Deriving proved equality tests, compositionally*

Subtitle†

Enrico Tassi‡
Position1
Department1
Institution1
City1, State1, Country1
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 comparison function, 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 (others?). It is capable of generating automatically the recursor of a datatype that corresponds to induction principle associated to that datatype, such as lists, but generates a quite disappointing principle when containers are used to define other types:

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

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

Coq provides a facility to synthesize comparison functions and their proofs called scheme equality, but it does not support containers.

The need for such tools is made urgent by the structure of modern formal libraries that are now based on hierarchies of interfaces. Machinery such as type classes or canonical structures are used to described the interfaces, and the user is expect to declare CS or TC instances in order to take advantage of these libraries. 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 comparison function on `T` as well as a proof of its correctness.

The reason for the status quo is probably caused by a concomitance of twofold. On one hand the very expressive type declarations makes it hard to implement a derivation that fully cover the theory. Even more when the derivation has to be proved correct. On the other hand meta-programming tools only recently started to appear.

In this paper we focus on the first, base, structure or MC, that is eqType, and we use the framework for metaprogramming based on elpi developed by the author.

It turns out that generation of eq tests is easy, while their proofs are hard. One problem is that containers, like lists, come with standard induction principles that are too weak.

eg

and from the fact that termination checking is purely sytnactic, hence proofs need to be transparent..

In this paper we describe a derivation proedure where Programs and proofs are derived compositionally, reusing code and proofs derived previously.

Contributions:

- a technique to separate, compartimentalize, thee termination checking issue by reifying the subterm relation checked by purely syntactic means by Coq
- modular and structured process to derive eqOK. each procedure generates terms that can be (re)used separately and
- actual implementation

```
Elpi derive rtree.

rtree.eq :
  ∀A, (A → A → bool) → rtree A → rtree A → bool
```

*with title note
†with subtitle note
‡with author1 note

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

```
rtree.eq.OK :
  ∀A (fa : A → A → bool) (r : rtree A),
    is_rtree A (axiom A fa) r →
    axiom (rtree A) (rtree.eq A fa) r

axiom := λT (eqb : T → T → bool) (x : T) =>
   ∀y : T, reflect (x = y) (eqb x y)
```

Strucure.

2 The problem: eq proofs meets syntactic termination checking

induction principles are not primitive, but encoded as fix + match, hence one can still prove

fix f => ... eqlistok f ..

This, in order to be "type checked" requires the system to inspect the full proof of eqlistok since The check is syntactic.

What the syntactic check tries to capture is the following relation: Is a subterm of the same type.

this is

3 decorrelating terms and types: unary parametricity by examples

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

for containers

```
Inductive is_list A (PA : A → Type) : list A → Type :=
| 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)
```

An instance of [1]

3.1 (truncated?) induction principle from tR

```
list_ind :
 ∀A (P : list A → Prop),
   P nil →
   (∀a l, P l → P (a :: l)) →
 ∀l : list A, P l

list.induction.principle :
 ∀A (PA : A → Type) (P : list A → Type),
   P nil →
   (∀a (pa : PA a) l, P l → P (a :: l)) →
 ∀l, is_list A PA l → P l

is_list_ind :
 ∀A (PA : A → Type) (P : ∀l, is_list A PA l → Prop),
   P nil (is_list.nil A PA) →
   (∀a (pa : PA a) l (pl : is_list A PA l), P l pl →
      P (a :: l) (is_cons A PA a pa l pl)) →
 ∀l (pl : is_list A PA s), P l pl *)
```

3.2 compartmentalizing of the syntactic check

```
is_rtreeP : ∀A (PA : A → Type),
  (∀x, PA x) → ∀t, is_rtree A PA t
```

this is needed to close the thing we recall here

```
rtree.eq.OK : ∀A (fa : A → A → bool) (s1 : rtree A),
  is_rtree A (axiom A fa) s1 →
  axiom (rtree A) (rtree.eq A fa) s1

nat.eq.OK : ∀n, axiom nat nat.eq n
```

4 structure

structure of the proof

4.1 param1 and param1P

```
Inductive is_rtree A (PA : A → Type) : rtree A → Type :=
| Leaf a (pa : PA a) : is_rtree A PA (Leaf A a)
| Node l (pl : is_list (is_rtree A) (is_rtree A PA) l) :
    is_rtree A PA (Node A l)

is_rtreeP : ∀A (PA : A → Type),
  (∀x, PA x) → ∀t, is_rtree A PA t
```

4.2 map

map for containers is what one expects.

```
rtree.map : ∀A1 A2, (A1 → A2) → rtree A1 → rtree A2
```

indexes are not mapped, and hence the variables used in their types are not mapped

predicates are made implications

fails on rtree

```
is_list.map : ∀A PA PB l,
  (∀x, PA x → PB x) → is_list A PA l → is_list A PB l
```

functoriality

4.3 induction

take tR and do the obvious induction for it, then truncate P.

4.4 isK and discriminate

```
rtree.is.Node : ∀A : Type, rtree A → bool
rtree.is.Leaf : ∀A : Type, rtree A → bool
eq_f : ∀T1 T2 (f : T1 → T2) a b, a = b → f a = f b.
bool_discr : true = false → ∀T : Type, T.
```

4.5 projK and injection

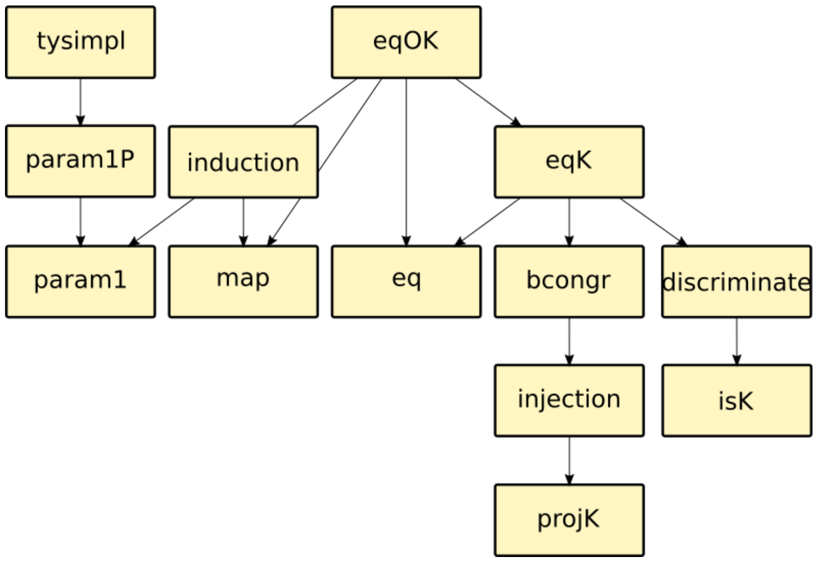
```
list.injection.cons1 : ∀A, A → list A → list A → A
list.injection.cons2 : ∀A, A → list A → list A → list A
```

to be used in conjunction with `eq_f` in case one has

```
H : cons x xs = cons y ys
eqf H (list.injection.cons2 A x xs) : xs = ys
```

The type is so that the function can be total and given the use case one can always pass the arguments of the constructor on the LHS of H.

```
list.injection.cons1 =
 λ(A : Type) (default1 : A) (default2 : list A) (l :
     list A) =>
   match l with
```




```
  | nil => default1
  | x :: _ => x
  end
```

4.6 bcongr

```
list.eq.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)

rtree.eq.bcongr.Leaf : ∀A (x y : A) b,
 reflect (x = y) b → reflect (Leaf A x = Leaf A y) b
```

4.7 eq

```
rtree.eq =
 λ(A : Type) (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
```

4.8 eqK and eqOK

```
rtree.eq.axiom.Node : ∀A (f : A → A → bool) l,
   axiom (list (rtree A)) (list.eq (rtree A) (rtree.eq A
        f)) l →
 axiom (rtree A) (rtree.eq A f) (Node A l)

list.eq.correct : ∀A (fa : A → A → bool) l,
   is_list A (axiom A fa) l →
 axiom (list A) (list.eq A fa) l

rtree.eq.correct = λA (fa : A → A → bool) =>
 rtree.induction.principle 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.correct (rtree a) (rtree.eq a fa) l Pl))
: ∀(A : Type) (fa : A → A → bool) (t : rtree A),
   is_rtree A (axiom A fa) t →
   axiom (rtree A) (rtree.eq A fa) t
```

4.9 tysimpl

```
nat.induction.principle : ∀P : nat → Type,
   P 0 → (∀p : nat, P p → P (S p)) →
 ∀n, is_nat n → P n

nat.induction = λP HO HS n =>
   nat.induction.principle P HO HS n (is_natP n)
: ∀P : nat → Type,
   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

A Appendix

Text of appendix …