1 \ t_2 \ n_1 \ n_2 \ e_1 \ e_2 : (\texttt{eval} \texttt{\_Plus'} t_1 \ t_2) \ (\texttt{Nval} (n_1+n_2)) \\
\text{end.}
**Definition** \( \text{eval\_eval'} \{ t \; v \} : \text{eval} \; t \; v \rightarrow \text{eval'} \; t \; v. \)

**Proof.** intro \( e \); destruct \( e \); constructor; assumption. **Qed.**

**Definition** \( \text{eval\_eval'}\_bavard \{ t \; v \} : \text{eval} \; t \; v \rightarrow \text{eval'} \; t \; v := \lambda\; e, \)

match \( e \) in \( \text{eval} \; t_0 \; v_0 \) return \( \text{eval'} \; t_0 \; v_0 \) with

| \( \text{E\_Const} \; n \) (*) \( t_0 := \text{Te\_const} \; n, \; v_0 := \text{Nval} \; n \) *)

=> \( \text{E\_Const'} \; n : (\text{eval\_Const'} \; n) \; \text{Nval} \; n \)

| \( \text{E\_Plus} \; t_1 \; t_2 \; n_1 \; n_2 \; e_1 \; e_2 \) (*) \( t_0 := \text{Te\_plus} \; t_1 \; t_2, \; v_0 := \text{Nval} \; (n_1+n_2) \) *)

=> \( \text{E\_Plus'} \; t_1 \; t_2 \; n_1 \; n_2 \; e_1 \; e_2 : (\text{eval\_Plus'} \; t_1 \; t_2) \; \text{Nval} \; (n_1+n_2) \)

end.
Definition eval_eval’ \{t \ v\} : eval t v \rightarrow eval’ t v.
Proof. intro e; destruct e; constructor; assumption. Qed.

Definition eval_eval’_bavard \{t \ v\} : eval t v \rightarrow eval’ t v := \lambda e, match e in eval t_0 v_0 return eval’ t_0 v_0 with
| E_Const n (*) t_0:= Te_const n, v_0:= Nval n *) => E_Const’ n : (eval_Const’ n) (Nval n)
| E_Plus t1 t2 n1 n2 e1 e2 (*t_0:= Te_plus t1 t2, v_0:= Nval (n1+n2)*) => E_Plus’ t1 t2 n1 n2 e1 e2 : (eval_Plus’ t1 t2) (Nval (n1+n2)) end.
Definition **eval_eval'** {t v} : eval t v → eval' t v.

**Proof.** intro e; destruct e; constructor; assumption. Qed.

Definition **eval_eval'_bavard** {t v} : eval t v → eval' t v := λ e, match e in eval t₀ v₀ return eval' t₀ v₀ with

| E_Const n (* t₀ := Te_const n, v₀ := Nval n *) => E_Const' n : (eval_Const' n) (Nval n)

| E_Plus t₁ t₂ n₁ n₂ e₁ e₂ (* t₀ := Te_plus t₁ t₂, v₀ := Nval (n₁+n₂) *) => E_Plus' t₁ t₂ n₁ n₂ e₁ e₂ : (eval_Plus' t₁ t₂) (Nval (n₁+n₂)) end.
Inductive eval_Const’ n : val → Prop :=
  | E_Const’ : eval_Const’ n (Nval n).

Inductive eval_Plus’ t1 t2 : val → Prop :=
  | E_Plus’ : ∀ n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval_Plus’ t1 t2 (Nval (n1 + n2)).

e : eval (Te_const 1) v

v = Nval 1

destruct (eval_eval’ e).

e : eval (Te_plus (Te_const 1) (Te_const 0)) v

v = Nval 1

destruct (eval_eval’ e) as [n1 n2 e1 e2].

No additional equality
**Inductive** \texttt{eval\_Const'} \texttt{n} : \texttt{val} → \texttt{Prop} :=
\hspace{1em} | \texttt{E\_Const'} : \texttt{eval\_Const'} \texttt{n} (\texttt{Nval} \texttt{n}).

**Inductive** \texttt{eval\_Plus'} \texttt{t1} \texttt{t2} : \texttt{val} → \texttt{Prop} :=
\hspace{1em} | \texttt{E\_Plus'} : \forall \texttt{n1} \texttt{n2},
\hspace{2em} \texttt{eval} \texttt{t1} (\texttt{Nval} \texttt{n1}) → \texttt{eval} \texttt{t2} (\texttt{Nval} \texttt{n2}) →
\hspace{2em} \texttt{eval\_Plus'} \texttt{t1} \texttt{t2} (\texttt{Nval} (\texttt{n1} + \texttt{n2})).

\texttt{e} : \texttt{eval} (\texttt{Te\_const} 1) \texttt{v}

\texttt{v} = \texttt{Nval} 1
destruct \texttt{(eval\_eval'} \texttt{e}).

\texttt{e} : \texttt{eval} (\texttt{Te\_plus} (\texttt{Te\_const} 1) (\texttt{Te\_const} 0)) \texttt{v}

\texttt{v} = \texttt{Nval} 1
destruct \texttt{(eval\_eval'} \texttt{e}) as [\texttt{n1} \texttt{n2} \texttt{e1} \texttt{e2}].
Inductive eval_Const’ n : val → Prop :=
  | E_Const’ : eval_Const’ n (Nval n).

Inductive eval_Plus’ t1 t2 : val → Prop :=
  | E_Plus’ : ∀ n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval_Plus’ t1 t2 (Nval (n1 + n2)).

\begin{align*}
e : \text{eval } (Te_{\text{const} 1}) v \\
\texttt{v = Nval 1} \texttt{=} \\
\text{destruct } \text{eval}_eval’ e.
\end{align*}

\begin{align*}
e : \text{eval } (Te_{\text{plus} (Te_{\text{const} 1}) (Te_{\text{const} 0})}) v \\
\texttt{v = Nval 1} \texttt{=} \\
\text{destruct } \text{eval}_eval’ e \texttt{as } [n1 n2 e1 e2].
\end{align*}

\textbf{No additional equality}
Inductive eval_Const' n : val \to Prop :=
  | E_Const' : eval_Const' n (Nval n).

Inductive eval_Plus' t1 t2 : val \to Prop :=
  | E_Plus' : \forall n1 n2,
      eval t1 (Nval n1) \to eval t2 (Nval n2) \to
      eval_Plus' t1 t2 (Nval (n1 + n2)).

\[ e : eval (Te_const 1) v \]
\[ v = Nval 1 \]

destruct (eval_eval' e).

\[ e : eval (Te_plus (Te_const 1) (Te_const 0)) v \]
\[ v = Nval 1 \]

destruct (eval_eval' e) as [n1 n2 e1 e2].
Inductive eval_Const’ n : val → Prop :=
  | E_Const’ : eval_Const’ n (Nval n).

Inductive eval_Plus’ t1 t2 : val → Prop :=
  | E_Plus’ : ∀ n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval_Plus’ t1 t2 (Nval (n1 + n2)).

e : eval (Te_const 1) v
=-------------------------------
v = Nval 1

destruct (eval_eval’ e).

e : eval (Te_plus (Te_const 1) (Te_const 0)) v
=--------------------------------------------------
v = Nval 1

destruct (eval_eval’ e) as [n1 n2 e1 e2].

No additional equality
Inductive eval_Const’ n : val → Prop :=
  | E_Const’ : eval_Const’ n (Nval n).

Inductive eval_Plus’ t1 t2 : val → Prop :=
  | E_Plus’ : ∀ n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval_Plus’ t1 t2 (Nval (n1 + n2)).

e : eval (Te_const 1) v

v = Nval 1

destruct (eval_eval’ e).

e : eval (Te_plus (Te_const 1) (Te_const 0)) v

v = Nval 1

destruct (eval_eval’ e) as [n1 n2 e1 e2].

No additional equality
Inductive eval_Const' n : val → Prop :=
  | E_Const' : eval_Const' n (Nval n).

Inductive eval_Plus' t1 t2 : val → Prop :=
  | E_Plus' : ∀ n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval_Plus' t1 t2 (Nval (n1 + n2)).

e : eval (Te_const 1) v

v = Nval 1

destruct (eval_eval' e).

e : eval (Te_plus (Te_const 1) (Te_const 0)) v

v = Nval 1

destruct (eval_eval' e) as [n1 n2 e1 e2].

No additional equality
Inductive eval_Const’ n : val → Prop :=
  | E_Const’ : eval_Const’ n (Nval n).

Inductive eval_Plus’ t1 t2 : val → Prop :=
  | E_Plus’ : ∀ n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval_Plus’ t1 t2 (Nval (n1 + n2)).

\[
\begin{align*}
  e & : \text{eval } (\text{Te_const } 1) v \\
  v & = \text{Nval } 1 \\
  \text{destruct } (\text{eval_eval’ e}).
\end{align*}
\]
\[
\begin{align*}
  e & : \text{eval } (\text{Te_plus } (\text{Te_const } 1) (\text{Te_const } 0)) v \\
  v & = \text{Nval } 1 \\
  \text{destruct } (\text{eval_eval’ e}) \text{ as } [n1 n2 e1 e2].
\end{align*}
\]

No additional equality
Inductive `eval_Const' n : val → Prop :=
  | E_Const' : eval_Const' n (Nval n).

Inductive `eval_Plus' t1 t2 : val → Prop :=
  | E_Plus' : ∀ n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval_Plus' t1 t2 (Nval (n1 + n2)).

e : eval (Te_const 1) v
=================================
v = Nval 1

destruct (eval_eval' e).

e : eval (Te_plus (Te_const 1) (Te_const 0)) v
==============================================
v = Nval 1

destruct (eval_eval' e) as [n1 n2 e1 e2].

NO ADDITIONAL EQUALITY
Inductive eval(Const)_1_2 n : nat → Prop :=
  | E_Const’’ : eval(Const)_1_2 n n.

Inductive eval(Plus)_1_2 t1 t2 : nat → Prop :=
  | E_Plus’’ : ∀ n1 n2,
    eval t1 (Nval n1) → eval t2 (Nval n2) →
    eval(Plus)_1_2 t1 t2 (n1 + n2).

Definition eval_1_2 : te → val → Prop := fun t v =>
  match t, v with
  | Te_const c, Nval n => eval(Const)_1_2 c n
  | Te_plus t1 t2, Nval n => eval(Plus)_1_2 t1 t2 n
  | _, _ => False
end.

Definition eval_eval_1_2 {t v} : eval t v → eval_1_2 t v :=
  fun e =>
    match e with
    | E_Const n => E_Const’’ n
    | E_Plus t1 t2 n1 n2 e1 e2 => E_Plus’’ t1 t2 n1 n2 e1 e2
    end.
Small inversions V1, how/why it works

Separation of concerns

The usually complicated pattern-matching working on $R$ is decomposed and isolated in $\text{rel}'$ and $\text{rel}_{\text{rel}}$

Pattern-matching is very powerful in Type Theory

- Relevant bindings
  automatically performed in the course of pattern-matching
- A single pattern-matching
  $=$ multiple simultaneous rewrite steps for free
- No additional rewrite in scripts
- Using equalities and rewrite $=$ complications $+$ steps backwards
Small inversions V1, how/why it works

Separation of concerns
The usually complicated pattern-matching working on $R$ is decomposed and isolated in $\text{rel}'$ and $\text{rel}_{\text{rel}}'$

Pattern-matching is very powerful in Type Theory
- Relevant bindings automatically performed in the course of pattern-matching
- A single pattern-matching = multiple simultaneous rewrite steps for free
- No additional rewrite in scripts
- Using equalities and rewrite = complications + steps backwards
\[ \forall v, P(v) \rightarrow \text{eval}(\text{Te\_const}\ 1)\ v \rightarrow v = \text{Nval}\ 1. \]

**Small inversions V1**

\[
\begin{align*}
& (\text{fun } (v : \text{val}) (p : P\ v) (e : \text{eval}\ (\text{Te\_const}\ 1)\ v) \Rightarrow \\
& (\text{let } e0 : \text{eval}\_1\ (\text{Te\_const}\ 1)\ v := \text{eval}\_\text{eval}\_1\ e \text{ in} \\
& \text{match } e0 \text{ in } (\text{eval}\_\text{Const}\_1\ _ \ v0) \text{ return } (\text{eval}\ (\text{Te\_const}\ 1)\ v0 \rightarrow P\ v0 \rightarrow v0 = \text{Nval}\ 1) \text{ with} \\
& \mid \text{E\_Const}'\ _ = \Rightarrow \text{fun } (_ : \text{eval}\ (\text{Te\_const}\ 1)\ (\text{Nval}\ 1)) (_ : P\ (\text{Nval}\ 1)) \Rightarrow \text{eq}\_\text{refl} \\
& \text{end } e)\ p) \\
\end{align*}
\]

**Coq inversion (2021)**

\[
\begin{align*}
& (\text{fun } (v : \text{val}) (_ : P\ v) (e : \text{eval}\ (\text{Te\_const}\ 1)\ v) \Rightarrow \\
& \text{let } H : \text{Te\_const}\ 1 = \text{Te\_const}\ 1 \rightarrow v = v \rightarrow v = \text{Nval}\ 1 := \\
& \text{match } e \text{ in } (\text{eval}\ t\ v0) \text{ return } (t = \text{Te\_const}\ 1 \rightarrow v0 = v \rightarrow v = \text{Nval}\ 1) \text{ with} \\
& \mid \text{E\_Const}\ n = \Rightarrow \\
& \hspace{1cm} \text{fun } (H : \text{Te\_const}\ n = \text{Te\_const}\ 1) \ (H0 : \text{Nval}\ n = v) \Rightarrow \\
& \hspace{1cm} (\text{fun } H1 : \text{Te\_const}\ n = \text{Te\_const}\ 1 \Rightarrow \\
& \hspace{2cm} \text{let } H2 : n = 1 := \\
& \hspace{3cm} \text{f\_equal } (\text{fun } e0 : \text{te} \Rightarrow \text{match } e0 \text{ with} \\
& \hspace{4cm} \mid \text{Te\_const}\ n0 = \Rightarrow n0 \\
& \hspace{4cm} \mid \text{Te\_plus}\ _\ _ = \Rightarrow n \\
& \hspace{3cm} \text{end } H1 \text{ in} \\
& \hspace{1cm} (\text{fun } H3 : n = 1 \Rightarrow \\
& \hspace{2cm} \text{let } H4 : n = 1 := H3 \text{ in} \\
& \hspace{3cm} \text{eq\_ind\_r } (\text{fun } n0 : \text{nat} \Rightarrow \text{Nval}\ n0 = v \rightarrow v = \text{Nval}\ 1) \\
& \hspace{2cm} (\text{fun } H5 : \text{Nval}\ 1 = v = \\
& \hspace{3cm} \text{let } H6 : \text{Nval}\ 1 = v := H5 \text{ in} \\
& \hspace{4cm} \text{eq\_ind } (\text{Nval}\ 1) (\text{fun } v0 : \text{val} \Rightarrow v0 = \text{Nval}\ 1) \text{ eq\_refl } v H6 H4 H2 H0) H0)
\end{align*}
\]
∀ v, P v -> eval (Te_const 1) v -> v = Nval 1.

**Small inversions V1**

(fun (v : val) (p : P v) (e : eval (Te_const 1) v) =>
(let e0 : eval_1 (Te_const 1) v := eval_eval_1 e in
 match e0 in (eval_Const_1 _ v0) return (eval (Te_const 1) v0 -> P v0 -> v0 = Nval 1) with
 | E_Const' _ => fun (_ : eval (Te_const 1) (Nval 1)) (_ : P (Nval 1)) => eq_refl
 end e) p)

**Coq inversion (2021)**

(fun (v : val) (_ : P v) (e : eval (Te_const 1) v) =>
 let H : Te_const 1 = Te_const 1 -> v = v -> v = Nval 1 :=
 match e in (eval t v0) return (t = Te_const 1 -> v0 = v -> v = Nval 1) with
 | E_Const n =>
   fun (H : Te_const n = Te_const 1) (H0 : Nval n = v) =>
     (fun H1 : Te_const n = Te_const 1 =>
      let H2 : n = 1 :=
       f_equal (fun e0 : te => match e0 with
                | Te_const n0 => n0
                | Te_plus _ _ => n
               end) H1 in
       eq_ind_r (fun n0 : nat => Nval n0 = v -> v = Nval 1)
       (fun H5 : Nval 1 = v =>
        let H6 : Nval 1 = v := H5 in
        eq_ind (Nval 1) (fun v0 : val => v0 = Nval 1) eq_refl v H6) H4) H2) H H0)
(fun (v : val) (_, : P v) (e : eval (Te_const 1) v) =>
  let H : Te_const 1 = Te_const 1 -> v = v -> v = Nval 1 :=
    match e in (eval t v0) return (t = Te_const 1 -> v0 = v -> v = Nval 1) with
    | E_Const n =>
      fun (H : Te_const n = Te_const 1) (H0 : Nval n = v) =>
        (fun H1 : Te_const n = Te_const 1 =>
          let H2 : n = 1 :=
            f_equal (fun e0 : te => match e0 with
              | Te_const n0 => n0
              | Te_plus _ _ => n
            end) H1 in
            (fun H3 : n = 1 =>
              let H4 : n = 1 := H3 in
              eq_ind_r (fun n0 : nat => Nval n0 = v -> v = Nval 1)
                (fun H5 : Nval 1 = v =>
                  let H6 : Nval 1 = v := H5 in
                  eq_ind (Nval 1) (fun v0 : val => v0 = Nval 1) eq_refl v H6) H4) H2) H0
        | E_Plus t1 t2 n1 n2 H H0 =>
          fun (H1 : Te_plus t1 t2 = Te_const 1) (H2 : Nval (n1 + n2) = v) =>
            (fun H3 : Te_plus t1 t2 = Te_const 1 =>
              let H4 : False :=
                eq_ind (Te_plus t1 t2)
                (fun e0 : te => match e0 with
                  | Te_const _ => False
                  | Te_plus _ _ => True
                end) I (Te_const 1) H3 in
              False_ind (Nval (n1 + n2) = v -> eval t1 (Nval n1) -> eval t2 (Nval n2) -> v = Nval 1)
                H4) H1 H2 H H0
        end in
    H eq_refl eq_refl)
Full V0 can be seen as a purely functional translation of V1

Continuation Passing Style / polymorphic lambda-calculus
Bounded natural numbers – or finite sets \( t \, n \) of size \( n \)

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

Even bounded numbers

Inductive \( \text{even} \) : \( \forall \{n\}, \, t \, n \rightarrow \text{Prop} \) :=
   | even_0 \(\{n\} \) : \((@\text{FO} \, n) \)
   | even_SS \(\{n\} \) \(i : t \, n \) : \(\text{even} \, i \rightarrow \text{even} \,(FS \,(FS \, i))\).

Issues on lemmas such as

\[ \forall n \, (i : t \, n), \, \text{even} \,(FS \,(FS \, i)) \rightarrow \text{even} \, i. \]
\[ \forall n \, m \, (i : t \, n) \, (j : t \, m), \]
\[ \text{even} \,(F\text{plus} \, i \, j) \rightarrow \text{even} \, i \rightarrow \text{even} \, j. \]
Bounded natural numbers – or finite sets $t \, n$ of size $n$

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

Even bounded numbers

Inductive $\text{even} : \forall \{n\}, \, t \, n \rightarrow \text{Prop}$ :=
| $\text{even}_0 \, \{n\} : \text{even} \, (@\text{FO} \, n)$
| $\text{even}_\text{SS} \, \{n\} \, (i : t \, n) : \text{even} \, i \rightarrow \text{even} \, (\text{FS} \, (\text{FS} \, i))$.

Issues on lemmas such as

$\forall \, n \, (i : t \, n), \text{even} \, (\text{FS} \, (\text{FS} \, i)) \rightarrow \text{even} \, i$.

$\forall \, n \, m \, (i : t \, n) \, (j : t \, m)$,
\[ \text{even} \, (\text{Fplus} \, i \, j) \rightarrow \text{even} \, i \rightarrow \text{even} \, j \]
Bounded natural numbers – or finite sets $t \ n$ of size $n$

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

Even bounded numbers

Inductive even : forall $\{n\}$, $t \ n \rightarrow \text{Prop}$ :=
\[
\begin{align*}
| \text{even}_0 \ {n} : \quad \text{even} \ (@\text{FO} \ n) \\
| \text{even_SS} \ {n} \ (i: t \ n) : \text{even} \ i \rightarrow \text{even} \ (FS \ (FS \ i)).
\end{align*}
\]

Issues on lemmas such as
\[
\begin{align*}
\forall \ n \ (i: t \ n), \quad \text{even} \ (FS \ (FS \ i)) & \rightarrow \text{even} \ i. \\
\forall \ n \ m \ (i: t \ n) \ (j: t \ m), \quad \\
\text{even} \ (Fplus \ i \ j) \rightarrow \text{even} \ i \rightarrow \text{even} \ j.
\end{align*}
\]
Inductive even0 : Prop := | even_0' : even0.

Inductive evenSS {n} (i: t n) : Prop := | even_SS' : even i → evenSS i.

Definition even' : ∀ {n}, t n → Prop := fun n i =>
  match i with
  | FO => even0
  | FS (FS i) => evenSS i
  | _ => ⊥
  end.

Definition even_even' {n} {i: t n} (e : even i) : even' i :=
  match e with
  | even_0 => even_0'
  | even_SS i e => even_SS' i e
  end.
Inductive even0: Prop :=
| even_0': even0.

Inductive evenSS {n} (i: t n) : Prop :=
| even_SS': even i \rightarrow evenSS i.

Definition even': \forall {n}, t n \rightarrow Prop := fun n i =>
match i with
| FO => even0
| FS (FS i) => evenSS i
| _ => ⊥
end.

Definition even_even': {n} {i: t n} (e : even i) : even' i :=
match e with
| even_0 => even_0'
| even_SS i e => even_SS' i e
end.
Inductive even0 : Prop :=
  | even_0' : even0.

Inductive evenSS {n} (i : t n) : Prop :=
  | even_SS' : even i → evenSS i.

Definition even' : ∀ {n}, t n → Prop := fun n i =>
  match i with
  | FO => even0
  | FS (FS i) => evenSS i
  | _ => ⊥
end.

Definition even_even’ {n} {i: t n} (e : even i) : even’ i :=
  match e with
  | even_0 => even_0’
  | even_SS i e => even_SS’ i e
end.
Fixpoint lift1 m {n} (i : t n) : t (m + n) :=
  match i in t n return t (m + n) with
  | FO => t_n_Sm FO
  | FS i => t_n_Sm (FS (lift1 m i))
end.

Fixpoint Fplus {n m : nat} (i : t n) (j : t m) : t (n + m) :=
  match i with
  | @FO n => lift1 (S n) j
  | FS i => FS (Fplus i j)
end.
Fixpoint lift1 m {n} (i : t n) : t (m + n) :=
  match i in t n return t (m + n) with
  | FO => t_n_Sm FO
  | FS i => t_n_Sm (FS (lift1 m i))
end.

Fixpoint Fplus {n m : nat} (i : t n) (j : t m) : t (n + m) :=
  match i with
  | @FO n => lift1 (S n) j
  | FS i => FS (Fplus i j)
end.
i : t n ; j : t m
eij : even (FS (FS (Fplus i j)))
ei : even i
IHei : even (Fplus i j) → even j

destruct (even_even' eij) as [eij']
eij' : even (Fplus i j)

Coq inversion eij; subst
(* FAILURE! *)
i : t n; j : t m
eij : even (FS (FS (Fplus i j)))
ei : even i
IHei : even (Fplus i j) → even j

Coq inversion eij; subst  (* FAILURE! *)

i0 : t (n + m)
H1 : even i0
H0 : existT (fun n : nat => t n) (n + m) i0 =
    existT (fun n : nat => t n) (n + m) (Fplus i j)

-------------------------------
even j

destruct (even_even' eij) as [eij']
eij' : even (Fplus i j)
-------------------------------
even j
i : t n; j : t m

eij : even (FS (FS (Fplus i j)))

ei : even i

IHei : even (Fplus i j) → even j

Coq inversion eij; subst (* FAILURE! *)

i0 : t (n + m)

H1 : even i0

H0 : existT (fun n : nat => t n) (n + m) i0 =

existT (fun n : nat => t n) (n + m) (Fplus i j)

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

even j

destruct (even_even' eij) as [eij']

eij' : even (Fplus i j)

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

even j
Beating Coq inversion: on dependent types (4/4)

\[
i \colon t \ n \quad j \colon t \ m
\]
\[e_{ij} : \text{even } (FS (FS (Fplus \ i \ j)))\]
\[e_i : \text{even } i\]
\[\text{IHei} : \text{even } (Fplus \ i \ j) \rightarrow \text{even } j\]

Coq inversion \(e_{ij}\); subst (* FAILURE! *)

\[
i_0 : t \ (n + m)
\]
\[H1 : \text{even } i_0\]
\[H0 : \text{existT } (\text{fun } n : \text{nat} \Rightarrow t \ n) \ (n + m) \ i_0 =
\]
\[
\text{existT } (\text{fun } n : \text{nat} \Rightarrow t \ n) \ (n + m) \ (Fplus \ i \ j)
\]

\[\text{even } j\]

destruct (even\_even' \(e_{ij}\)) as [\(e_{ij}'\)]

\[
e_{ij}' : \text{even } (Fplus \ i \ j)
\]

\[\text{even } j\]
Small inversions V1

(fun (n m : nat) (i : t n) (j : t m) (eij : even (Fplus i j)) (ei : even i) =>
  even_ind (fun (n0 : nat) (i0 : t n0) => even (Fplus i0 j) -> even j)
  (fun (n0 : nat) (eij0 : even (Fplus FO j)) => even_lift1 (S n0) eij0)
  (fun (n0 : nat) (i0 : t n0) (ei0 : even i0) (IHei : even (Fplus i0 j) -> even j)
    (eij0 : even (Fplus (FS (FS i0)) j)) =>
    let e : even' (FS (FS (Fplus i0 j))) := even_even' eij0 in
    match e with
    | even_SS' _ eij' => _
  end) n i ei eij)

Coq inversion (2021)

(fun (n m : nat) (i : t n) (j : t m) (eij : even (Fplus i j)) (ei : even i) =>
  even_ind (fun (n0 : nat) (i0 : t n0) => even (Fplus i0 j) -> even j)
  (fun (n0 : nat) (eij0 : even (Fplus FO j)) => even_lift1 (S n0) eij0)
  (fun (n0 : nat) (i0 : t n0) (ei0 : even i0) (IHei : even (Fplus i0 j) -> even j)
    (eij0 : even (Fplus (FS (FS i0)) j)) =>
    let H :
      S (S (n0 + m)) = S (S (n0 + m)) ->
      existT (fun n1 : nat => t n1) (S (S (n0 + m))) (FS (FS (Fplus i0 j))) =
      existT (fun n1 : nat => t n1) (S (S (n0 + m))) (FS (FS (Fplus i0 j))) ->
      even j :=
      match
      eij0 in (@even n1 t0)
      return
      (n1 = S (S (n0 + m)) ->
       existT (fun n2 : nat => t n2) n1 t0 =
Small inversions V1

(fun (n m : nat) (i : t n) (j : t m) (eij : even (Fplus i j)) (ei : even i) =>
  even_ind (fun (n0 : nat) (i0 : t n0) => even (Fplus i0 j) -> even j)
  (fun (n0 : nat) (eij0 : even (Fplus F0 j)) => even_lift1 (S n0) eij0)
  (fun (n0 : nat) (i0 : t n0) (e0 : even i0) (IHei : even (Fplus i0 j) -> even j)
    (eij0 : even (Fplus (FS (FS i0)) j)) =>
    let e : even’ (FS (FS (Fplus i0 j))) := even_even’ eij0 in
    match e with
    | even_SS’ _ eij’ => _
    end) n i ei eij)

Coq inversion (2021)

(fun (n m : nat) (i : t n) (j : t m) (eij : even (Fplus i j)) (ei : even i) =>
  even_ind (fun (n0 : nat) (i0 : t n0) => even (Fplus i0 j) -> even j)
  (fun (n0 : nat) (eij0 : even (Fplus F0 j)) => even_lift1 (S n0) eij0)
  (fun (n0 : nat) (i0 : t n0) (e0 : even i0) (IHei : even (Fplus i0 j) -> even j)
    (eij0 : even (Fplus (FS (FS i0)) j)) =>
    let H :
      S (S (n0 + m)) = S (S (n0 + m)) ->
      existT (fun n1 : nat => t n1) (S (S (n0 + m))) (FS (FS (Fplus i0 j))) =
      existT (fun n1 : nat => t n1) (S (S (n0 + m))) (FS (FS (Fplus i0 j))) ->
      even j :=
      match
        eij0 in (@even n1 t0)
      return
        (n1 = S (S (n0 + m)) ->
         existT (fun n2 : nat => t n2) n1 t0 =
existT (fun n2 : nat => t n2) (S (S (n0 + m))) (FS (FS (Fplus i0 j))) ->
even j)

with
| @even_0 n1 =>
  fun (H : S n1 = S (S (n0 + m)))
  (H0 : existT (fun n2 : nat => t n2) (S n1) FO =
  existT (fun n2 : nat => t n2) (S (S (n0 + m))) (FS (FS (Fplus i0 j)))) =>
  (fun H1 : S n1 = S (S (n0 + m)) =>
   let H2 : n1 = S (n0 + m) :=
   f_equal (fun e : nat => match e with
     | 0 => n1
     | S n2 => n2
   end) H1 in
   (fun H3 : n1 = S (n0 + m) =>
    let H4 : n1 = S (n0 + m) := H3 in
    eq_ind_r
    (fun n2 : nat =>
     existT (fun n3 : nat => t n3) (S n2) FO =
     existT (fun n3 : nat => t n3) (S (S (n0 + m))) (FS (FS (Fplus i0 j))) =>
     even j)
    (fun
     H5 : existT (fun n2 : nat => t n2) (S (S (n0 + m))) FO =
     existT (fun n2 : nat => t n2) (S (S (n0 + m))) (FS (FS (Fplus i0 j))) =>
     let H6 : False :=
     eq_ind (existT (fun n2 : nat => t n2) (S (S (n0 + m))) FO)
     (fun e : n2 : nat & t n2 =>
      let (x, t0) := e in match t0 with
        | FO => True
        | FS _ => False
      end) I
     (existT (fun n2 : nat => t n2) (S (S (n0 + m))) (FS (FS (Fplus i0 j)))) H5 in
     False_ind (even j) H6) H4) H2) H H0)
Coq inversion viz small inversions V1 (3/4)

| @even_SS n1 i1 H =>
| fun (H0 : S (S n1) = S (S (n0 + m)))
| (H1 : existT (fun n2 : nat => t n2) (S (S n1)) (FS (FS i1)) =
| existT (fun n2 : nat => t n2) (S (S (n0 + m))) (FS (FS (Fplus i0 j)))) =>
| (fun H2 : S (S n1) = S (S (n0 + m)) =>
| let H3 : n1 = n0 + m :=
| f_equal (fun e : nat => match e with
| | S (S n3) => n3
| | _ => n1
| end) H2 in
| (fun H4 : n1 = n0 + m =>
| (let H5 : n1 = n0 + m := H4 in
| eq_ind_r
| (fun n2 : nat =>
| forall i2 : t n2,
| existT (fun n3 : nat => t n3) (S (S n2)) (FS (FS i2)) =
| existT (fun n3 : nat => t n3) (S (S (n0 + m))) (FS (FS (Fplus i0 j))) =>
| even i2 -> even j)
| (fun (i2 : t (n0 + m))
| (H6 : existT (fun n2 : nat => t n2) (S (S (n0 + m))) (FS (FS i2)) =
| existT (fun n2 : nat => t n2) (S (S (n0 + m))) (FS (FS (Fplus i0 j)))) =>
| let H7 :
| existT (fun n2 : nat => t n2) (n0 + m) i2 =
| existT (fun n2 : nat => t n2) (n0 + m) (Fplus i0 j) :=
| f_equal
| (fun e : n2 : nat & t n2 =>
| let (x, t0) := e in
| match t0 with
| | F0 => existT (fun n3 : nat => t n3) (n0 + m) i2
| | FS F0 => existT (fun n4 : nat => t n4) (n0 + m) i2
| | FS (@FS n3 t2) => existT (fun n4 : nat => t n4) n3 t2
| | _ => true
| end) H2 H6 H7 in

J-F. Monin  
small inversions & recursion  
June 2021 49 / 71
end) H6 in
(fun
  (H8 : existT (fun n2 : nat => t n2) (n0 + m) i2 =
   existT (fun n2 : nat => t n2) (n0 + m) (Fplus i0 j))
  (H9 : even i2) =>
  _ ) H7) H5) i1)
H3) H0 H1 H
end in
H eq_refl eq_refl) n i ei eij)
Until this slide

- V0 (2010-2013): light but headache maker
  pseudo-impredicative auxiliary definition pattern-matching on types
- V1: easier
  auxiliary inductive + pattern-matching

Using inversions in recursive programs, next slides

Issue solved with V2 & V3: \((\text{pattern-matching})^n\)
Recursive programs

Number of steps

Given

- A function $g : X \to X$
- A halting test function $b : X \to \text{bool}$
- An initial value $x : X$

Compute the minimum $n$ such that $b (g^n x) = \text{true}$

let rec ns x = if $b$ x then 0 else 1 + ns (g x)

let rec nsa x n = if $b$ x then n else nsa (g x) (1 + n)

Equivalent ?

Does $\text{nsa} x 0$ always return the same value as $\text{ns} x$?
Looks ridiculously impossible in Coq

- Write Coq programs for \texttt{ns} and \texttt{nsa}
- Such that they are extracted exactly as expected
- Reason about them

**Issue**

No clue about the (identical) termination of \texttt{ns} and \texttt{nsa}
Braga method, 1st idea: inductive domain + projections

Inductive characterization of the domain of ns and nsa

Inductive \( \mathbb{D}_{ns} (x : X) : \text{Prop} \) :=
| \( \mathbb{D}_{ns\_tr} : \text{b x = true} \rightarrow \mathbb{D}_{ns} x \)
| \( \mathbb{D}_{ns\_fa} : \text{b x = false} \rightarrow \mathbb{D}_{ns} (g x) \rightarrow \mathbb{D}_{ns} x \).

Target

Fixpoint \( \text{fct} x (D : \mathbb{D}_{ns} x) \{ \text{struct D} \} : \text{N} := \)
match \( \text{b x} \) with
| true \( \Rightarrow \ldots \)
| false \( \Rightarrow \ldots \text{fct (g x) (proj D)} \ldots \)
end.

With \( (\text{proj D}) < D \)
Requirement on the projection

Inductive characterization of the domain of \( \text{ns} \) and \( \text{nsa} \)

\[
\text{Inductive } \text{Dns} (x: X) : \text{Prop} := \\
\mid \text{Dns\_tr} : \text{b x = true } \rightarrow \text{Dns} x \\
\mid \text{Dns\_fa} : \text{b x = false } \rightarrow \text{Dns} (g x) \rightarrow \text{Dns} x.
\]

\((\text{proj } D)\) defined for all \( x \) such that \( \text{b x = false} \)

Definition \( \text{prj\_Dns} x \) (\( E : \text{b x = false} \)) (\( D : \text{Dns} x \)): \( \text{Dns} (g x) := \)
\[
\text{match } D \text{ with} \\
\mid \text{Dns\_tr } _ E' \Rightarrow \text{match } \text{false\_true} E E' \text{ with end} \\
\mid \text{Dns\_fa } _ _ D \Rightarrow D \\
\text{end}.
\]

Lemma \( \text{false\_true} x : x = \text{false } \rightarrow x = \text{true } \rightarrow \bot \).
Recursive programs using Trojan horses

Fixpoint ns x (D: Dns x) : nat :=
  match b x as bx return b x = bx → _ with
  | true => λ E, 0
  | false => λ E, S (ns (g x) (prj_Dns E D))
end eq_refl.

Fixpoint nsa x (n: nat) (D: Dns x) : nat :=
  match b x as bx return b x = bx → _ with
  | true => λ E, n
  | false => λ E, nsa (g x) (S n) (prj_Dns E D)
end eq_refl.
Fixpoint \( \text{ns} \ x \ (D: \text{Dns} \ x) : \text{nat} := \)
\[
\text{match } b \ x \ \text{as } bx \ \text{return } b \ x = bx \to _\ \text{with}
\mid \text{true } => \lambda \ E, 0
\mid \text{false } => \lambda \ E, S (\text{ns} \ (g \ x) \ (\text{prj}_D \text{Dns} \ E \ D))
\text{end eq_refl.}
\]

Fixpoint \( \text{nsa} \ x \ (n: \text{nat}) \ (D: \text{Dns} \ x) : \text{nat} := \)
\[
\text{match } b \ x \ \text{as } bx \ \text{return } b \ x = bx \to _\ \text{with}
\mid \text{true } => \lambda \ E, n
\mid \text{false } => \lambda \ E, \text{nsa} \ (g \ x) \ (S \ n) \ (\text{prj}_D \text{Dns} \ E \ D)
\text{end eq_refl.}
\]
2nd idea of the Braga method: input-output graph

\[ \text{Inductive } \mathbb{G}_\text{ns} (x: X) : \text{nat} \to \text{Prop} := \]
\[ | \text{in}_\text{grns}_0 : b x = \text{true} \to x \mapsto \text{ns} \ 0 \]
\[ | \text{in}_\text{grns}_1 o : b x = \text{false} \to g x \mapsto \text{ns} \ 0 \to x \mapsto \text{ns} \ S \ o \]
where "\( x \mapsto \text{ns} \ o \)" := (\( \mathbb{G}_\text{ns} \ x \ o \)).

\[ \text{Fixpoint } \text{ns}_\text{pwc} \ x (D: \mathbb{D}_\text{ns} \ x) : \{ o | x \mapsto \text{ns} \ o \}. \]
\[ \text{Proof. refine(} \]
\[ \quad \text{match } b \ x \text{ as } bx \text{ return } b \ x = bx \to _{\_} \text{ with} \]
\[ \quad | \text{true} \to \lambda E, \text{exist } _{\_} 0 _ {\_} \]
\[ \quad | \text{false} \to \lambda E, \text{let } (o,Go) := \text{ns}_\text{pwc} (g \ x) (\text{prj}_\mathbb{D}_\text{ns} \ E \ D)\]
\[ \quad \quad \text{in } \text{exist } _{\_} (S \ o) _ {\_} \]
\[ \quad \text{end eq_refl)}. \]
\[ - \text{ constructor 1; exact } E. \]
\[ - \text{ constructor 2; assumption.} \]
\[ \text{Defined}. \]
Remarks about inversion

- \texttt{prj_{Dns}} is a special case of inversion

- The previous inversion technique does not provide structurally smaller terms: the different components of a constructor have to be recovered one by one.

- It can still be used here, in order to prove that \texttt{Gns} is deterministic.

- Coq automated inversion provide structurally smaller terms!
  
  ...when it works.
Remarks about inversion

- prj_Dns is a special case of inversion
- The previous inversion technique does not provide structurally smaller terms: the different components of a constructor have to be recovered one by one
- It can still be used here, in order to prove that Gns is deterministic
- Coq automated inversion provide structurally smaller terms! ...
  ...when it works
The previous example (ns and nsa) happens to be closer to the LISP style. The guard used as a Trojan horse is typically an equality to be used to get \( \bot \) in absurd cases.

For ML style programs, we can instead use Trojan horses based on \( \bot \) and \( \top \), which can be directly exploited. See below a typical programming pattern.
Reasoning on fold_left

**Functional specification**

``` ocaml
let rec foldl_ref l = match l with (* fake *)
| []    → b0
| u :+: z → f (foldl_ref u) z
```

**Inductive specification**

\[
\begin{align*}
\text{u} & \mapsto \text{fl} \ b \\
\text{[]} & \mapsto \text{fl} \ b_0 \\
\text{u} :+: \text{z} & \mapsto \text{fl} \ f \ b \ z
\end{align*}
\]
Reasoning on fold_left

Functional specification

```ocaml
let rec foldl_ref l = match l with (* fake *)
| [] → b0
| u :+: z → f (foldl_ref u) z
```

Inductive specification

\[
\begin{align*}
\text{[]} & \xrightarrow{\text{fl}} b0 \\
\text{u :+: z} & \xrightarrow{\text{fl}} f (\text{foldl_ref u}) z
\end{align*}
\]
Reasoning on fold_left

Specification

```plaintext
let rec foldl_ref l = match l with (* fake *)
| [] → b0
| u +: z → f (foldl_ref u) z
```

Regular program

```plaintext
type α lr = Nilr | Consr of α list * α

let rec foldl_ref l = match l2r l with
| Nilr → b0
| Consr (u, z) → f (foldl_ref u) z
```
Fixpoint foldl_pwc l (D : Dlz l) : \{ b \mid l \mapsto fl b\}.
Proof.

\text{gen\_help l G\_foldl ; apply up\_llP in D; revert D.}
\text{refine ( match l2r l with}
  \text{ | \text{\texttt{Nilr}}} \Rightarrow \lambda D \ T, \ \text{exist } _{b0 } _
  \text{ | \text{\texttt{Consr u z}}} \Rightarrow \lambda D \ T,
    \text{let \texttt{(b, Cb ) := foldl\_pwc u (\pi Dlz D)}
      \text{in \text{exist } _{(f b z) } _
end).}
- apply T ; constructor 1.
- apply T ; constructor 2; exact Cb .
Qed.
Correctness of standard fold_left

Easy

Use \( \text{foldl } f \ b \ (u \ +: \ z) = f \ (\text{foldl } f \ b \ u) \ z \)

No use of associativity of append
\((\text{append is not in the vocabulary})\)
Projection — Common programming pattern for ML style programs

\[ \text{Dlz} \ u \quad \text{Dlr} \ (l2r \ l) \]

\[ \text{Dlr} \ Nilr \quad \text{Dlr} \ (\text{Consr} \ u \ z) \quad \text{Dlz} \ l \]

*Projection for second rule*

Let \( \pi_\text{Dlr} \ \{u \ z\} \ (D: \text{Dlr} \ (\text{Consr} \ u \ z)) : \text{Dlz} \ u := \)

match \( D \) in \( \text{Dlr} \ r \) return

let \( g := \) match \( r \) with \( \text{Consr} \ u \ z \) => \( \top \) | _ => \( \bot \) end in

let \( u := \) match \( r \) with \( \text{Consr} \ u \ z \) => \( u \) | _ => \( u \) end in

\( g \rightarrow \text{Dlz} \ u \) with

| \( \text{Dlr}_\text{Consr} \ u \ z \ D \) => \( \lambda \ G, D \)
| _ => \( \lambda \ G, \) match \( G \) with end (* < D as well *)
end \( I \) (* proof of \( \top \) *).

The guard \( G:g \) filters the relevant shape.
The \( u \) component in the type of \( D \) has to be recovered
from \( r \) in the general type of \( D \).
The original \( u \) is just a light suitable default value for
this computation.
Projection – Common programming pattern for ML style programs

\[\begin{align*}
\text{D}lz \ u & \quad \text{D}lr \ (\text{l2r} \ 1) \\
\text{D}lr \ \text{Nilr} & \quad \text{D}lr \ (\text{Consr} \ u \ z)
\end{align*}\]

Projection for second rule

Let \(\pi_{\text{D}lr} \ {u \ z} \ (D: \text{D}lr \ (\text{Consr} \ u \ z)) : \text{D}lz \ u :=\)

\[
\begin{array}{l}
\text{match } D \text{ in } \text{D}lr \ r \text{ return } \\
\quad \text{let } g := \text{match } r \text{ with } \text{Consr} \ u \ z \Rightarrow \top | _ \Rightarrow \bot \text{ end in } \\
\quad \text{let } u := \text{match } r \text{ with } \text{Consr} \ u \ z \Rightarrow u | _ \Rightarrow u \text{ end in } \\
\quad g \rightarrow \text{D}lz \ u \text{ with } \\
\quad | \text{D}lr_{-\text{Consr}} \ u \ z \ D \Rightarrow \lambda \ G, \ D \\
\quad | _ \Rightarrow \lambda \ G, \ \text{match } G \text{ with end (* < } D \text{ as well *) end I (* proof of } \top \text{ *).}
\end{array}
\]

The guard \(G:g\) filters the relevant shape.

The \(u\) component in the type of \(D\) has to be recovered from \(r\) in the general type of \(D\).

The original \(u\) is just a light suitable default value for this computation.
Projection – Common programming pattern for ML style programs

\[ \begin{array}{c}
\text{Projection for second rule} \\
\text{Let } \pi_{\mathcal{D}lr} \{u \, z\} (D: \mathcal{D}lr (\text{Consr} \, u \, z)) : \mathcal{D}lz \, u := \\
\text{match } D \text{ in } \mathcal{D}lr \, r \text{ return} \\
\quad \text{let } g := \text{match } r \text{ with } \text{Consr} \, u \, z \Rightarrow \top \mid _\_ \Rightarrow \bot \text{ end in} \\
\quad \text{let } u := \text{match } r \text{ with } \text{Consr} \, u \, z \Rightarrow u \mid _\_ \Rightarrow u \text{ end in} \\
\quad g \rightarrow \mathcal{D}lz \, u \text{ with} \\
\quad | \mathcal{D}lr\_\text{Consr} \, u \, z \, D \Rightarrow \lambda G, D \\
\quad | _\_ \Rightarrow \lambda G, \text{match } G \text{ with end } (* < D \text{ as well } *) \\
\text{end I } (* \text{ proof of } \top \, *) .
\end{array} \]

The guard \( G:g \) filters the relevant shape.

The \( u \) component in the type of \( D \) has to be recovered from \( r \) in the general type of \( D \).

The original \( u \) is just a light suitable default value for this computation.
Projection – Common programming pattern for ML style programs

Projection for second rule

Let \( \pi_{Dlr} \{u \; z\} \ (D : Dlr \ (Consr \ u \ z)) \ : \ Dlz \ u \ := \)

match \( D \) in \( Dlr \ r \) return
  let \( g \) := match \( r \) with Consr \( u \; z \) => \( \top \) | _ => \( \bot \) end in
  let \( u \) := match \( r \) with Consr \( u \; z \) => \( u \) | _ => \( u \) end in
  \( g \rightarrow Dlz \ u \) with
  | \( Dlr\_Consr \ u \; z \; D \) => \( \lambda \; G, \; D \)
  | _ => \( \lambda \; G, \) match \( G \) with end (* < D as well *)
end I (* proof of \( \top \) *).

*The guard \( G : g \) filters the relevant shape.*

*The \( u \) component in the type of \( D \) has to be recovered from \( r \) in the general type of \( D \).*

*The original \( u \) is just a light suitable default value for this computation.*
Limitation of the previous trick

Need for a default value in functions such as pred, tail, or the inlined function of previous slide:

```
let u := match r with Cons r u z => u | _ => u end in
```

Fortunately, something like the original `u` on previous slide is always available when dealing with usual (non-dependent) algebraic types.

Provides a cheap solution.

This trick is used in Coq automated inversion.

But it is no longer the case with inductive families, such as bounded natural numbers above, vectors, etc.
Limitation of the previous trick

Need for a default value in functions such as pred, tail, or the inlined function of previous slide:

\[
\text{let } u ::= \text{match } r \text{ with } \text{Cons } u \ z \Rightarrow u \mid _{} \Rightarrow u \text{ end } in
\]

Fortunately, something like the original \(u\) on previous slide is always available when dealing with usual (non-dependent) algebraic types.

Provides a cheap solution.

This trick is used in Coq automated inversion.

But it is no longer the case with inductive families, such as bounded natural numbers above, vectors, etc.
Other approaches

Ad-hoc

0 (or F0) for (bounded) nats
The simplest thing to do in handcrafted approaches

General

match something-reducing-to-a-proof-of-⊥ with end

Suspicious subsingleton elimination, should be avoided

Can be circumvented using loops
Other approaches

Ad-hoc

0 (or F0) for (bounded) nats
The simplest thing to do in handcrafted approaches

General

match something-reducing-to-a-proof-of-⊥ with end

Suspicious subsingleton elimination, should be avoided

Can be circumvented using loops
Fixpoint half n (i: t n) (D: even i) {struct D} : nat := 
  match i with 
  | FO   => λ D, 0 
  | FS i => match i return even (FS i) -> nat with 
  | FO   => λ D, Fexc (even_even' D) 
  | FS i => λ D, S (half i (π_even D)) 
  end 
end D.

Definition π_even {n} {i: t n} (D: even (FS (FS i))) : even i := 
  match D in even j return ∀ G: shape j, even (fpred2 j G) with 
  | even_SS i e => λ G, e 
  | _ => λ G, match G with end 
end I.
Fixpoint \( \text{half} \ n \ (i: \ t \ n) \ (D: \ \text{even} \ i) \ {\text{struct} \ D} : \ \text{nat} \) :=
\[
\begin{align*}
\text{match} \ i \ \text{with} \\
| \ \text{FO} & \Rightarrow \lambda \ D, 0 \\
| \ \text{FS} \ i & \Rightarrow \\
\quad \text{match} \ i \ \text{return} \ \text{even} \ (\text{FS} \ i) \rightarrow \ \text{nat} \ \text{with} \\
\quad | \ \text{FO} & \Rightarrow \lambda \ D, \text{Fexc} \ \left(\text{even}\_\text{even'} \ D\right) \\
\quad | \ \text{FS} \ i & \Rightarrow \lambda \ D, \text{S} \ \left(\text{half} \ i \ \left(\pi\_\text{even} \ D\right)\right) \\
\quad \text{end} \\
\end{align*}
\]
\text{end} \ D.

Definition \( \pi\_\text{even} \ \{n\} \ \{i: \ t \ n\} \ (D: \ \text{even} \ (\text{FS} \ (\text{FS} \ i))) : \ \text{even} \ i \) :=
\[
\begin{align*}
\text{match} \ D \ \text{in} \ \text{even} \ j \ \text{return} \ \forall \ G: \ \text{shape} \ j, \ \text{even} \ (\text{fpred2} \ j \ G) \ \text{with} \\
| \ \text{even}\_\text{SS} \ i \ e & \Rightarrow \lambda \ G, e \\
| \ _ & \Rightarrow \lambda \ G, \text{match} \ G \ \text{with} \ \text{end} \\
\end{align*}
\]
Definition pr2 n : sh n -> nat :=
  match n with
  | S (S x) => \ G, x
  | _ => \ G, Fexc G
end.

Definition fpred2 {m} (j: t m) : \forall G : shape j, t (pr2 m (shape_sh G)) :=
  match j in t m return \forall G: shape j, t (pr2 m (shape_sh G)) with
  | FS j =>
    match j in t m return \forall G: shape (FS j), t (pr2 (S m) (shape_sh G)) with
    | FS j => \ G, j
    | _ => \ G, Fexc G
    end
  | _ => \ G, Fexc G
end.
Definition shape {n} (i : t n) : Prop :=
  match i with
    | FS (FS i) => ⊤
    | _ => ⊥
  end.

Definition sh n : Prop :=
  match n with
    | S (S n) => ⊤
    | _ => ⊥
  end.

Lemma shape_sh_inter n (i : t n): shape i -> sh n.
Proof. destruct i as [ | n1 [ | n2 i]]; intro G; now case G. Qed.

(* Explicit term *)
Definition shape_sh n i : t n : shape i -> sh n :=
  match i in t n return shape i -> sh n with
    | FO => λ G, match G with end
    | FS i =>
      match i in t n return shape (FS i) -> sh (S n) with
        | FO => λ G, match G with end
        | FS i => λ G, I
      end
  end.

Examples:
- f91
- Paulson's normalisation of if-then-else expressions
- first-order unification

\( G \) is used inside \( D \), hence has to be defined first
Variants of the Braga method

- Accessibility binary relation instead of custom inductive domain predicate
- Simulating induction-recursion instead of reasoning on $\mathcal{G}$
  Inductive-recursive equations are derived from $\mathcal{G}$ for a deterministic $\mathcal{G}$