Génération automatique de tests d'égalité corrects en Coq, en pratique

Benjamin Grégoire, Jean-Christophe Léchenet, Enrico Tassi

AFADL 2023 – June 7th, 2023

Equality: Prop vs. bool

- Equality in Prop: Logic.eq, a.k.a. "="
  - polymorphic: $\forall$ A : Type, A $\to$ A $\to$ Prop
  - 4 = 4, "er" = "er", 4 $\neq$ 2, ~~4 $\neq$ "er"~~
- Equality test in bool:
  - one per type/type family
  - For a type T, T_eqb : T $\to$ T $\to$ bool
- Correction: $\forall$ t1 t2 : T, T_eqb t1 t2 = true $\leftrightarrow$ t1 = t2
- Motivation
  - Decidable equality: $\forall$ t1 t2 : T, t1 = t2 $\vee$ t1 $\neq$ t2
  - Equality that computes

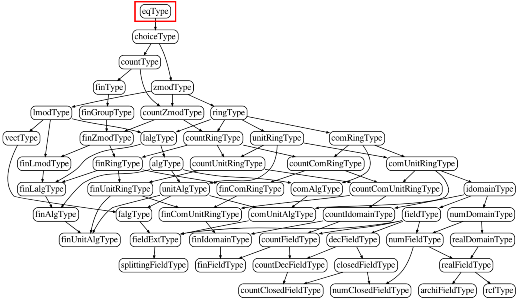
An example: nat

```
Inductive nat : Set := O : nat | S : nat → nat.

Fixpoint nat_eqb (n1 n2 : nat) :=
  match n1 with
  | O    ⇒ match n2 with | O ⇒ true  | S _  ⇒ false           end
  | S n1 ⇒ match n2 with | O ⇒ false | S n2 ⇒ nat_eqb n1 n2 end
  end.
```

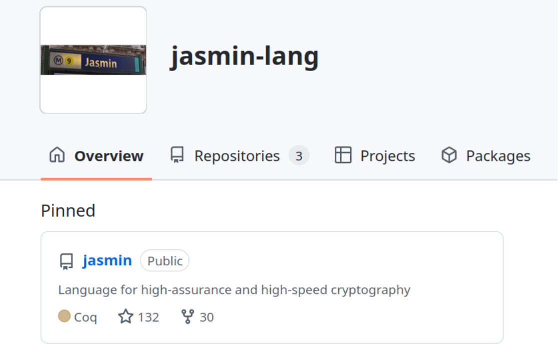
An example: nat

```
Lemma nat_eqb_correct : ∀ n1 n2, nat_eqb n1 n2 = true → n1 = n2.
Proof.
  induction n1; intros n2.
  - destruct n2; [reflexivity|discriminate].
  - destruct n2; [discriminate|intros ?; f_equal; apply IHn1; assumption].
Qed.

Lemma nat_eqb_refl : ∀ n, nat_eqb n n = true.
Proof. induction n; [reflexivity|assumption]. Qed.

Lemma nat_eqb_OK : ∀ n1 n2, nat_eqb n1 n2 = true ↔ n1 = n2.
Proof.
  split; [apply nat_eqb_correct|intros ← ; apply nat_eqb_refl].
Qed.
```

Context



Mathematical components



Problem #1

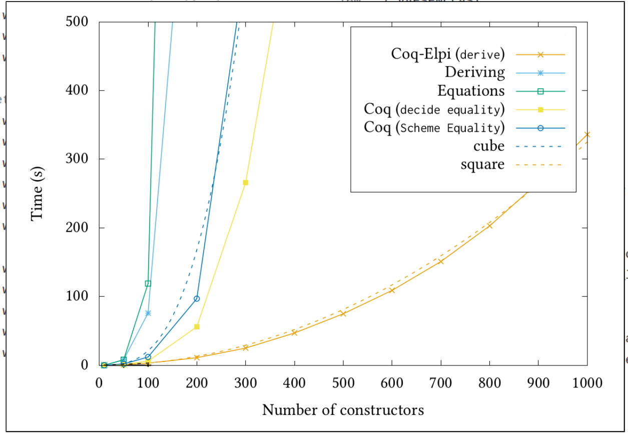
```
Variant x86_op : Type :=
  (* Data transfert *)
| MOV    of wsize              (* copy *)
| MOVSX  of wsize & wsize      (* sign-extend *)
| MOVZX  of wsize & wsize      (* zero-extend *)
| CMOVcc of wsize              (* conditional copy

  (* Arithmetic *)
| ADD    of wsize                  (* add unsigned
| SUB    of wsize                  (* sub unsigned
| MUL    of wsize                  (* mul unsigned
| IMUL   of wsize                             (* mu
| IMULr  of wsize  (* oprd * oprd *)          (* mu
| IMULri of wsize  (* oprd * oprd * imm *)    (* mu

| DIV    of wsize                       (* div uns
| IDIV   of wsize                       (* div   s
| CQO    of wsize                           (*
| ADC    of wsize                 (* add with carry
| SBB    of wsize                 (* sub with borro
```

```
| AESENC
| VAESENC
| AESENCLAST
| VAESENCLAST
| AESIMC
| VAESIMC
| AESKEYGENASSIST
| VAESKEYGENASSIST
.

Scheme Equality for x86_op.

Lemma x86_op_eq_axiom : Equality.axiom x86_op_beq.
Proof.
  move=> x y;apply:(iffP idP).
  + by apply: internal_x86_op_dec_bl.
  by apply: internal_x86_op_dec_lb.
Qed.

Definition x86_op_eqMixin := Equality.Mixin x86_op_eq_axiom.
Canonical x86_op_eqType := EqType x86_op x86_op_eqMixin.
```

Problem #1

```
Variant x86_op : Type :=
  (* Data transfert *)
| MOV    of wsize                (* copy *)
| MOVSX  of w
| MOVZX  of w
| CMOVcc of w

  (* Arithmet
| ADD    of w
| SUB    of w
| MUL    of w
| IMUL   of w
| IMULr  of w
| IMULri of w

| DIV    of w
| IDIV   of w
| CQO    of w
| ADC    of w
| SBB    of w
```

```
| AESENC
| VAESENC
| AESENCLAST
| VAESENCLAST
```




```
.axiom x86_op_beq.

c_bl.
lb.

ality.Mixin x86_op_eq_axiom.
e x86_op x86_op_eqMixin.
```

Problem #2

```
Variant wsize := U8 | U16 | U32 | U64 | U128 | U256.
Definition wsize_size (s: wsize) : nat := ... (* omitted *)
Record word (nbits: wsize) := mkWord {
  w : Z;
  _ : 0 <=? w && w <? 2^(wsize_size nbits)
}.
Variant value : Type :=
| Vbool  : bool → value
| Vint   : Z → value
| Vword  : ∀ s, word s → value.
```

Problem #3

```
Inductive instr :=
| Cassgn   : lval → assgn_tag → stype → pexpr → instr
| Copn     : lvals → assgn_tag → sopn → pexprs → instr
| Csyscall : lvals → syscall_t → pexprs → instr
| Cif      : pexpr → cmd → cmd  → instr
| Cfor     : var_i → range → cmd → instr
| Cwhile   : align → cmd → pexpr → cmd → instr
| Ccall    : iinfo → lvals → funname → pexprs → instr
where "'cmd'" := (list (info * instr)).
```

Problem #3

```
Inductive
| Cassgn
| Copn
| Csyscal
| Cif
| Cfor
| Cwhile
| Ccall
where "'cr
```

```
Inductive instr_r :=
| Cassgn   : lval -> assgn_tag -> stype -> pexpr -> instr_r
| Copn     : lvals -> assgn_tag -> sopn -> pexprs -> instr_r
| Csyscall : lvals -> syscall_t -> pexprs -> instr_r
| Cif      : pexpr -> seq instr -> seq instr  -> instr_r
| Cfor     : var_i -> range -> seq instr -> instr_r
| Cwhile   : align -> seq instr -> pexpr -> seq instr -> instr_r
| Ccall    : inline_info -> lvals -> funname -> pexprs -> instr_r

with instr := MkI : instr_info -> instr_r ->  instr.

End ASM_OP.

Notation cmd := (seq instr).

Section CMD_RECT.

  Context `{asmop:asmOp}.

  Variables (Pr:instr_r -> Type) (Pi:instr -> Type) (Pc : cmd -> Type).
  Hypothesis Hmk  : forall i ii, Pr i -> Pi (MkI ii i).
  Hypothesis Hnil : Pc [::].
  Hypothesis Hcons: forall i c, Pi i -> Pc c -> Pc (i::c).
  Hypothesis Hasgn: forall x tg ty e, Pr (Cassgn x tg ty e).
  Hypothesis Hopn : forall xs t o es, Pr (Copn xs t o es).
  Hypothesis Hsyscall : forall xs o es, Pr (Csyscall xs o es).
  Hypothesis Hif  : forall e c1 c2, Pc c1 -> Pc c2 -> Pr (Cif e c1 c2).
  Hypothesis Hfor : forall v dir lo hi c, Pc c -> Pr (Cfor v (dir,lo,hi) c).
  Hypothesis Hwhile : forall a c e c', Pc c -> Pc c' -> Pr (Cwhile a c e c').
  Hypothesis Hcall: forall i xs f es, Pr (Ccall i xs f es).

  Section C.
```

```
pr → instr
prs → instr
nstr

nstr
rs → instr
```

The paper, the talk

- Problem #1: linear schema, this talk!
- Problem #2: heterogeneous tests, see paper
- Problem #3: deep induction, see paper

The paper, the talk

- Problem #1: linear schema, this talk!
- Problem #2: heterogeneous tests, see paper
- Problem #3: deep induction, see paper

```
Variant value : Type :=
| Vbool  : bool → value
| Vint   : Z → value
| Vword  : ∀ s, word s → value.

   word_eqb : ∀s     , word s  → word s  → bool
   word_eqb : ∀s1 s2, word s1 → word s2 → bool
```

The paper, the talk

- Problem #1: linear schema, this talk!
- Problem #2: heterogeneous tests, see paper
- Problem #3: deep induction, see paper

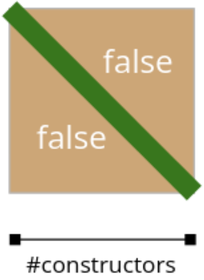
Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq

Enrico Tassi
Université Côte d'Azur – Inria, France
Enrico.Tassi@inria.fr

Abstract

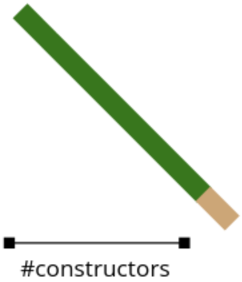
The root of all evil

```
Fixpoint eqb (x y:nat) :=
  match x with
  | O   ⇒ match y with O ⇒ true  | S _ ⇒ false   end
  | S n ⇒ match y with O ⇒ false | S m ⇒ eqb n m end
  end.

Lemma eqb_refl : ∀ x, eqb x x :=
  nat_ind (fun n: nat ⇒ eqb n n)
    eq_refl (* eqb O O *)
    (fun p (IH: eqb p p) ⇒ IH). (* eqb (S p) (S p) *)

Lemma eqb_correct : ∀ x y, eqb x y → x = y :=
  nat_ind (fun n: nat ⇒ ∀ y: nat, eqb n y → n = y)
    (fun y: nat ⇒
      match y as n return (eqb O n → O = n) with
      | O ⇒ fun _: eqb O O ⇒ eq_refl O
      | S q ⇒ ... (* absurd: eqb O (S q) → false *)
      end)
    (fun p (IH: ∀ z: nat, eqb p z → p = z) y ⇒
      match y as n return (eqb (S p) n → S p = n) with
      | O ⇒ ... (* absurd: eqb (S p) O → false *)
      | S q ⇒ fun h: eqb (S p) (S q) ⇒
          (* since: eqb (S p) (S q) ≈ eqb p q *)
          f_equal S (IH q h)
      end).
```



The idea

```
Section Core.
Variable A : Type.
Variable tag : A → positive.
Variable fields_t : positive → Type.
Variable fields : ∀ (a:A), fields_t (tag a).
Variable eqb_fields : ∀ t, fields_t t → fields_t t → bool.

Definition eqb_body_nice (x y: A) :=
  let t1 := tag x in
  let t2 := tag y in
  match Pos.eq_dec t2 t1 with
  | left heq ⇒
    let f1: fields_t t1 := fields x in
    let f2: fields_t t2 := fields y in
    eqb_fields t1 f1
      (match heq with eq_refl ⇒ f2 end)
  | right _ ⇒ false
  end.
```



Nat

```
Definition nat_tag n := match n with O ⇒ 1 | S _ ⇒ 2 end.

Definition nat_fields_t p :=
  match p with
  | 1 ⇒ unit (* no argument *)
  | 2 ⇒ nat
  | _ ⇒ unit (* dummy case *)
  end.

Definition nat_fields n : nat_fields_t (nat_tag n) :=
  match n with
  | O ⇒ tt
  | S p ⇒ p
  end.

Definition nat_eqb_fields rec t :
  nat_fields_t t → nat_fields_t t → bool :=
  match t with
  | 1 ⇒ fun _ _ ⇒ true
  | 2 ⇒ fun x y ⇒ rec x y
  | _ ⇒ fun _ _ ⇒ false (* dead code *)
  end.
```

```
Fixpoint nat_eqb (n1 n2: nat) {struct n1} :=
  eqb_body_nice nat nat_tag nat_fields_t nat_fields
    (nat_eqb_fields nat_eqb) n1 n2.
```

Term size is O(#constructors) !!!

And then... you benchmark

...and you are still quadratic

The next 4 slides are very Coq specific

Tame the termination checker

```
Definition nat_tag n := match n with O => 1 | S _ => 2 end.

Definition nat_fields_t p :=
  match p with
  | 1 => unit (* no argument *)
  | 2 => nat
  | _ => unit (* dummy case *)
  end.

Definition nat_fields n : nat_fields_t (nat_tag n) :=
  match n with
  | O => tt
  | S p => p
  end.

Definition nat_eqb_fields rec t :
  nat_fields_t t -> nat_fields_t t -> bool :=
  match t with
  | 1 => fun _ _ => true
  | 2 => fun x y => rec x y
  | _ => fun _ _ => false (* dead code *)
  end.
```

```
Fixpoint nat_eqb(n1 n2: nat) {struct n1} :=
  eqb_body_nice nat nat_tag nat_fields_t nat_fields
    (nat_eqb_fields nat_eqb) n1 n2.
```

```
Fixpoint nat_eqb (n1 n2: nat) {struct n1} :=
  let body :=
    eqb_body nat nat_tag nat_fields_t nat_fields
     (nat_eqb_fields nat_eqb) in
  match n1 with
  | O => body 1 tt n2
  | S p => body 2 p n2
  end.
```

Non-linear factor: pairs & implicit arguments

```
Inductive tree :=
  | Leaf
  | Node : nat → tree → tree → tree.

Definition tree_fields_t p :=
  match p with
  | 1 ⇒ unit
  | 2 ⇒ prod (prod nat tree) tree
  | _ ⇒ unit
  end.

Definition tree_fields x :=
  match x with
  | Leaf ⇒ tt
  | Node n l r ⇒
     pair (prod nat tree) tree (pair nat tree n l) r
  end.
```

```
Inductive box_for_Leaf := Box_Leaf.
Inductive box_for_Node :=
  Box_Node : nat → tree → tree → box_for_Node.

Definition tree_fields_t p :=
  match p with
  | 1 ⇒ box_for_Leaf
  | 2 ⇒ box_for_Node
  | _ ⇒ unit (* dummy case *)
  end.

Definition tree_fields x : tree_fields_t (tree_tag x) :=
  match x with
  | Leaf ⇒ Box_Leaf
  | Node n l r ⇒ Box_Node n l r
  end.
```

Non-linear factor: conjunctions

```
nat_eqb n1 n2 && rec l1 l2 && rec r1 r2
----------------------------------------
Node n1 l1 r1  =  Node n2 l2 r2


 Lemma andE: ∀ a b (P: Prop): (a → b → P) → a && b → P.
```

```
andE (nat_eqb n1 n2 && rec l1 l2) (rec r1 r2) (Node n1 l1 r1 = Node n2 l2 r2)
  (andE (nat_eqb n1 n2) (rec l1 l2) (rec r1 r2 → Node n1 l1 r1 = Node n2 l2 r2) ?Goal)))
```

```
Fixpoint implies (l: list bool) (P: Prop) : Prop :=
  match l with
  | [::] ⇒ P
  | b :: l ⇒ b → implies l P
  end.
Fixpoint allr (l: list bool) :=
  match l with
  | [::] ⇒ true
  | b :: l ⇒ b && allr l
  end.
Lemma impliesP (l: list bool) (P:Prop) : implies l P → allr l → P.
```

Non-linear factor: rewritings

```
hn : n1 = n2
hl : l1 = l2
hr : r1 = r2

Node n1 l1 r1 = Node n2 l2 r2
```

```
      eq_ind: ∀A (P: A→Prop) (x y: A), x = y → P x → P y



eq_ind nat (fun n ⇒ Node n1 l1 r1 = Node n l2 r2) n1 n2 hn
  (eq_ind tree (fun l ⇒ Node n1 l1 r1 = Node n1 l r2) l1 l2 hl
    (eq_ind tree (fun r ⇒ Node n1 l1 r1 = Node n1 l1 r) r1 r2 hr
      refl_equal))
```

Non-linear factor: rewritings

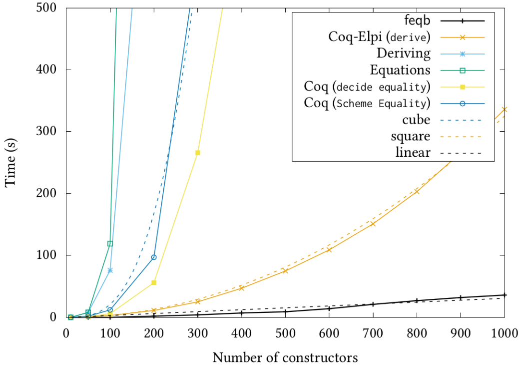
```
hn : n1 = n2
hl : l1 = l2
hr : r1 = r2

Node n1 l1 r1 = Node n2 l2 r2
```

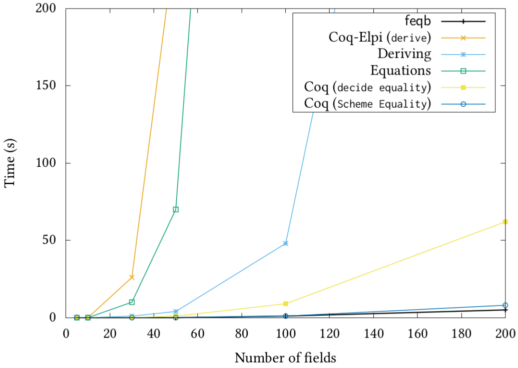
```
eq_ind: ∀A (P: A→Prop) (x y: A), x = y → P x → P y
```

```
eq_in
  (ec
```

```
eq_ind_r_nP [:: nat; tree; tree]
    (fun n l r ⇒ Node n1 l1 r1 = Node n l r)
    n1 n2 hn l1 l2 hl r1 r2 hr refl_equal.
```

Benchmarks: synthesis time





Implementation

Written in Coq-Elpi

- 800 LOC for equality tests and proofs
- 800 LOC for deep induction and related lemmas

	nat	list	instr/deep	forest	word	value	vect
Coq (decide equality)	✓	✓	✓	✓	✗	✗	✗
Coq (Scheme Equality)	✓	✓	✗	✗	✗	✗	✗
Coq-Elpi (derive)	✓	✓	✓	✗	✗	✗	✗
Deriving	✓	✓	✋	✓	✗	✗	✗
Equations	✓	✓	✗	✗	✓	✗	✓
feqb (this work)	✓	✓	✓	✗	✓	✓	✗

Conclusions

feqb: *automatic* synthesis in *linear time* covering *subtypes* and *containers*

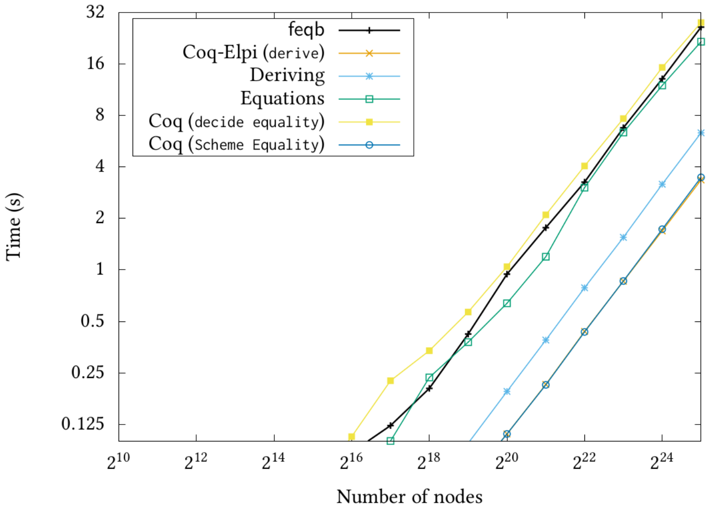
Status: integrated in Coq-Elpi 1.16

Next:

- synthesis of order relation
- speed up discriminate, inversion

Thank you!

Bench: execution time of the derived eq test



Comparison with related tools

Features	Induction		Equality test			Constructors' arguments		
Tool	deep	modular	separate	heterogeneous	size/kno	containers	dependent	irrelevant
Coq (`decide equality`)	✓	✗	✗	✗	$o(n^2)$	✓	✗	✗
Coq (`Scheme Equality`)	✗	✗	✓(1)	✗	$o(n^3)$	✗	✗	✗
Coq-Elpi (`derive`)	✓	✓	✓	✗	$o(n^2)$	✓	✗	✗
Deriving	✋	✗	✓	✗	$o(n)$?	✓	✗	✗
Equations	✗	✗	✗	✗	$o(n^2)$	✗	✓	✓
feqb (this work)	✓	✓	✓	✓	$o(n)$	✓	✓(2)	✓