Deriving proved equality tests in Coq-elpi

Coq induction principles done right

Enrico Tassi
Université côte d'Azur - Inria
Enrico.Tassi@inria.fr

Abstract

We describe a procedure to derive from inductive type declarations equality tests and their correctness proofs. Programs and proofs are derived compositionally, reusing code and proofs derived previously. Finally we provide an implementation for the Coq proof assistant based on the Elpi extension language.

Keywords keyword1, keyword2, keyword3

1 Introduction

Modern typed programming languages come with the ability of generating boilerplate code automatically. Typically when a data type is declared a substantial amount of code is made available to the programmer at little cost, code such as a comparison function, a printing function, generic visitors etc. The `derive` directive of Haskell or the `ppx_deriving` OCaml preprocessor provide these features for the respective programming language.

The situation is less than ideal in the Coq proof assistant. It is capable of synthesizing the recursor of a datatype, that, following the Curry-Howard isomorphism, implements the induction principle associated to that datatype. It supports all datatypes, containers such as lists included, but generates a quite disappointing principle when a datatype *uses* a container.

For example, let's take the data type rose tree, where `U` stands for a universe (such as `Prop` or `Type`):

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

Its associated induction principle is the following one:

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

Remark that the recursive step, line 3, lacks any induction hypotheses on (the elements of) `l` while one would expect `P` to hold on each and every subtree. Coq provides an additional facility to synthesize equality tests and their proofs called `Scheme Equality`, but containers are not supported. The `decide equality` tactic can be manually iterated in order to generate a (proof) term implementing an equality tests for the type above, but this requires human intervention and also generates large terms since it inlines both the equality tests and the correctness proofs for all the containers used.

The state of affairs is particularly unfortunate because the need for the automatic generation of boilerplate code in the Coq ecosystem is higher than ever. Modern formal libraries structure their contents in a hierarchy of interfaces and some machinery such as type classes or canonical structures are used to link the abstract library to the concrete data the user is working on. For example first interface one is required to implement in order to use the theorems in Mathematical Components library on a type `T` is the `eqType` one, that requires a correct equality test on `T`.

In this paper we use the framework for meta programming based on Elpi developed by the author and we focus on the derivation of an instance of the `eqType` structure for a given data type. The aim is to provide a practical tool that is both automatic and avoids duplication and inlining whenever possible.

It turns out that generation of equality tests is relatively easy, while their proofs are hard, for two reasons. The first problem is that the standard induction principles generated by Coq, as depicted before, are too weak. In order to fix them one needs quite some extra boilerplate, such as the derivation of the unary parametricity translation of the data types involved. The second one is that termination checking is purely syntactic in Coq. Rephrased along the Curry-Howard isomorphism this means that in order to check that the induction hypothesis is applied to a smaller term, Coq may need to unfold all terms involved in the proof. This, in practice, it forces all proof to be transparent breaking modularity: a statement is no more contract, changing its proof script may impact users.

In this paper we describe a derivation procedure for the `eqType` structure where programs and proofs are both derived compositionally, reusing code and proofs derived previously. This procedure also confines the termination check issue, allowing proofs to be mostly opaque.

More precisely the contributions of this paper are the following ones:

- A technique to confine the termination checking issue by reifying the subterm relation checked by purely syntactic means by Coq's type checker. We apply it

PL'18, January 14–15, 2019, Lisbon, Protugal
2019.

to the proof of equality tests, but it is applicable to all proofs by structural induction.

- A modular and structured process to derive instances of the `eqType` structure and, en passant, stronger induction principles. Indeed procedure generates terms that can be (re)used separately.
- An actual implementation based on the Elpi extension language.

Straight to the point, by installing the `coq-elpi-derive` package[1] one obtains the following definition, where `reflect P b` is a predicate stating the equivalence between a predicate `P` and a boolean test `f`.

```
Definition axiom T f x :=
  ∀y, reflect (x = y) (f x y).
```

Then by issuing the `Elpi derive rtree` command one gets the following terms automatically synthesized out of the type declaration for `rtree`:

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

Lemma rtree_eq_OK : ∀A (fa : A → A → bool),
    (∀a, axiom A fa a) →
  ∀r, axiom (rtree A) (rtree_eq A fa) r.
```

The former is a (transparent) equality test for `rtree` while the latter is a (opaque) proof of its correctness.

The paper introduces the problem in section 2 by describing the shape of an equality test and of its correctness proof and explaining the modularity problem that stems for the termination checker of Coq. It then presents the main idea behind the modular derivation procedure in section 3. Section 4 describes all the bricks composing the derivation, while section 5 briefly describes the implementation in Elpi.

2 The problem: equality tests proofs meet syntactic termination checking

Recursors, or induction principles, are not primitive notions in Coq. The language provides constructors for fix point and pattern matching that work on any inductive data the user can declare.

For example to test two lists `l1` and `l2` for equality one first takes in input an equality test `fa` for the elements of type `A` and then performs the recursion:

```
1  Definition list_eq A (fa : A → A → bool) :=
2    fix rec (l1 l2 : list A) {struct l1} : bool :=
3      match l1, l2 with
4      | nil, nil => true
5      | x :: xs, y :: ys => fa x y && rec xs ys
6      | _, _ => false
7      end.
```

Lets new define the equality test for the `rtree` data type by reusing the test for lists:

```
8  Definition rtree_eq B (fb : B → B → bool) :=
9    fix rec (t1 t2 : rtree B) {struct t1} : bool :=
10     match t1, t2 with
11     | Leaf x, Leaf y => fb x y
12     | Node l1, Node l2 =>
13        list_eq (rtree B) rec l1 l2
14     | _, _ => false
15     end.
```

Note that `list_eq` is called passing as the `fa` argument the fixpoint `rec` itself (line 13). In order to check that the latter definition is sound, Coq looks at the body of `list_eq` to see weather its parameter `fa` is applied to a term smaller than `t1` (the argument labelled as decreasing by the `{struct t1}` annotation). Since `l1` is a subterm of `t1` and that `x` is a subterm of `l1`, the recursive call (line 5) is legit.

This is pretty reasonable for programs. We want both `list_eq` and `rtree_eq` to compute, hence their body matters to us. The fact that checking the soundness of `rtree_eq` requires inspecting the body of `list_eq` is not very annoying this time.

On the contrary proof terms are typically hidden to the type checker once they have been validated, for both performance and modularity reasons. In particular in order to make only the statement of theorems binding, while having the freedom to clean, refactor, simplify proofs without breaking the rest of the formal development.

Unfortunately the following attempt is unsuccessful if the body of `list_eq_OK` is hidden to the type checker:

```
1  Lemma list_eq_OK : ∀A (fa : A → A → bool),
2      (∀a, axiom A fa a) →
3    ∀l, axiom A (list_eq A fa) l.
4
5  Lemma rtree_eq_OK B fb (Hfb : ∀b, axiom B fb b) :
6    ∀t, axiom (rtree B) (rtree_eq B fb) t
7  :=
8    fix IH (t1 t2 : rtree B) {struct t1} :=
9    match t1, t2 with
10   | Node l1, Node l2 =>
11   ..list_eq_OK (rtree B) (tree_eq B fb) IH l1 l2..
12   | Leaf b1, Leaf b2 => ..Hfb b1 b2..
13   | .. => ..
14   end.
```

We pass `IH`, the induction hypothesis, as the witness that `(tree_eq B fb)` is a correct equality test (the argument at line 10). Without knowing how this argument is used by `list_eq_OK` Coq rejects the term.

The issue seems unfixable without changing Coq in order to use a more modular check for termination, for example based on sized types[2]. We propose a less ambitious but more practical approach here, that consists in putting the transparent terms that the termination checker is going to inspect outside of the main proof bodies so that they can be kept opaque.

The intuition is to reify the property the termination checker wants to enforce. It can be phrased as "x is a subterm of t

[1]See https://github.com/LPCIC/coq-elpi for the installation instructions

and has the same type". More in general we model "x is a subterm of t with property P" and "P" is going to be "beign of the same type" for subterms of "t" that are of the same type, while "P" will be an arbitrary property for terms of an arbitrary type such as the elements of a list.

This relation is naturally expressed by the unary parametricity translation of types [3]. Thanks to the work of Keller and Lasson [1] we have this translation for Coq.

3 The idea: separating terms and types via the unary parametricity translation

Given an inductive type `T` we systematically name `is_T` an inductive predicate describing the type of the inhabitants of `T`. This is the one for natural numbers:

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

The one for a container such as `list` is more interesting:

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

Remark that all the elements of the list validate `PA`.

When a type `T` is defined in terms of another other type `C`, typically a container, the `is_C` predicate shows up inside `is_T`. For example:

```
1 Inductive is_rtree A (PA: A → U): rtree A → U :=
2 |is_Leaf a (pa: PA a) : is_rtree A PA (Leaf A n)
3 |is_Node l (pl: is_list (rtree A) (is_rtree A PA) l):
4    is_rtree A PA (Node A l).
```

Note how line 3 expresses the fact that all elements in the list `l` validate `(is_rtree A PA)`.

Our intuition is that these predicates "reify" the notion of being of a certain type, structurally. What we typically write `(t : T)` can now be also phrased as `(is_T t)` as one would do in a framework other than type theory, such as a mono-sorted logic.

It turns out that the inductive predicate `is_T` corresponds to the unary parametricity translation of the type `T`. Keller and Lasson [1] give us an algorithm to synthesize these predicates automatically.

What we look for now is a way to synthesize a reasoning principle for a term `t` when `(is_T t)` holds.

3.1 Better induction principles

Let's have a look at the standard induction principles of lists.

```
Lemma list_ind A (P : list A → U) :
   P nil →
   (∀a l, P l → P (a :: l)) →
  ∀l : list A, P l.
```

This reasoning principle is purely parametric on `A`, no knowledge on any term of type `A` such as `a` is ever available.

What we want to obtain is a more powerful principle that let as choose some invariant for the subterms of type `A`. The one we synthesise is the following one, where the differences are underlined.

```
1 Lemma list_induction A (PA: A → U) (P: list A → U):
2    P nil →
3    (∀a (pa : PA a) l, P l → P (a :: l)) →
4   ∀l, is_list A PA l → P l.
```

Note the extra premise `(is_list A PA l)`: The implementation of this induction principle goes by recusion on of the term of this type and finds as an argument of the `is_cons` constructor the proof evince `(pa : PA a)` it feeds to the second premise (line 3). Our intuition is that all terms of type `(list A)` validate the property `P`, while all terms of type `A` validate the property `PA`.

More in general to each type we attach a property. For parameters we let the user choose (we take another parameter, `PA` here). For the type being analyzed, `list A` here, we take the usual induction predicate `P`. For terms of other types we use their unary parametricity translation.

Take for example the induction principle for `rtree`.

```
1 Lemma rtree_induction A PA (P : rtree A → U) :
2    (∀a, PA a → P (Leaf A a)) →
3    (∀l, is_list (rtree A) P l → P (Node A l)) →
4   ∀t, is_rtree A PA t → P t.
```

Line 3 uses `is_list` to attach a property to `l`, and given that `l` has type `(list (rtree A))` the property for the type parameter `(rtree A)` is exactly `P`. Note that this induction principle give us access to `P`, the property one is proving, on the subtrees contained in `l`.

3.2 Synthesizing better induction principles

It turns out that there is a systematic way to generate this better induction principle for a type `T`: trimming the standard elimination principle for the unary parametricity translation `is_T`.

We use the word trim to indicate the operation of turning a dependent elimination into a non-dependent one. The typing rule for dependent pattern matching lets one simultaneously replace both the eliminated term and the indexes of its type. The trimmed eliminator only replaces the indexes.

Lets take the non-trimmed eliminator for `is_list` and let's underline the parts that we want to remove.

```
1 Lemma is_list_ind A PA (P: ∀l, is_list A PA l → U) :
2    P nil (is_nil A PA) →
3    (∀a (pa: PA a) l (pl: is_list A PA l), P l pl →
4        P (a :: l) (is_cons A PA a pa l pl)) →
5   ∀l (pl: is_list A PA l), P l pl.
```

First `P` only talks about the index `l` (a list). Then, intuitively, redundant assumptions on variables are removed: we have both `P` and `is_list A PA` holding on `l` at line 3.

If we do the same operation on `is_tree` we get the elimination principle we need:

```
Lemma is_rtree_ind A PA (P: ∀t,is_rtree A PA t→ U):
  (∀a (Pa: PA a), P (Leaf A n) (is_Leaf A PA n Pa)) →
  (∀l, (Pl: is_list (rtree A) (is_rtree A PA) l),
    P (Node A l) (is_Node A PA l Pl)) →
 ∀t (pt: is_rtree A PA t), P t pt.
```

We now have a reasoning principle on `rtree` that is likely to be powerful enough to prove `rtree_eq` correct.

Unexpectedly these better induction principles also suggest a ways to confine the terms the termination checker of Coq has to inspect.

3.3 Isolating the syntactic termination check

As one expects, it is possible to prove that `is_T` holds for terms of type `T`.

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

For containers we can do so when the property on the parameter is true on the entire type.

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

Definition rtree_is_rtree : ∀A (PA : A → U),
  (∀a, PA a) → ∀t, is_rtree A PA t.
```

These facts are then to be used in order to satisfy the premise of our induction principles. We can build correctness proofs of equality tests in two steps.

For example, for natural numbers

```
1 Lemma nat_eq_sound :
2   ∀n, is_nat n → axiom nat nat_eq n
3 :=
4   nat_induction (axiom nat nat_eq) P0 PS.
5
6 Lemma nat_eq_OK n : axiom nat nat_eq n :=
7   nat_eq_sound n (nat_is_nat n).
```

Where `P0` and `PS` (line 2) stand for the two proof terms corresponding to the base case and the inductive step of the proof. We omit them because they play no role in the current discussion.

For containers things go smoothly. For example the correctness proof for the equality test on the `list A` data type can be proved as follows, where again line 6 omits the steps for `nil` and `cons`.

```
1 Lemma list_eq_sound A fa :
2   ∀l, is_list A (axiom A fa) l →
3     axiom list A (list_eq A fa)
4 :=
5   list_induction A (axiom A fa)
6     (axiom (list A) (list_eq A fa))
7     Pnil Pcons.
8
9 Lemma list_eq_OK A fa (Pfa : ∀a, axiom A fa a) l :
10   axiom (list A) (list_eq A fa)
11 :=
12   list_eq_sound l (list_is_list (axiom A fa) Pfa l).
```

What is more interesting is to look at the proof of the equality test for `rtree`. Note how the induction hypothesis `Pl` given by `rtree_induction` perfectly fits the premise of `list_eq_sound`.

```
Lemma rtree_eq_sound A fa :
  ∀t, is_tree A (axiom A fa) t →
    axiom (rtree A) (rtree_eq A fa)
:=
  rtree_induction A (axiom A fa)
    (axiom (rtree A) (rtree_eq Afa))
    PLeaf
    (λl Pl : is_list (rtree A)
                (axiom (rtree A) (rtree_eq A fa)) l =>
     .. list_eq_sound (rtree A) (rtree_eq A fa) l Pl ..).

Lemma rtree_eq_OK A fa (Pfa : ∀a, axiom A fa a) t :
  axiom (rtree A) (rtree_eq A fa)
:=
  rtree_eq_sound t (tree_is_tree A (axiom A fa) Pfa t).
```

Type checking the terms above does not require any term to be transparent. Actually they are applicative terms, there is no apparently recursive function involved.

Still there is no magic, we just swept the problem under the rug. In order to type check the proof of `tree_is_tree` Coq needs the body of the proof of `list_is_list`:

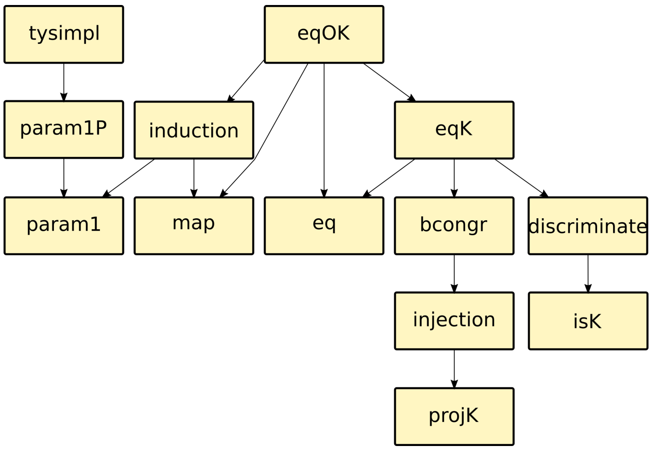
```
Definition rtree_is_rtree A PA (HPA : ∀a, PA a) :=
  fix IH t {struct t} : is_rtree A PA t :=
  match t with
  | Leaf a => is_Leaf A PA a (HPA a)
  | Node l =>
     is_Node A PA l
       (list_is_list (rtree A) (is_rtree A) IH l)
  end.
```

As we explained in section 2 Coq needs to know the body of `list_is_list` in order to agree that the argument `IH` is only used on sub terms of `t`.

Even we can't make the problem disappear (without changing the way Coq checks termination), we claim we confined the termination checking issue to the world of reified type information. The transparent proofs of `is_T` predicates are separate from the other, more relevant, proofs that can hence remain opaque as desired.

4 Anatomy of the derivation

The structure of the derivation is depicted in the following diagram. Each box represents a component deriving a complete term. Arrows depict the direct dependency of a box on the terms synthesized by another box.



4.1 Equality test

```
rtree_eq =
  λ(A : U) (eqA : A → A → bool) =>
    fix rec (t1 t2 : rtree A) {struct t1} : bool :=
    match t1, t2 with
    | Leaf a, Leaf b => eqA a b
    | Node l, Node s => list_eq (rtree A) rec l s
    | _, _ => false
    end
```

4.2 Parametricity

The `pram1` component is able to generate the unary parametricity translation of types and term following [1]. We already gave a few examples in section 3, we repeat here just the one for rose trees:

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

The `pram1P` component synthesizes proofs that terms of type `T` validate `is_T`. In section ?? we explained why these proofs needs to be transparent.

```
Definition rtree_is_rtree A (PA : A → U) :
  (∀x, PA x) → ∀t, is_rtree A PA t.
```

It is worth pointing out that the premise `(∀x, PA x)` can be proved not only for trivial `PA`. In particular, during induction on a term of type `T` the predicate being proved, say `P`, is true by induction hypothesis on (smaller) terms of type `T`. See for example line 10 in the proof of `rtree_eq_sound` in section ??.

4.3 Functoriality

The `map` components implements a double service.

For simple containers it synthesizes what one expects. For example:

```
Definition rtree_map A1 A2 :
  (A1 → A2) → rtree A1 → rtree A2.
```

The derivation covers polynomial types and is not needed in order to derive equality tests nor their correctness proofs. Its extension to indexed relations is, on the contrary, needed.

On indexed data types the derivation avoids to map the indexes and consequently type variable occurring in the types of the indexes. For example, mapping the `is_list` inductive relation gives:

fails on rtree

```
Lemma is_list_map : A PA PB,
  (∀a, PA a → PB a) →
    ∀l, is_list A PA l → is_list A PB l.
```

This property corresponds to the functoriality of the class of predicates `is_T` over the properties about the type parameters. Later in section we use this property to handle containers instantiated to base types, as in `list nat`.

In this optic map is a way to prove F A -> F B from A -> B.

4.4 Induction

```
Definition rtree_induction A PA P
    (HLeaf : ∀a, PA a → P (Leaf A a))
    (HNode : ∀l, is_list (rtree A) P l → P (Node A l)) :
  ∀t, is_rtree A PA t → P t
:=
  fix IH (t : rtree A) (x : is_rtree A PA 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 PA) l *)
     HNode l
       (is_list_map (rtree A) (is_rtree A PA) P IH l Pl)
  end.
```

Note how IH is the function being mapped trough. that is also saying that things that are trees validate P.

We map each constructor argument from its type to the one expected. The one expected is the input one whenre self is replaced by P.

scheme of an eq-test proof?

4.5 No confusion property

In order to prove that an equality test is correct one has to show the "no confusion" property, that is that constructors are injective and disjoint.

Lets start by provide they are disjoint. The simples form of this property can be expressed on bool:

```
Lemma bool_discr : true = false → ∀T : U, T.
```

This lemma is proved by hand once and forall. What the `isK` component synthesizes is a per-constructor test to be used in order to reduce a discrimination problem on type `T` to a discrimination problem on `bool`. For the rose tree data type `isK` generates the following consants:

```
Definition rtree_is_Node A (t : rtree A) : bool :=
  match t with Node _ => true | _ => false end.
Definition rtree_is_Leaf A (t : rtree A) : bool :=
  match t with Node _ => false | _ => true end.
```

The discriminate components uses one more trivial fact, `eq_f` in order to assemble these tests together with `bool_discr`.

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

From a term `H` of type `(Node l = Leaf a)` the discriminate procedure synthesizes a term of type `(∀T : U, T)` as follows:

```
1 bool_discr
2   (eq_f (rtree A) (rtree A) (rtree_is_Node A) H)
```

Note that the type of the term on line 2 is:

```
rtree_is_Node A (Node l) = rtree_is_Node A (Leaf a)
```

that is convertible to `(true = false)`.

In order to prove the injectivity on constructors the projK produce synthesizes a projector for each argument of each constructor. For example

```
Definition list_get_cons1 A (d1 : A) (d2 : list A)
  (l : list A) : A
:=
  match l with
  | nil => d1
  | cons x _ => x
  end.

Definition list_get_cons2 A (d1 : A) (d2 : list A)
  (l : list A) : list A
:=
  match l with
  | nil => d2
  | cons _ xs => xs
  end.
```

Each projector takes in input default values for each and every argument of the constructor. It is designed to be used by the injection procedure as follows. Given a term `H` of type `(cons x xs = cons y ys)`, in order to obtain a term of type `(xs = ys)` it generates:

```
eqf H (list_get_cons2 A x xs)
```

This term is easy to build given the type of `H` that contains the default values to be passed to the projector. Note that the type of the entire term is:

```
list_get_cons2 A x xs (cons x xs) =
  list_get_cons2 A x xs (cons y ys)
```

that is convertible to the desired `(xs = ys)`.

4.6 Congruence and `reflect`

In the definition of `axiom` we used the `reflect` predicate. It is a form of if-and-only-if specialized to link a proposition and a boolean test. It is defined as follows:

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

To prove the correctness of equality tests the shape of `P` is always an equation between two terms of the inductive type, most of the time constructors. When it find the same constructor on both sides, as in `(k x1 .. xn = k y1 .. y2)`, the equality tests calls appropriate equality tests for the arguments and forgets about the constructor. The bcongr component synthesizes lemmas helping this step. For example:

```
Lemma list_bcongr_cons A :
    ∀(x y : A) b, reflect (x = y) b →
    ∀(xs ys : list A) c, reflect (xs = ys) c →
  reflect (x :: xs = y :: ys) (b && c)

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
```

Note that these lemmas are not related to the equality test specific to the inductive type. Indeed they deal with the `reflect` predicate, but not with the `axiom` that we use every time we talk about equality tests.

The derivation goes as follows: if any of the premises about `reflect` is false, then the result is probed by `ReflectF` and the injectivity of constructors. If all premises are `ReflectT` their argument, an equation, can be used to rewrite the conclusion.

```
Lemma list_bcongr_cons A :
    ∀(x y : A) b, reflect (x = y) b →
    ∀(xs ys : list A) c, reflect (xs = ys) c →
  reflect (x :: xs = y :: ys) (b && c)
:=
  λx y b (hb : reflect (x = y) b)
      xs ys c (hc : reflect (xs = ys) c) =>
  match hb, hc with
  | ReflectT eq_refl, ReflectT eq_refl => ReflectT eq_refl
  | ReflectF (e : x = y → False), _ =>
    ReflectF
      (λH : (x :: xs) = (y :: ys) =>
       e (eq_f (list A) A (list_get_cons1 A x xs)
         (x :: xs) (y :: ys) H))
  | _, ReflectF e => ..injection..
end.
```

This is a sort of injection but specialized to Ks, to avoid n-ary injection.

This step corresponds to rec call + propagate return value.

4.7 Congruence and `axiom`

Now discrimination, the real switch, eg 2 different constructors. generate false in one case, call bcongr in the other.

```
Lemma rtree_axiom_Node A (f : A → A → bool) l1 :
    axiom (list (rtree A)) (list_eq (rtree A) (rtree_eq A f)) l1 →
  axiom (rtree A) (rtree_eq A f) (Node A l1)
:=
  λh (t2 : rtree A) =>
  match t2 with
```

```
  | Leaf n =>
   ReflectF (λabs : Node A l1 = Leaf A n =>
     bool_discr
       (eq_f (rtree A) bool (rtree_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 f) l1 l2) (h l2)
end.
```

4.8 Correctness

```
list_eq_sound : ∀A (fa : A → A → bool) l,
   is_list A (axiom A fa) l →
  axiom (list A) (list_eq A fa) l

rtree_eq_sound = λA (fa : A → A → bool) =>
  rtree_induction A (axiom A fa)
   (axiom (rtree A) (rtree_eq A fa)) (* P *)
   (rtree_eq.axiom.Leaf A fa)
   (λl (Pl : is_list (rtree a)
               (axiom (rtree a) (rtree_eq a fa)) l) =>
     rtree_eq.axiom.Node A fa l
       (list_eq_sound (rtree a) (rtree_eq a fa) l Pl))
: ∀(A : U) (fa : A → A → bool) (t : rtree A),
   is_rtree A (axiom A fa) t →
   axiom (rtree A) (rtree_eq A fa) t

Inductive histogram := Draw (color : bool) (bars : list nat).

Lemma histogram_induction (P : histogram → Type) :
   (∀c, is_bool c →
      ∀l, is_list nat is_nat l → P (Draw color l)) →
  ∀h, is_histogram h → P h.

Lemma histogram_axiom_Draw :
   ∀c, axiom bool bool_eq c →
   ∀l : list nat, axiom (list nat) (list_eq nat nat_eq) l →
  ∀h, axiom_at histogram histogram_eq (Draw c l) h.

Lemma histogram_eq_sound
  ∀(h : histogram), axiom (histogram A) (histogram_eq A fa) h
:=
  histogram_induction
   (axiom histogram histogram_eq)
   (λc (Pc : is_bool c)
       l (Pl : is_list nat is_nat l) =>
      histogram_axiom_Draw
        c (bool_eq_sound c Pc)
        l (list_eq_sound nat nat_eq
             l (is_list_map nat
                  is_nat (axiom nat nat_eq)
                  nat_eq_sound l Pl))).
```

4.9 tysimpl

```
nat_induction : ∀P : nat → U,
   P 0 → (∀p : nat, P p → P (S p)) →
  ∀n, is_nat n → P n

nat.induction = λP H0 HS n =>
   nat_induction P H0 HS n (is_natP n)
: ∀P : nat → U,
   P 0 → (∀p, P p → P (S p)) →
  ∀n, P n
```

5 implementation

Coq-elpi links a PL based on lambda Prolog and CHR. The latter fragment plays no role in this paper. lambda Prolog uses HOAS to describe Coq terms. logic programming has an obvious way of describing the db of knowledge, for example in eq-db.

api do provide access rw to the env

5.1 incompleteness and user intervention

mut ind no supported by elpi. while they make code longer we don't see which additional difficulty they could bring.

univ polymorphism not supported by elpi. no additional complexity.

eqtype is prerequisite for indexes decidable. the algorithm consists in packing inductive .. for contextual reasoning and finally projecting. As of today it is not fully automatized, but the chain can be used by manually providing the bloks that are missing.

6 related work

Coq: scheme equality (no containers in ty of constructors), decide equality works but one has to do the fix by hand + inlines everything + termination check.

Lean: rec/ind + discr Agda: no. Haskell: TODO. OCaml: ppx deriving. McBride: polytypes. Isabelle?

7 conclusion

not done before because of the lack of a platform that makes experimentation easy.

some bricks are reusable, eg in tactics.

call for size types.

Acknowledgments

We thank Maxime Denes and Cyril Cohen for many discussions shedding light on the subject; Cyril Cohen for the code implementing the parametricity translation in Elpi; Luc Chabassier for working on an early prototype of Elpi on the synthesis of equality tests, an experiment that convinced the author it was actually doable.

References

[1] Chantal Keller and Marc Lasson. 2012. Parametricity in an Impredicative Sort. In *CSL - 26th International Workshop/21st Annual Conference of the EACSL - 2012 (CSL)*, Patrick Cégielski and Arnaud Durand (Eds.), Vol. 16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Fontainebleau, France, 381–395. https://doi.org/10.4230/LIPIcs.CSL.2012.399

[2] Jorge Luis Sacchini. 2011. *On type-based termination and dependent pattern matching in the calculus of inductive constructions.* Theses. École Nationale Supérieure des Mines de Paris. https://pastel.archives-ouvertes.fr/pastel-00622429

[3] Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*. ACM, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

A Appendix

Text of appendix ...