Deriving proved equality tests in Coq-elpi

(Stronger induction principles for containers in Coq)

Enrico Tassi – ITP2019 – 10/9/2019


WARNING
λ-TERMS

Wanted

Automatic, schematic, derivation of "`eqType`"

```
Inductive rtree A : U :=
| Leaf (a : A)
| Node (l : list (rtree A)).
```

```
Definition eq_axiom A A_eq := ∀x y, reflect (x = y) (A_eq x y).
Definition eq_axiom_at A A_eq x := ∀y, reflect (x = y) (A_eq x y).

Definition rtree_eq : ∀A, (A → A → bool) → rtree A → rtree A → bool.

Lemma rtree_eq_OK : ∀A (A_eq : A → A → bool), eq_axiom A A_eq →
      eq_axiom (rtree A) (rtree_eq A A_eq).
```

Problem n°1

- Standard Coq “induction” principle for a data type defined using a container (list here)

```
Lemma rtree_ind      A (P : rtree A → U):
    (∀a : A, P (Leaf A a)) →
    (∀l : list (rtree A), P (Node A l)) →
  ∀r : rtree A, P r.
```

- `l` contains rose trees on which `P` holds!

Running example

recap: termination checking in Coq

```
Definition list_eq A (A_eq : A → A → bool) :=
  fix rec (l1 l2 : list A) {struct l1} : bool :=
    match l1, l2 with
    | nil, nil => true
    | x :: xs, y :: ys => A_eq x y && rec xs ys
    | _, _ => false
    end.
```

```
Definition rtree_eq B (B_eq : B → B → bool) :=
  fix rec (t1 t2 : rtree B) {struct t1} : bool :=
    match t1, t2 with
    | Leaf x, Leaf y => B_eq x y
    | Node l1, Node l2 => list_eq (rtree B) rec l1 l2
    | _, _ => false
    end.
```

Problem n°2

```
Lemma list_eq_OK : ∀A (A_eq : A → A → bool),
         eq_axiom A A_eq →
       eq_axiom (list A) (list_eq A A_eq).
Proof. .. Qed. (* proof is opaque, hence hidden *)
```

```
Lemma rtree_eq_OK B B_eq (HB: eq_axiom B B_eq ) :
      eq_axiom (rtree B) (rtree_eq B B_eq)
:=
  fix IH (t1 t2 : rtree B) {struct t1} :=
    match t1, t2 with
    | Node l1, Node l2 => .. list_eq_OK (rtree B) (tree_eq B B_eq) IH l1 l2 ..
    | Leaf b1, Leaf b2 => .. HB b1 b2 ..
    | .. => ..
    end.
```

Problems

- Modularity of proofs
  - How to reuse proofs without piercing opacity
- Modularity of statements
  - How to express that `P` holds on “some subterms of ...” in a generic/schematic way

Solution to *both* problems

Unary Parametricity Translation of a type "`T`" is a predicate that means "`is_of_type_T`"

```
Inductive is_nat : nat → U :=
| is_O : is_nat 0
| is_S n (pn : is_nat n) : is_nat (S n).
```

```
Inductive is_list A (is_A : A → U) : list A → U :=
| is_nil : is_list A is_A nil
| is_cons a (pa : is_A a) l (pl : is_list A is_A l) : is_list A is_A (a :: l).
```

```
Inductive is_rtree A (is_A : A → U) : rtree A → U :=
| is_Leaf a (pa : is_A a) : is_rtree A is_A (Leaf A a)
| is_Node l (pl : is_list (rtree A) (is_rtree A is_A) l) : is_rtree A is_A (Node A l).
```

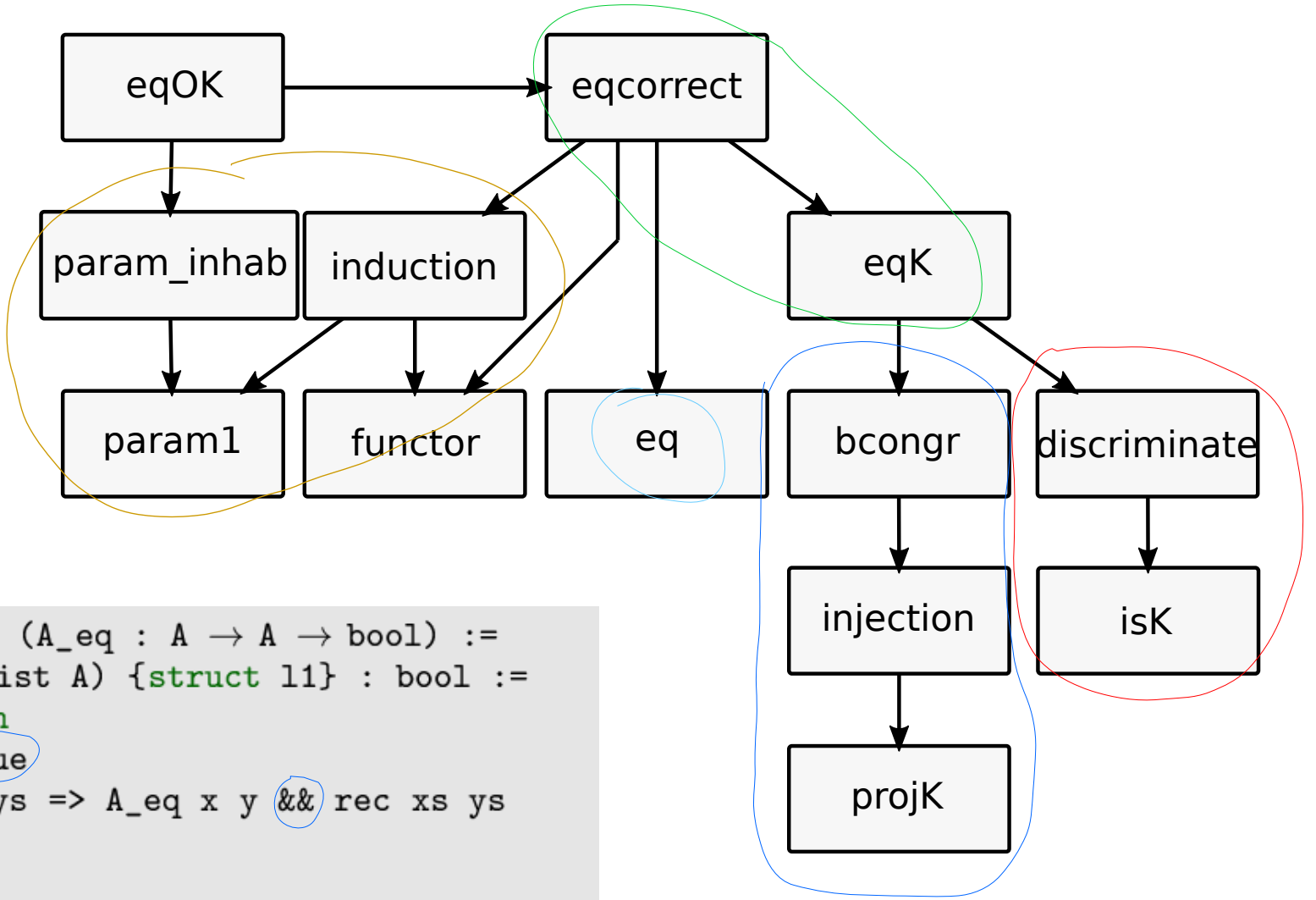
Derivation algorithm for Coq by Keller & Lasson [CSL 2012]

Induction principle for rose trees using unary parametricity

```
Lemma rtree_ind      A (P : rtree A → U):
    (∀a : A, P (Leaf A a)) →
    (∀l : list (rtree A), P (Node A l)) →
  ∀r : rtree A, P r.
```

```
Lemma rtree_induction A is_A (P : rtree A → U) :
    (∀a, is_A a → P (Leaf A a)) →
    (∀l, is_list (rtree A) P l → P (Node A l)) →
  ∀t, is_rtree A is_A t → P t.
```

Deriving proved equality tests




```
Definition list_eq A (A_eq : A → A → bool) :=
  fix rec (l1 l2 : list A) {struct l1} : bool :=
    match l1, l2 with
    | nil, nil => true
    | x :: xs, y :: ys => A_eq x y && rec xs ys
    | _, _ => false
    end.
```

Functoriality of param1

```
Inductive is_list A (is_A : A → U) : list A → U :=
| is_nil : is_list A is_A nil
| is_cons a (pa : is_A a) l (pl : is_list A is_A l) : is_list A is_A (a :: l).
```

```
Lemma is_list_funct A P Q : (∀a, P a → Q a) → ∀l, is_list A P l → is_list A Q l.
```

Inhab = reified typing

```
Definition nat_is_nat : ∀n : nat, is_nat n :=
  fix rec n : is_nat n :=
    match n as i return (is_nat i) with
    | 0 => is_0
    | S p => is_S p (rec p)
    end.
```

```
Definition list_is_list : ∀A (is_A : A → U), (∀a, is_A a) → ∀l, is_list A is_A l.
```

```
Definition rtree_is_rtree A PA (HPA : ∀a, PA a) :=
  fix IH t {struct t} : is_rtree A PA t :=
    match t with
    | Leaf a => is_Leaf A PA a (HPA a)
    | Node l => is_Node A PA l (list_is_list (rtree A) (is_rtree A PA) IH l)
    end.
```

Induction

```
Inductive is_rtree A (is_A : A → U) : rtree A → U :=
| is_Leaf a (pa : is_A a) : is_rtree A is_A (Leaf A a)
| is_Node l (pl : is_list (rtree A) (is_rtree A is_A) l) : is_rtree A is_A (Node A l).
```

Reading: if `t` validates `(is_rtree A is_A)` then `t` validates `P`

```
Definition rtree_induction A is_A P
   (HLeaf : ∀a, is_A a → P (Leaf A a))
   (HNode : ∀l, is_list (rtree A) P l → P (Node A l)) :
 ∀t, is_rtree A is_A t → P t
:=
 fix IH (t: rtree A) (x: is_rtree A is_A t) {struct x}: P t :=
   match x with
   | is_Leaf a pa => HLeaf a pa
   | is_Node l pl => (* pl: is_list (rtree A) (is_rtree A is_A) l *)
       HNode l (is_list_funct (rtree A) (is_rtree A is_A) P IH l pl)
   end.
```

No confusion: isK+discriminate

```
Lemma bool_discr : true = false → ∀T : U, T.
Lemma eq_f T1 T2 (f : T1 → T2) : ∀a b, a = b → f a = f b.
```

isK generates, for each constructor:

```
Definition is_Node A (t : rtree A) := match t with Node _ => true | _ => false end.
Definition is_Leaf A (t : rtree A) := match t with Leaf _ => true | _ => false end.
```

discriminate assembles:

```
                    H : (Node l = Leaf a)

(bool_discr (eq_f (rtree A) (rtree A) (is_Node A) H)) : ∀T : U, T
                              ..
            (is_Node A (Node l) = is_Node A (Leaf a))
```

No confusion: projK+injection

```
Lemma eq_f T1 T2 (f : T1 → T2) : ∀a b, a = b → f a = f b.
```

projK generates, for each constructor:

```
Definition get_cons1 A (d1 : A) (d2 : list A) (l : list A) : A :=
  match l with nil => d1 | x :: _ => x end.

Definition get_cons2 A (d1 : A) (d2 : list A) (l : list A) : list A :=
  match l with nil => d2 | _ :: xs => xs end.
```

injection assembles:

```
                        H:(x :: xs = y :: ys)
(eq_f (list A) A          (get_cons1 A x xs) (x :: xs) (y :: ys) H) : x = y
(eq_f (list A) (list A) (get_cons2 A x xs) (x :: xs) (y :: ys) H) : xs = ys
                                    ..
      get_cons2 A x xs (cons x xs) = get_cons2 A x xs (cons y ys)
```

Boolean congruence

```
Inductive reflect (P : U) : bool → U :=
| ReflectT (p : P) : reflect P true
| ReflectF (np : P → False) : reflect P false.
```

```
Lemma rtree_bcongr_Leaf A (x y : A) b :
  reflect (x = y) b → reflect (Leaf A x = Leaf A y) b

Lemma rtree_bcongr_Node A (l1 l2 : list (rtree A)) b :
  reflect (l1 = l2) b → reflect (Node A l1 = Node A l2) b
```

Boolean congruence

```
Inductive reflect (P : U) : bool → U :=
| ReflectT (p : P) : reflect P true
| ReflectF (np : P → False) : reflect P false.
```

```
Lemma list_bcongr_cons A
    (x y : A) b (hb : reflect (x = y) b)
    (xs ys : list A) c (hc : reflect (xs = ys) c) :
  reflect (x :: xs = y :: ys) (b && c) :=
match hb, hc with
| ReflectT eq_refl, ReflectT eq_refl => ReflectT eq_refl
| ReflectF (e : x = y → False), _ =>
    ReflectF (fun H : x :: xs = y :: ys =>
                e (eq_f (list A) A (get_cons1 A x xs) (x :: xs) (y :: ys) H))
| _, ReflectF e =>
    ReflectF .. (e (eq_f .. (get_cons2 ..) ..) ..) ..
end.
```

eqK: case split on the second term

```
Lemma rtree_eq_axiom_Node A (A_eq : A → A → bool) l1 :
    eq_axiom_at (list (rtree A)) (list_eq (rtree A) (rtree_eq A A_eq)) l1 →
  eq_axiom_at (rtree A) (rtree_eq A A_eq) (Node A l1)
:=
  fun H (t2 : rtree A) =>
  match t2 with
  | Leaf n =>
    ReflectF (fun abs : Node A l1 = Leaf A n =>
      bool_discr (eq_f (rtree A) bool (is_Node A) (Node A l1) (Leaf A n) abs) False)
  | Node l2 =>
      rtree_bcongr_Node A l1 l2 (list_eq (rtree A) (rtree_eq A A_eq) l1 l2) (H l2)
end.
```

eqcorrect: glues all pieces with induction

```
Lemma list_eq_correct A A_eq :
  ∀l, is_list A (eq_axiom_at A A_eq) l → eq_axiom_at (list A) (list_eq A A_eq) l.
```

```
Lemma rtree_eq_correct A A_eq : ∀t, is_tree A (eq_axiom A A_eq) t →
    eq_axiom (rtree A) (rtree_eq A A_eq)
:=
  rtree_induction A (eq_axiom_at A A_eq)
(*P*)      (eq_axiom_at (rtree A) (rtree_eq A A_eq))
(*HLeaf*) (rtree_eq_axiom_Leaf A A_eq)
(*HNode*) (fun l (Pl : is_list (rtree A) (eq_axiom_at (rtree A) (rtree_eq A A_eq)) l) =>
           rtree_eq_axiom_Node A A_eq l (list_eq_correct (rtree A) (rtree_eq A A_eq) l Pl)).
```

eqOK

```
Definition list_is_list : ∀A (is_A : A → U), (∀a, is_A a) → ∀l, is_list A is_A l.
```

```
Lemma list_eq_correct A A_eq :
  ∀l, is_list A (eq_axiom_at A A_eq) l → eq_axiom_at (list A) (list_eq A A_eq) l.
```

```
Lemma list_eq_OK A A_eq (A_eq_OK : eq_axiom A A_eq)
                 l : eq_axiom_at (list A) (list_eq A A_eq) l :=
  list_eq_correct A A_eq l (list_is_list A (eq_axiom_at A A_eq) A_eq_OK).
```

Implementation

- Embeddable
- Lambda
- Prolog
- Interpreter

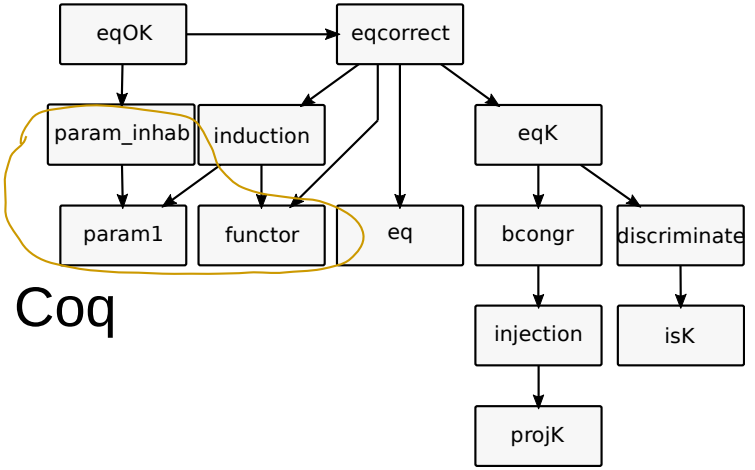
```
 134 derive/bcongr.elpi
 132 derive/eqcorrect.elpi
 179 derive/eq.elpi
 112 derive/eqK.elpi
  47 derive/eqOK.elpi
 172 derive/induction.elpi
  41 derive/isK.elpi
 207 derive/param1.elpi
 223 derive/param1_functor.elpi
 106 derive/param1_inhab.elpi
 147 derive/projK.elpi
1500 total
```

https://github.com/LPCIC/coq-elpi

(closing up on a release...)

Conclusion

- Contribution:
  - Schema for induction & proved equality tests for polynomial types and containers
  - Technique to tame the termination checker of Coq
  - Good test case for Coq-Elpi



- Future work:
  - Types with decidable indexes (fuel: trivial & contractible + functional extensionality)

- Related work:
  - recent work on deep data types and deep induction by Johann and Polonsky generalizes this pattern

Thanks!

Questions?