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

`https://github.com/DmxLarchey/The-Braga-Method`

Dominique Larchey-Wendling and Jean-François Monin.
The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq, chapter 8, pages 305–386.
In Klaus Mainzer, Peter Schuster, and Helmut Schwichtenberg, editors.
Proof and Computation II: From Proof Theory and Univalent Mathematics to Program Extraction and Verification.
World Scientific, September 2021.

Topics

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

Recursive programs in Type Theory

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

OCaml and Coq: differences

In OCaml only

- primitive data types (int, char, etc.)
- imperative features
- exceptions – inhabit any type
- non terminating computations – inhabit any type

In Coq only

- Computations on types; types have a type (called a universe)
- 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 `Prop` is just `False` ($\perp$)
  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: $P \to Q$
- 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 $Q : A \to$ `Prop`
  Predicates are just dependent types

Computations on proof trees

Lemma elimination

Lemma `many_primes` : ∀ `n: nat`, ∃ `p: nat`, `n` ≤ `p` ∧ `prime p`.
Proof of the statement

Theorem `thm1` : *some other statement*
Proof using (`many_primes 1960`) *and* (`many_primes` 24^3)

Can be computed into

Theorem `thm1` : *some other statement*
Proof including specific proofs
of ∃ `p: nat`, `1960` ≤ `p` ∧ `prime p`
and ∃ `p: nat`, 24^3 ≤ `p` ∧ `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
```

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
```

From OCaml to Coq and conversely: extraction (2/2)

Separation between

- "real" data (and fonctions on them)
- (logical) knowledge or reasoning about them

No information leakage between Prop and Type

- Statically ensured by a constraint on pattern-matching
- Some debattable 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

$\rightarrow$ 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 (head l) + f2 (tail l)
```

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

ML

```
match l with
| [] -> 0
| h :: t -> f1 h + f2 t
```

Type checking does the job :)

LISP viz ML style

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

```
match l with
| [] -> 0
| h :: t -> f1 h + f2 t
```

Type checking does the job :)

Easy dependent pattern matching

Generalization is mandatory

```
Definition deptyp n : Type :=
  match n with
  | 0 ⇒ bool
  | 1 ⇒ nat
  | _ ⇒ unit
  end.
```

```
Definition fct1 n : deptyp n :=
  match n return deptyp n with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt (* () in OCaml *)
  end.
```

```
Definition fct2 n : deptyp (n*n) :=
  match n*n as n2 return deptyp n2 with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt
  end.
```

Easy dependent pattern matching

Generalization is mandatory

```
Definition deptyp n : Type :=
  match n with
  | 0 ⇒ bool
  | 1 ⇒ nat
  | _ ⇒ unit
  end.
```

```
Definition fct1 n : deptyp n :=
  match n return deptyp n with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt (* () in OCaml *)
  end.
```

```
Definition fct2 n : deptyp (n*n) :=
  match n*n as n2 return deptyp n2 with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt
  end.
```

Easy dependent pattern matching

Generalization is mandatory

```
Definition deptyp n : Type :=
  match n with
  | 0 ⇒ bool
  | 1 ⇒ nat
  | _ ⇒ unit
  end.
```

```
Definition fct1 n : deptyp n :=
  match n return deptyp n with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt (* () in OCaml *)
  end.
```

```
Definition fct2 n : deptyp (n*n) :=
  match n*n as n2 return deptyp n2 with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt
  end.
```

Easy dependent pattern matching

Generalization is mandatory

```
Definition deptyp n : Type :=
  match n with
  | 0 ⇒ bool
  | 1 ⇒ nat
  | _ ⇒ unit
  end.
```

```
Definition fct1 n : deptyp n :=
  match n return deptyp n with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt (* () in OCaml *)
  end.
```

```
Definition fct2 n : deptyp (n*n) :=
  match n*n as n2 return deptyp n2 with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt
  end.
```

Easy dependent pattern matching

Generalization is mandatory

```
Definition deptyp n : Type :=
  match n with
  | 0 ⇒ bool
  | 1 ⇒ nat
  | _ ⇒ unit
  end.
```

```
Definition fct1 n : deptyp n :=
  match n return deptyp n with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt (* () in OCaml *)
  end.
```

```
Definition fct2 n : deptyp (n*n) :=
  match n*n as n2 return deptyp n2 with
  | 0 ⇒ false
  | 1 ⇒ 3
  | _ ⇒ tt
  end.
```

Implicative dependent pattern-matching : Trojan horses

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 is_cons l : Prop :=
  match l with :: ⇒ ⊤ | _ ⇒ ⊥ end.

Definition head l : is_cons l → X :=
  match l with
  | x :: t ⇒ λG, x
  | _      ⇒ λG, match G 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.
```

Universal realizer

Usual universal realizer: exception

```
let univ : α = assert false
```

Another universal realizer: loop

```
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
```

Loops in an inconsistent environment in Coq

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

The same in 2 shorter lines (with $\bot$ 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.
```

Loops with an absurd parameter in Coq

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

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

The same in 2 shorter lines (with $\bot$ 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.
```

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.
```

Removing loops at extraction

```
Definition Fexc {X: Type} (f: ⊥) : X :=
  match Floop Empty_set f with end.
```

Empty_set = empty informative inductive type

Floop Empty_set f has type Empty_set

At Coq level, no leakage from Prop to Type

match whatever with end

extracted at OCaml level as `assert false`

Floop (params) considered as dead code → canceled

Inversion, simple example

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

e : eval (Te_div0 (Te_const 1)) v
=================================
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)

```
H:eval_expr (Genv.globalenv prog_adc) e m RV
   (Ecall (Evalof (Evar copy_StatusRegister T14) T14)
      (Econs
         (Eaddrof
            (Efield (Ederef (Evalof (Evar proc T3) T3) T6)
              adc_compcert.cpsr T7) T8)
         (Econs
            (Ecall (Evalof (Evar spsr T15) T15)
               (Econs (Evalof (Evar proc T3) T3) Enil) T8)
                Enil))
      T12) t m' a'
==========================================================
   proc_state_related m' e st'

  inv H. inv H4. inv H9. inv H5. inv H4. inv H5.
  inv H15. inv H4. inv H5. inv H14. inv H4. inv H3.
  inv H15. inv H5. inv H4. inv H5. inv H21. inv H13.
  ...
```

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

Small Inversions V0: absurd Cases

```
e : eval (Te_div0 (Te_const 1)) v
=================================
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 => ∀ 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_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 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 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

Receipe

Given an inductive relation `rel : Tx → Ty1 → ... Prop`
with "input" argument `x : Tx`, define:

- For each input case (constructor `C`) in `Tx`,
  an *auxiliary inductive relation* of type `Ty1 → ... Prop`
  by copy and paste of relevant telescopes of `rel`
  No recursion
- A *dispatch function* `rel’` from `x : Tx` to `Ty1 → ... Prop`
  by pattern matching on `x`
- A trivial proof `rel_rel’ : rel` implies `rel’`

Usage

- Given a hypothesis `R : rel (C...) expr_1...`
  invoke a pattern matching on `rel_rel’ R`
- Boils down to the relevant *aux. inductive relation* corresponding to `(C...)`

Small inversions V1, with auxiliary inductive types

Receipe

Given an inductive relation `rel : Tx → Ty1 → ... Prop`
with "input" argument `x : Tx`, define:

- For each input case (constructor `C`) in `Tx`,
  an *auxiliary inductive relation* of type `Ty1 → ... Prop`
  by copy and paste of relevant telescopes of `rel`
  No recursion
- A *dispatch function* `rel'` from `x : Tx` to `Ty1 → ... Prop`
  by pattern matching on `x`
- A trivial proof `rel_rel' : rel` implies `rel'`

Usage

- Given a hypothesis `R : rel (C...) expr_1...`
  invoke a pattern matching on `rel_rel' R`
- Boils down to the relevant *aux. inductive relation* corresponding to `(C...)`

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/

Small inversion V1, example (1/4)

```
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.
```

Small inversion V1, example (1/4)

```
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.
```

Small inversion V1, example (1/4)

```
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.
```

Small inversion V1, example (1/4)

```
 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.
```

```
Definition eval_eval' {t v} : eval t v → eval' t v.
Proof. intro e; destruct e; constructor; assumption. Qed.
```

```
Definition eval_eval' {t v} : eval t v → eval' t v := λ 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.
```

```
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

  |  E_Plus  t1 t2 n1 n2 e1 e2
  => E_Plus' t1 t2 n1 n2 e1 e2
  end.
```

Small inversion V1, example (2/4)

```
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.
```

Small inversion V1, example (2/4)

```
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.
```

Small inversion V1, example (2/4)

```
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.
```

Small inversion V1, example (2/4)

```
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.
```

Small inversion V1, example (3/4)

```
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

Small inversion V1, example (3/4)

```
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

Small inversion V1, example (3/4)

```
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

Small inversion V1, example (3/4)

```
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)
```

No additional equality

Small inversion V1, example (3/4)

```
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

Small inversion V1, example (3/4)

```
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

Small inversion V1, example (3/4)

```
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

Small inversion V1, example (3/4)

```
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

Small inversion V1, example (3/4)

```
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

Small inversion V1, example (4/4)

```
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 `rel'` and `rel_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 `rel'` and `rel_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

Coq inversion viz small inversions V1 (1/2)

```
∀ 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
        (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) H H0
```

Coq inversion viz small inversions V1 (1/2)

```
∀ 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
        (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) H H0
```

Coq inversion viz small inversions V1 (2/2)

```
(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) H 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)
```

Small inversions from V1 to full V0

Full V0 can be seen as a purely functional translation of V1

Continuation Passing Style / polymorphic lambda-calculus

Beating Coq inversion: on dependent types (1/4)

Bounded natural numbers – or finite sets t n of size n

```
Inductive t : nat → Set :=
  | F0 {n} : t (S n)
  | FS {n} : t n → t (S n).
```

Even bounded numbers

```
Inductive even : forall {n}, t n → Prop :=
  | even_0 {n} :               even (@F0 n)
  | even_SS {n} (i: t n) : even i → even (FS (FS i)).
```

Issues on lemmas such as

```
∀ n (i: t n), even (FS (FS i)) → even i.
∀ n m (i: t n) (j: t m),
    even (Fplus i j) → even i → even j.
```

Beating Coq inversion: on dependent types (1/4)

Bounded natural numbers – or finite sets `t n` of size `n`

```
Inductive t : nat → Set :=
  | F0 {n} : t (S n)
  | FS {n} : t n → t (S n).
```

Even bounded numbers

```
Inductive even : forall {n}, t n → Prop :=
  | even_0 {n} :              even (@F0 n)
  | even_SS {n} (i: t n) : even i → even (FS (FS i)).
```

Issues on lemmas such as

```
∀ n (i: t n), even (FS (FS i)) → even i.
∀ n m (i: t n) (j: t m),
   even (Fplus i j) → even i → even j.
```

Beating Coq inversion: on dependent types (1/4)

Bounded natural numbers – or finite sets `t n` of size `n`

```
Inductive t : nat → Set :=
  | F0 {n} : t (S n)
  | FS {n} : t n → t (S n).
```

Even bounded numbers

```
Inductive even : forall {n}, t n → Prop :=
  | even_0 {n} :              even (@F0 n)
  | even_SS {n} (i: t n) : even i → even (FS (FS i)).
```

Issues on lemmas such as

```
∀ n (i: t n), even (FS (FS i)) → even i.
∀ n m (i: t n) (j: t m),
    even (Fplus i j) → even i → even j.
```

```
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
  | F0         => 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.
```

Beating Coq inversion: on dependent types (2/4)

```
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
  | F0         => 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.
```

Beating Coq inversion: on dependent types (2/4)

```
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
  | F0          => 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.
```

Beating Coq inversion: on dependent types (3/4)

```
Fixpoint lift1 m {n} (i : t n) : t (m + n) :=
  match i in t n return t (m + n) with
  | F0   => t_n_Sm F0
  | 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
  | @F0 n => lift1 (S n) j
  | FS i  => FS (Fplus i j)
  end.
```

Beating Coq inversion: on dependent types (3/4)

```
Fixpoint lift1 m {n} (i : t n) : t (m + n) :=
  match i in t n return t (m + n) with
  | F0   => t_n_Sm F0
  | 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
  | @F0 n => lift1 (S n) j
  | FS i  => FS (Fplus i j)
  end.
```

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

```
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 : 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 : 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 : 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
```

Coq inversion viz small inversions V1 (1/4)

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) (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 F0 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 =
```

Coq inversion viz small inversions V1 (1/4)

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) (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 F0 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 =
```

Coq inversion viz small inversions V1 (2/4)

```
     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) F0 =
            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) F0 =
         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))) F0 =
                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))) F0)
             (fun e : n2 : nat & t n2 =>
              let (x, t0) := e in match t0 with
                                  | F0 => 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
```

Coq inversion viz small inversions V1 (4/4)

```
              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)
```

Small inversions, summary

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: (pattern-matching)^n

Recursive programs

Number of steps

Given

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

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

```
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 `nsa x 0` always return the same value as `ns x`?

Looks ridiculously impossible in Coq

- Write Coq programs for `ns` and `nsa`
- Such that they are extracted exactly as expected
- Reason about them

Issue

No clue about the (identical) termination of `ns` and `nsa`

Braga method, 1st idea : inductive domain + projections

Inductive characterization of the domain of ns and nsa

```
Inductive 𝔻ns (x: X) : Prop :=
  | 𝔻ns_tr : b x = true → 𝔻ns x
  | 𝔻ns_fa : b x = false → 𝔻ns (g x) → 𝔻ns x.
```

Target

```
Fixpoint fct x (D : 𝔻ns x) {struct D} : N :=
  match b x with
  | true ⇒ . . .
  | false ⇒ . . . fct (g x) (proj D) . . .
  end.
```

With (proj D) $<$ D

Requirement on the projection

Inductive characterization of the domain of ns and nsa

```
Inductive 𝔻ns (x: X) : Prop :=
  | 𝔻ns_tr : b x = true → 𝔻ns x
  | 𝔻ns_fa : b x = false → 𝔻ns (g x) → 𝔻ns x.
```

(proj D) defined for all x such that b x = false

```
Definition prj_𝔻ns x (E: b x = false) (D: 𝔻ns x): 𝔻ns (g x) :=
  match D with
  | 𝔻ns_tr _ E'   => match false_true E E' with end
  | 𝔻ns_fa _ _ D  => D
  end.
```

```
Lemma false_true x : x = false -> x = true -> ⊥.
```

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.
```

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.
```

2nd idea of the Braga method: input-output graph

```
Inductive 𝔾ns (x: X) : nat → Prop :=
 | in_grns_0  : b x = true  →  x ⟼ns 0
 | in_grns_1 o: b x = false →  g x ⟼ns o  →  x ⟼ns S o
where "x ⟼ns o" := (𝔾ns x o).

Fixpoint ns_pwc x (D: 𝔻ns x) : {o | x ⟼ns o}.
Proof. refine(
   match b x as bx return b x = bx → _ with
   | true  => λ E, exist _ 0 _
   | false => λ E, let (o,Go) := ns_pwc (g x) (prj_𝔻ns E D)
                   in exist _ (S o) _
   end eq_refl).
  - constructor 1; exact E.
  - constructor 2; assumption.
Defined.
```

Remarks about inversion

- `prj_`$\mathbb{D}$`ns` 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 $\mathbb{G}$`ns` is deterministic
- Coq automated inversion provide structurally smaller terms!
  ...when it works

Remarks about inversion

- `prj_`$\mathbb{D}$`ns` 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 $\mathbb{G}$`ns` is deterministic
- Coq automated inversion provide structurally smaller terms!
  ...when it works

Braga method for LISP / ML style programs

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 $\perp$ in absurd cases.

For ML style programs, we can instead use Trojan horses
based on $\perp$ and $\top$, which can be directly exploited.
See below a typical programming pattern.

Reasoning on fold_left

Functional specification

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

Inductive specification

$$\frac{}{\texttt{[]} \longmapsto_{\texttt{fl}} \texttt{b0}} \qquad \frac{\texttt{u} \longmapsto_{\texttt{fl}} \texttt{b}}{\texttt{u +: z} \longmapsto_{\texttt{fl}} \texttt{f b z}}$$

Reasoning on fold_left

Functional specification

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

Inductive specification

```
                              u    ⟼fl   b
    -------------            --------------------
    []   ⟼fl   b0            u +: z   ⟼fl   f b z
```

Reasoning on fold_left

Specification

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

Regular program

```
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
```

Braga version

```
Fixpoint foldl_pwc l (D : 𝔻lz l) : {b | l ⟼fl b}.
Proof.
gen_help l G_foldl ; apply up_llP in D; revert D.
refine ( match l2r l with
    | Nilr ⇒ λD T, exist _ b0 _
    | Consr u z ⇒ λD T,
                 let (b, Cb ) := foldl_pwc u (π𝔻lz D)
                 in  exist _ (f b z) _
end).
- apply T ; constructor 1.
- apply T ; constructor 2; exact Cb .
Qed.
```

Correctness of standard fold_left

Easy

Use $\quad$ foldl f b (u +: z) = f (foldl f b u) z

No use of associativity of append
(append is not in the vocabulary)

```
                        𝔻lz u            𝔻lr (l2r l)
 ---------     ----------------    -------------
 𝔻lr Nilr      𝔻lr (Consr u z)         𝔻lz l
```

Projection for second rule

```
Let π_𝔻lr {u z} (D: 𝔻lr (Consr u z)) : 𝔻lz u :=
  match D in 𝔻lr r return
      let g := match r with Consr u z => ⊤ | _ => ⊥ end in
      let u := match r with Consr u z => u | _ => u end in
      g → 𝔻lz u with
  | 𝔻lr_Consr u z D => λ G, D
  | _              => λ G, match G with end (* < D as well *)
  end I (* proof of ⊤ *).
```

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

```
                   𝔻lz u            𝔻lr (l2r l)
---------     ---------------     ------------
𝔻lr Nilr      𝔻lr (Consr u z)       𝔻lz l
```

Projection for second rule

```
Let π_𝔻lr {u z} (D: 𝔻lr (Consr u z)) : 𝔻lz u :=
  match D in 𝔻lr r return
        let g := match r with Consr u z => ⊤ | _ => ⊥ end in
        let u := match r with Consr u z => u  | _ => u end in
        g → 𝔻lz u with
  | 𝔻lr_Consr u z D => λ G, D
  |  _              => λ G, match G with end (* < D as well *)
  end I (* proof of ⊤ *).
```

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

$$\frac{}{\mathbb{D}\texttt{lr Nilr}} \qquad \frac{\mathbb{D}\texttt{lz u}}{\mathbb{D}\texttt{lr (Consr u z)}} \qquad \frac{\mathbb{D}\texttt{lr (l2r l)}}{\mathbb{D}\texttt{lz l}}$$

Projection for second rule

```
Let π_𝔻lr {u z} (D: 𝔻lr (Consr u z)) : 𝔻lz u :=
  match D in 𝔻lr r return
        let g := match r with Consr u z => ⊤ | _ => ⊥ end in
        let u := match r with Consr u z => u | _ => u end in
        g → 𝔻lz u with
  | 𝔻lr_Consr u z D => λ G, D
  | _               => λ G, match G with end (* < D as well *)
  end I (* proof of ⊤ *).
```

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

```
                              𝔻lz u         𝔻lr (l2r l)
---------         ----------------    -------------
𝔻lr Nilr          𝔻lr (Consr u z)         𝔻lz l
```

Projection for second rule

```
Let π_𝔻lr {u z} (D: 𝔻lr (Consr u z)) : 𝔻lz u :=
  match D in 𝔻lr r return
        let g := match r with Consr u z => ⊤ | _ => ⊥ end in
        let u := match r with Consr u z => u | _ => u end in
        g → 𝔻lz u with
  | 𝔻lr_Consr u z D => λ G, D
  |  _                => λ G, match G with end (* < D as well *)
  end I (* proof of ⊤ *).
```

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 Consr 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:

let `u := match r with Consr 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.

Other approaches

Ad-hoc

`O` (or `FO`) 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

`O` (or `FO`) for (bounded) nats
The simplest thing to do in handcrafted approaches

General

`match something-reducing-to-a-proof-of-`$\perp$ `with end`

Suspicious subsingleton elimination, should be avoided

Can be circumvented using loops

Beating again Coq inversion: on dependent types (1/3)

```
Fixpoint half n (i: t n) (D: even i) {struct D} : nat :=
  match i with
  | F0   => λ D, 0
  | FS i =>
    match i return even (FS i) -> nat with
    | F0   => 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.
```

Beating again Coq inversion: on dependent types (1/3)

```
Fixpoint half n (i: t n) (D: even i) {struct D} : nat :=
  match i with
  | F0   => λ D, 0
  | FS i =>
    match i return even (FS i) -> nat with
    | F0   => 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.
```

Beating again Coq inversion: on dependent types (2/3)

```
Definition pr2 n : sh n -> nat :=
  match n with
  | S (S x) => λ G, x
  | _       => λ G, Fexc G
  end.

Definition fpred2 {m} (j: t m) : ∀ G : shape j, t (pr2 m (shape_sh G)) :=
  match j in t m return ∀ G: shape j, t (pr2 m (shape_sh G)) with
  | FS j =>
    match j in t m return ∀ G: shape (FS j), t (pr2 (S m) (shape_sh G)) wit
    | FS j => λ G, j
    | _    => λ G, Fexc G
    end
  | _ => λ G, Fexc G
  end.
```

Beating again Coq inversion: on dependent types (3/3)

```
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
 | F0   => λ G, match G with end
 | FS i =>
     match i in t n return shape (FS i) -> sh (S n) with
     | F0 =>   λ G, match G with end
     | FS i => λ G, I
     end
 end.
```

Embedded recursion

- Examples :
  - f91
  - Paulson's normalisation of if-then-else expressions
  - first-order unification
- $\mathbb{G}$ is used inside $\mathbb{D}$, 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 $\mathbb{G}$
  Inductive-recursive equations are derived from $\mathbb{G}$ for a deterministic $\mathbb{G}$