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

We describe a procedure to derive equality tests and their correctness proofs from inductive type declarations in Coq. Programs and proofs are derived compositionally, reusing code and proofs derived previously.

The key steps are two. First, we design appropriate induction principles for data types defined using parametric containers. Second, we develop a technique to work around the modularity limitations imposed by the purely syntactic termination check Coq performs on recursive proofs. The unary parametricity translation of inductive data types turns out to be the key to both steps.

Last but not least, we provide an implementation of the procedure for the Coq proof assistant based on the Elpi [6] extension language.

2012 ACM Subject Classification Software and its engineering → General programming languages

Keywords and phrases Coq, Containers, Induction, Equality test, Parametricity translation

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.29

Supplement Material Source code of the Coq package: `https://github.com/LPCIC/coq-elpi`

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 an equality test, a printing function, generic visitors etc. For example 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 data type, that, following the Curry-Howard isomorphism, implements the induction principle associated to that data type. It supports all data types, containers such as lists included, but generates a quite weak principle when a data type *uses* a container. Take for example 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   ∀t : rtree A, P t.
```

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. Even a very basic recursive program such as an equality test cannot be proved correct using this induction principle. To

10th International Conference on Interactive Theorem Proving (ITP 2019).
Editors: John Harrison, John O'Leary, and Andrew Tolmach; Article No. 29; pp. 29:1–29:18
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

be honest, the Coq user is not even supposed to write equality tests by hand, nor to prove them correct interactively. Coq provides two facilities to synthesize equality tests and their correctness proofs called `Scheme Equality` and `decide equality`. The former is fully automatic but is unfortunately very limited, for example it does not support containers. The latter requires human intervention and generates a single, large, term that mixes code and proofs.

As a consequence, users often need to manually write induction principles, equality tests and their correctness proofs. This situation is very unfortunate because the need for the automatic generation of boilerplate code such as equality tests is higher than ever in the Coq ecosystem. All modern formal libraries structure their contents in a hierarchy of interfaces and some machinery such as Type Classes [18] or Canonical Structures [9] are used to link the abstract library to the concrete instances the user is working on. For example the first interface one is required to implement in order to use the theorems in the Mathematical Components library [10] on a type `T` is the `eqType` one, requiring a correct equality test on `T`.

In this paper we use the framework for meta programming based on Elpi [6, 19] developed by the author and we focus on the derivation of equality tests. It turns out that generating equality tests is easy, while their correctness proofs are hard to synthesize, for two reasons. The first problem is that the standard induction principles generated by Coq, as shown before, are too weak. In order to strengthen them one needs quite some extra boilerplate, such as the derivation of the unary parametricity translation of the data types involved. The second reason is that termination checking is purely syntactic in Coq: in order to check that the induction hypothesis is applied to a smaller term, Coq may need to unfold all theorems involved in the proof. This forces proofs to be transparent that, in turn, breaks modularity: A statement is no more a contract, changing its proof may impact users.

In this paper we describe a derivation procedure for equality tests and their correctness proofs 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 issue stemming from the purely syntactic termination check implemented by Coq out of the main proofs. In this paper we apply it to the correctness proof of equality tests, but the technique is applicable to all proofs that proceed by structural induction.
- A modular and structured process to derive proved equality tests and, en passant, stronger induction principles for inductive types defined using containers.
- An implementation based on the Elpi extension language for the Coq proof assistant.

By installing the `coq-elpi` package[1] and issuing the command `Elpi derive rtree` one gets the following terms synthesized out of the type declaration for `rtree`:

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

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

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

`reflect` is a predicate stating the equivalence between the proposition `(x = y)` and the boolean test `(f x y)`; `rtree_eq` is a (transparent) equality test and `rtree_eq_OK` is its (opaque) correctness proof under the assumption that the equality test `A_eq` is correct.

[1] See the supplementary material URL for the installation instructions.

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 briefly introduces the Elpi extension language and section 5 describes the full derivation.

2 The problem: opaque proofs v.s. 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 in order to test two lists `l1` and `l2` for equality one typically takes in input an equality test `A_eq` for the elements of type `A` and then performs the recursion:

```
1 Definition list_eq A (A_eq : 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 => A_eq x y && rec xs ys
6     | _, _ => false
7     end.
```

Coq accepts this definition because the recursive call is on `xs` that is a syntactically smaller term of the argument labelled as decreasing by the `{struct l1}` annotation.

We can define the equality test for `rtree` by reusing the equality test for lists:

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

Note that `list_eq` is called passing as the `A_eq` argument the fixpoint `rec` itself (line 12). In order to check that the latter definition is sound, Coq looks at the body of `list_eq` to see whether its parameter `A_eq` is applied to a term smaller than `t1`. Since `l1` is a subterm of `t1` and since `x` is a subterm of `l1`, then the recursive call `(rec x y)` at line 5 is legit.

The fact that checking `rtree_eq` requires inspecting the body of `list_eq` is not very annoying: we want both `list_eq` and `rtree_eq` to compute, hence their body matters to us.

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

For example, lets assume that `list_eq_OK` is an opaque proof that `list_eq` is correct.

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

It seems desirable to use this lemma in order to prove the correctness of `rtree_eq`, since it calls `list_eq`.

```
5  Lemma rtree_eq_OK B B_eq (HB: ∀b, eq_axiom B B_eq b) :
6    ∀t, eq_axiom (rtree B) (rtree_eq B B_eq) t
7  :=
8    fix IH (t1 t2 : rtree B) {struct t1} :=
9      match t1, t2 with
10     | Node l1, Node l2 => .. list_eq_OK (rtree B) (tree_eq B B_eq) IH l1 l2 ..
11     | Leaf b1, Leaf b2 => .. HB b1 b2 ..
12     | .. => ..
13     end.
```

Unfortunately this term is rejected: we pass `IH`, the induction hypothesis, as the witness that `(tree_eq B B_eq)` is a correct equality test (the argument at line 10 preceding `IH`) but Coq does not know how `list_eq_OK` uses this argument, since its body is opaque.

The issue seems unfixable without changing Coq in order to use a more modular check for termination, for example based on sized types [1]. 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 and has the same type". More in general we model "x is a subterm of t with property P".

3 The idea: put unary parametricity translation to good use

Given an inductive type `T` we 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 (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).
```

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

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

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

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

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 [22] of the type `T`. Keller and Lasson in [8] 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 Stronger induction principles for containers

Let's have a look at the standard induction principle 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 principle is parametric on `A`: no knowledge on any term of type `A` such as `a` is ever available. We want to synthesize a more powerful principle that lets us choose an invariant for the subterms of type `A` (the differences are underlined):

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

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

More in general to each type we attach a property. For parameters we let the user choose (we take another parameter, `is_A` here). For the type being analysed, `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 is_A (P : rtree A → U) :
2    (∀a, is_A a → P (Leaf A a)) →
3    (∀l, is_list (rtree A) P l → P (Node A l)) →
4  ∀t, is_rtree A is_A 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 gives us access to `P`, the property one is proving, on the subtrees contained in `l`.

3.1.1 Synthesizing stronger induction principles

We postpone a detailed description of the synthesis to section 5.4, here we just sketch how to build the type on the induction principle.

It turns out that the types of the constructors of `is_T` give us a very good hint on the type of the induction principle. The type of the first premise

```
   (∀a, is_A a → P (Leaf A a)) →
```

is exactly the type of the `is_Leaf` constructor

```
| is_Leaf a (pa : is_A a) : is_rtree A is_A (Leaf A n)
```

where `(is_rtree A is_A)` is replaced by `P`. The same holds for the other premise: its type can be trivially obtained from the type of `is_Node`.

Our intuition is that the inductive predicate `is_T` provides the same information that typing provides. Induction principles give `P` on (smaller) terms of the same type, that would be terms for which `is_T` holds. Given their inductive nature, `is_T` predicates are able to propagate the desired property inside parametric containers.

3.2 Isolating the syntactic termination check problem

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

```
Definition nat_is_nat : ∀n : nat, 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 `(T A)` we can prove `(is_T A is_A)` when `is_A` is trivial.

```
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 (is_A : A → U), (∀a, is_A a) → ∀t, is_rtree A is_A t.
```

These facts are then to be used in order to satisfy the premise of our induction principles.

Going back to our goal, we can build correctness proofs of equality tests in two steps. For example, for natural numbers we can generate two lemmas:

```
1 Lemma nat_eq_correct : ∀n, is_nat n → eq_axiom nat nat_eq n :=
2   nat_induction (eq_axiom nat nat_eq) PO PS.
3
4 Lemma nat_eq_OK n : eq_axiom nat nat_eq n :=
5   nat_eq_correct n (nat_is_nat n).
```

where `PO` 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 here for brevity.

For containers such as `(list A)` we can link the pieces in a similar way (at line 3 we omit the proofs for `nil` and `cons` as before).

```
1 Lemma list_eq_correct A A_eq : ∀l, is_list A (eq_axiom A A_eq) l →
2     eq_axiom list A (list_eq A A_eq) l :=
3   list_induction A (eq_axiom A A_eq) (eq_axiom (list A) (list_eq A A_eq)) Pnil Pcons.
4
5 Lemma list_eq_OK A A_eq (HA : ∀a, eq_axiom A A_eq a) l :
6     eq_axiom (list A) (list_eq A A_eq) l :=
7   list_eq_correct A A_eq l (list_is_list A (eq_axiom A A_eq) HA l).
```

It is interesting to look at a data type that uses a container such as `rtree`: the induction hypothesis `Pl` given by `rtree_induction` perfectly fits the premise of `list_eq_correct` (line 7).

```
1  Lemma rtree_eq_correct A A_eq : ∀t, is_tree A (eq_axiom A A_eq) t →
2      eq_axiom (rtree A) (rtree_eq A A_eq)
3  :=
4    rtree_induction A (eq_axiom A A_eq) (eq_axiom (rtree A) (rtree_eq Afa))
5      PLeaf
6      (fun l (Pl : is_list (rtree A) (eq_axiom (rtree A) (rtree_eq A A_eq)) l) =>
7       .. list_eq_correct (rtree A) (rtree_eq A A_eq) l Pl ..).
8
9  Lemma rtree_eq_OK A A_eq (HA : ∀a, eq_axiom A A_eq a) t :
10     eq_axiom (rtree A) (rtree_eq A A_eq) t :=
11   rtree_eq_correct A A_eq t (rtree_is_rtree A (eq_axiom A A_eq) HA 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 `rtree_is_rtree` Coq needs to look at the proof term of `list_is_list`:

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

As we explained in section 2 Coq would reject this term if the body of `list_is_list` was opaque.

Even if we cannot 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 theorems such as `T_is_T` are separate from the other, more relevant, proofs that can hence remain opaque as desired.

4 Elpi: an extension language for Coq

Elpi [6] is a dialect of λProlog [13], a higher order logic programming language. Elpi can be used as an extension language for Coq [19] in order to develop new commands in a programming language that has native support for bound variables.

Coq terms are represented in λ−tree syntax style [12] (sometimes also called Higher Order Abstract Syntax) reusing the binders of the programming language to represent the ones of Coq. For example, the term `(fun x => fact x)` is represented as `(lam (λx, app["fact",x]))`. We say that `app` and `lam` are object level term constructors standing for iterated (n-ary) application and unary lambda abstraction; `"fact"` is a constant and `x` is a variable bound by `λx`, that is the binder of the programming language. [2]

Programs are organized in clauses that represent both a data base of known facts and a set of rules to derive new facts out of known ones. For example one could use a relation named `eq-db` to link a type to its equality test.

```
eq-db "nat" "nat_eq".
eq-db (app["list", B]) (app["list_eq", B, B_eq]) :- eq-db B B_eq.
```

The first clause is a fact stating that `nat_eq` is the equality test for type `nat`. The second clause is an inference one and reads: the equality test for `(list B)` is `(list_eq B B_eq)` *if* `B_eq` is the equality test for `B`.

The `eq-db` data base can be queried for an equality test for, say, `(list nat)` by writing the goal `(eq-db (app["list", "nat"]) F)` where `F` is a variable to be filled in. By chaining the two clauses Elpi answers `(F = app["list_eq", "nat", "nat_eq"])` that reads back in the Coq syntax as `(list_eq nat nat_eq)`, the desired equality test for `(list nat)`.

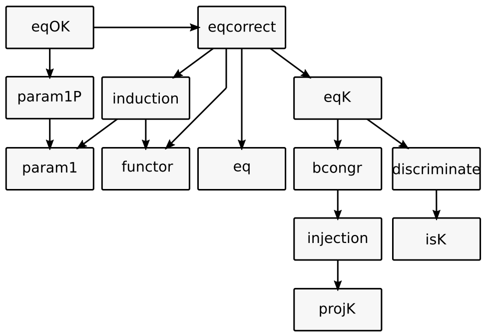
It is worth pointing out that in λProlog the set of clauses is dynamic: a program is allowed to add clauses inside a specific scope (typically the one of a binder) and the runtime collects them when the scope ends. As we will see, this feature is useful when a derivation takes place under an hypothetical context, e.g. when one assumes a parameter `A` and an equality test `A_eq`. No other feature of the Elpi language is relevant to this paper.

Finally, the integration of Elpi in Coq exposes to the extension language primitives to access the logical environment, e.g. to read an inductive data type declaration; to declare a new inductive type; to define a new constant; etc.

[2] Here we simplify a little the embedding and use strings to represent named terms, omitting their nodes: For example `nat`, an inductive type, is actually written `(indt "Coq.Init.Datatypes.nat")`, while `fact`, a defined constant, is written `(const "Coq.Arith.Factorial.fact")`.

5 Anatomy of the derivation

The structure of the derivation is depicted in the following diagram. Each box represents a component deriving a complete term. An arrow from component A to component B tells that the terms generated by B are used by the terms generated by A. The interfaces between these components are indeed types: one can replace the work done by each component with a few hand written terms, if necessary.



The *eq* component is in charge of synthesizing the program performing the equality test. The correctness proof generated by *eqcorrect* goes by induction on the first term of the two being compared and then goes on in a different branch for each constructor K. The property being proved by induction is expressed using `eq_axiom` that, as we will detail in section 5.6 is equivalent to a double implication. The *bcongr* component proves that the property is preserved by equal contexts, that is when the two terms are built using the same constructor. When they are not the program must return false and the equality be false as well: this is shown by *eqK*, that performs the case split on the second term. The no confusion property of constructor is key to this contextual reasoning. *projK* and *isK* generate utility functions that are then used by *injection* and *discriminate* to prove that constructors are injective and different. As we sketched in the previous sections the unary parametricity translation plays a key role in expressing the induction principle. The inductive predicate `is_T` for an inductive type `T` is generated by *param1* while *param1P* shows that terms of type `T` validate `is_T`. *functor* shows that `is_T` is a functor when `T` has parameters. This property is both used to synthesize induction principles and also to combine the pieces together in the correctness proof. The *eqOK* component hides the `is_T` relation from the theorems proved by *eqcorrect* by using the lemmas `T_is_T` proved by *param1P*.

5.1 Equality test

Synthesizing the equality test for a type `T` proceeds as follows. First the test takes in input each type parameter `A` together with an equality test `A_eq`. Then the recursive function takes in input two terms of type `T` and inspects both via pattern matching. Outside the diagonal, where constructors are different, it says `false`. On the diagonal it composes the calls on the arguments of the constructors using boolean conjunction. The code called to compare two arguments depends on their type: If it is `T` then it is a recursive call; if it is a type parameter `A` then we use `A_eq`; if it is another type it uses the corresponding equality test.

Let us take for example the equality test for rose trees:

```
1 Definition rtree_eq A (A_eq : A → A → bool) :=
2   fix rec (t1 t2 : rtree A) {struct t1} : bool :=
3     match t1, t2 with
4     | Leaf a, Leaf b => A_eq a b
5     | Node l, Node s => list_eq (rtree A) rec l s
6     | _, _ => false
7     end.
```

Line 5 calls `list_eq` since the type of `l` and `s` is `(list (rtree A))` and it passes to it `rec` since the type parameter of `list` is `(rtree A)`.

Here is an excerpt of Elpi code used to synthesize the body of the branches:

```
eq-db "A" "A_eq".
eq-db (app["rtree","A"]) "rec".
eq-db (app["list", B]) (app["list_eq", B, B_eq]) :- eq-db B B_eq.
```

The first clause says that `A_eq` is the equality test for type `A`, and is used to build the branch at line 4. The third clause, chained with the second one, combines `list_eq` with `rec` building the branch at line 5. The first two clauses are present only during the derivation of the body of the fixpoint, under the context formed by the type parameter `A`, its equality test `A_eq`, and the recursive call `rec` itself. Once the derivation is complete both clauses are removed from the data base and the following one is permanently added.

```
eq-db (app["rtree", B]) (app["rtree_eq", B, B_eq]) :- eq-db B_eq.
```

5.2 Parametricity

The *param1* component is able to generate the unary parametricity translation of types and terms following [8]. We already gave many examples in section 3. The *param1P* component synthesizes proofs that terms of type `T` validate `is_T` by a trivial structural recursion: constructor `K` is mapped to `is_K`. When `T` is a container we assume the triviality of the property on the type parameter. For example:

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

5.3 Functoriality

The *functor* component implements a double service. For non-indexed containers it synthesizes a simple map:

```
Definition list_map A B : (A → B) → list A → list B.
```

The derivation becomes more interesting when the container has indexes, e.g. when the container is a `is_T` inductive predicate. On indexed data types the derivation avoids to map the indexes and consequently all type variables occurring in the types of the indexes. For example, mapping the `is_list` inductive predicate gives:

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

This property corresponds to the functoriality of `is_list` over the property about the type parameter. Note that parameters of arity one, such as `P`, are mapped point wise.

As we did for the `eq-db` data base of equality tests, we can store these maps as clauses and use the data base later on in the *induction* and *eqcorrect* derivations. Here is an excerpt of Elpi code for this data base, that we call `funct-db`:

```
funct-db (app["is_list",A,P]) (app["is_list",A,Q]) (app["is_list_funct",A,P,Q,F]) :-
  funct-db P Q F.
```

Note that the terms involved are "point free", i.e. the first two arguments are terms of arity one, while the third term is of arity two. The identity is written as follows:

```
funct-db P P (lam (λa, lam (λp, p))).
```

This means that when one has a term `a` and a term `(p : P a)`, in order to obtain a term `(q : Q a)` he can query `funct-db` by asking Elpi to fill in `M` in `(funct-db "P" "Q" M)`. If the answer is `(M = f)` then the desired term is obtained by passing `a` and `p` to `f`, that is `(f a p : Q a)`.

5.4 Induction

In order to derive the induction principle for type `T` we first derive its unary parametricity translation `is_T`. The `is_T` inductive predicate has one constructor `is_K` for each constructor `K` of the type `T`. The type of `is_K` relates to the type of `K` in the following way. For each argument `(a : A)` of `K`, `is_K` takes two arguments: `(a : A)` and `(pa : is_A a)`. Finally the type of `(is_K a1 pa1 .. an pan)` is `(is_T (K a1 .. an))`.

The induction principle is synthesized by following these steps:

1. take in input each parameter `A1 is_A .. An is_A` of `is_T`.
2. take in input a predicate `(P : T A1 .. An → U)`.
3. for each constructor `is_K` of type
   `(∀A1 is_A .. An is_A, ∀a1 pa1 .. am pam, is_T A1 is_A .. An is_A (K a1 .. am))`
   take in input an assumption `HK` of type `(∀a1 pa1 .. am pam, P (K a1 .. am))`.
4. take in input `(t : T A1 .. An)`.
5. take in input `(x : is_T A1 is_A .. An is_A t)`.
6. perform recursion on `x` and a case split. Then in each branch
   a. bind all arguments of `is_K`, namely
      `(a1 : A1) (pa1 : is_A1 a1) .. (an : An) (pan : is_An an)`
   b. obtain `qai` by *mapping* the corresponding `pai` (as in `funct-db`, see below).
   c. return `(HK a1 qa1 .. an qan)`

Lets take for example the induction principle for rose trees:

```
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.
```

Note how, intuitively, the type of `HLeaf` can be obtained from the type of `is_Leaf` by replacing `(is_rtree A is_A)` with `P`.

Finally let us see how the second argument to `HNode` is synthesized. We take advantage of the fact that Elpi is a logic programming language and we query the data base `funct-db` as follows. First we temporarily register the fact that `IH` maps `(is_rtree A is_A)` to `P` obtaining, among others, the following clauses.

```
funct-db (app["is_rtree", "A", "is_A"]) "P" "IH".
funct-db (app["is_list",A,P]) (app["is_list",A,Q]) (app["is_list_funct",A,P,Q,F]) :-
  funct-db P Q F.
```

Then we query `funct-db` as follows:

```
funct-db (app["is_list", app["rtree","A"], app["is_rtree","A","is_A"]])
         (app["is_list", app["rtree","A"], "P"])
         Q.
```

The answer `(Q=app["is_list_funct",app["rtree","A"],app["is_rtree","A","is_A"],"P","IH"])` is exactly the second term we need to pass to `HNode` (once applied to `l` and `Pl`, line 10 above).

It is worth pointing out that, for the term to be accepted by the termination checker the map over `is_list` must be transparent.

To sum up the unary parametricity translation gives us the type of the induction principle, up to a trivial substitution. The functoriality property of the inductive predicates obtained by parametricity gives us a way to prove the branches.

5.5 No confusion property

In order to prove that an equality test is correct one has to show the so called "no confusion" property, that is that constructors are injective and disjoint (see for example [11]).

The simplest form of the property of being disjoint is expressed on `bool`:

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

This lemma is proved by hand once and for all. 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:

```
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.
```

The *discriminate* components uses one more trivial fact, `eq_f` [3], 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:

```
(bool_discr (eq_f (rtree A) (rtree A) (is_Node A) H)) : ∀T : U, T
```

Note that the type of the term `(eq_f .. H)` is `(is_Node A (Node l) = is_Node A (Leaf a))` that is convertible to `(true = false)`, the premise of `bool_discr`.

In order to prove the injectivity of constructors the *projK* component synthesizes a projector for each argument of each constructor. For the `cons` constructor of `list` we get:

```
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.
```

[3] `eq_f` is called `f_equal` in the Coq standard library.

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 `(x :: xs = y :: ys)` it synthesizes:

```
(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
```

These terms are easy to build given that the type of `H` contains the default values to be passed to the projectors. Note that the type of the second term is actually:

```
get_cons2 A x xs (x :: xs) = get_cons2 A x xs (y :: ys)
```

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

5.6 Congruence

In the definition of `eq_axiom` we use the `reflect` predicate [10]. It is a sort 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.
```

In our case the shape of `P` is always an equation between two terms of an inductive type, i.e. constructors. When the same constructor occurs in both sides, as in `(k x1.. xn = k y1.. y2)`, the equality test discards `k` and proceeds on each `(xi = yi)`. The *bcongr* component synthesizes lemmas helping to prove the correctness of 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 `eq_axiom` predicate that we use every time we talk about equality tests.

The derivation goes as follows: if any of the premises is false, then the result is proved 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 (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.
```

The elimination of hb and hc substitutes b and c by either true or false. In the branch at line 6 the boolean expression is hence (true && true) while the proposition is (x :: xs = x :: xs) given that the two equations (x = y) and (xs = ys) were eliminated as well.

The argument of e at line 9 is the term generated by the *injection* component. The branch at line 11, covering the case where the heads are equal but the tails different, is very close to lines 8 and 9 but for the fact that the projector for the second argument of cons is used, instead of the projection for the first one.

There are other ways one could have expressed these lemmas, for example by not mentioning the cons constructor explicitly but rather an abstract function k known to be injective on the first and second argument. Even if we find this presentation more appealing on paper, in practice we found no advantage and we hence opted for the current approach.

bcongr gives us lemmas to propagate equality and inequality only under the same constructor. *eqK* complements this work by proving eq_axiom also when the constructors differ.

Recall that the induction principle does a case split on one term, the first one of the two being compared. *eqK* generates a lemma for each constructor, to be used in the corresponding branch of the induction, that performs the case split on the second term being compared. This is the lemma generated for Node:

```
Lemma rtree_eq_axiom_Node A (A_eq : A → A → bool) l1 :
    eq_axiom (list (rtree A)) (list_eq (rtree A) (rtree_eq A A_eq)) l1 →
  eq_axiom (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.
```

Note that the code for the first branch is what *discriminate* synthesizes; while the code in the second branch is what *bcongr* generates.

5.7 Correctness

The *eqcorrect* component combines the induction principle generated by *induction* with the case split on the second term provided by *eqK*.

Let's recall the type of the correctness lemma for list_eq, of the induction principle and then let's analyse the proof of rtree_eq_correct:

```
Lemma list_eq_correct A (fa : A → A → bool) l,
    is_list A (eq_axiom A fa) l →
  eq_axiom (list A) (list_eq A fa) l.

Definition rtree_induction A is_A P
    (HLeaf : ∀y, is_A y → P (Leaf A y))
    (HNode : ∀l, is_list (rtree A) P l → P (Node A l)) :
  ∀t, is_rtree A is_A t → P t.

Lemma rtree_eq_axiom_Node A (f : A → A → bool) l1 :
    eq_axiom (list (rtree A)) (list_eq (rtree A) (rtree_eq A f)) l1 →
  eq_axiom (rtree A) (rtree_eq A f) (Node A l1).
```

The proof is a rather straightforward application of the induction principle to the property

```
eq_axiom (rtree A) (rtree_eq A fa)
```

Each branch is then proved by the corresponding lemma generated by *eqK* with only one caveat: one may need to adapt the induction hypothesis, `Pl` here, in order to make it fit the premise of the lemma generated by *eqK*. In this specific case the "adaptor" is `list_eq_correct`.

```
Lemma rtree_eq_correct A (fa : A → A → bool) :=
  rtree_induction A (eq_axiom A fa)
(*P*)     (eq_axiom (rtree A) (rtree_eq A fa))
(*HLeaf*) (rtree_eq_axiom_Leaf A fa)
(*HNode*) (fun l (Pl : is_list (rtree a) (eq_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)).
```

Logic programming provides a natural way to synthesize the adaptor. We load in the data base all the correctness proofs synthesized so far, as follows:

```
funct-db (app["is_list", A, is_A])
         (app["eq_axiom", app["list", A], app["list_eq", A, A_eq]]) R :-
  R = (app["list_eq_correct", A, A_eq]),
  funct-db is_A (app["eq_axiom", A, A_eq]).
```

This clause simply gives an operational reading to the type of `list_eq_correct`: the conclusion is true if the premise is. The only cleverness is to separate the premise in two parts, being a `(list A)` with property `is_A` and have `is_A` be a sufficient condition to prove that `A_eq` is correct. In this way clauses compose better: Search peels off just one type constructor at a time. Indeed we extend the `funct-db` predicate, instead of building a new one just for correctness lemmas, because functoriality lemmas are sometimes needed in addition to the correctness ones. Take for example this simple data type of a histogram.

```
Inductive histogram := Columns (bars : list nat).

Lemma histogram_induction (P : histogram → Type) :
    (∀l, is_list nat is_nat l → P (Columns l)) →
  ∀h, is_histogram h → P h.
```

Now look at the lemma synthesized by *eqK* for the `Columns` constructor.

```
Lemma histogram_eq_axiom_Columns l :
    eq_axiom (list nat) (list_eq nat nat_eq) l →
  ∀h, eq_axiom_at histogram histogram_eq (Columns l) h.
```

```
Lemma histogram_eq_correct h : eq_axiom histogram histogram_eq h :=
  histogram_induction
    (eq_axiom histogram histogram_eq)
    (fun l (Pl : is_list nat is_nat l) =>
       histogram_eq_axiom_Columns
         l (list_eq_correct nat nat_eq
              l (is_list_funct nat is_nat (eq_axiom nat nat_eq) nat_eq_correct l Pl))).
```

Note that the type of `Pl` is `(is_list nat is_nat)` and that it needs to be adapted to match `(is_list nat (eq_axiom nat nat_eq))`. The correctness lemma for `nat_eq`, namely `nat_eq_correct` of type `(∀n, is_nat n → eq_axiom nat nat_eq n)`, cannot be used directly but must undergo the `is_list_funct` functor.

5.8 eqOK

The last derivation hides the `is_T` predicate to the final user by combining the output of *eqcorrect* and *param1P*.

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

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

Both lemmas are needed. The former composes well and is needed if one defines a type using lists as a container. The latter is what the user needs in order to work with lists.

5.9 Assessment

The code is quite compact thanks to the fact that the programming language is very high level and that its programming paradigm is a good fit for this application.

On the average each components is about 200 lines of code. Simpler derivations like *projK*, *isK* or even *param1P* are under 100 lines.

Debugging this kind of code did not pose particular difficulties. The typical error results in the generated term being ill-typed. In that case the Coq type checker could be used to identify the culprit. Given how small the derivations are, it was simple to identify the lines generating the offending subterm.

The time required to design and develop the entire procedure amounts to approximatively six months, but spanned over more than one and a half year: most of the time has been spent improving the integration of Elpi in Coq in response to the experience gathered on this work. At the time of writing the Elpi integration in Coq does not support mutual inductive types, universe polymorphic definitions and primitive projections.

All derivations support polynomial types. Some derivations also support index data, e.g. *eq* is able to synthesize an equality test for vectors. Most of the derivations for contextual reasoning, such as *eqK* and *bcongr* do not support indexes.

6 Related work

Systems similar to Coq [20], e.g. Matita [2], Lean [5] and Isabelle [14] all generate induction principles automatically, with the exception of Agda [15], and some of them also the no confusion properties.

To our knowledge Isabelle is the only system that generates sensible induction principles and proved equality tests when containers are involved. As described in [4] the (co)datatype package is built on top of Bounded Natural Functors [21], a notion that makes the construction of (co)datatypes in Higher Order Logic compositional. Our starting point is very different since Coq, and type theory in general, internalizes the definitional mechanism for (co)datatypes. As a consequence a package like the one described in this paper cannot change it but only work around its eventual limitations. In particular the way Coq checks recursive functions for termination is a fixed, syntactic, non modular, criteria for which some alternatives have been studied (see for example [3, 16]) but never implemented. The non modular criteria applies to induction principles as well, since they are proved using recursion. It is a strength of the construction described in this paper to recover some modularity and hence be able to synthesize mechanically most of what [4] is able to synthesize.

Most Interactive Theorem Provers come with simple forms of Prolog-like automation, usually in the form of Type Classes. The user typically resorts to that in order to perform some of the inductive reasoning one needs in order to synthesize code in a type directed way. To our knowledge no ready-to-use package to synthesize equality tests and their proofs was written this way.

Some systems, notably Lean, come with a whole round meta programming framework. Still, to our knowledge, the primary application is the development of proof commands, not program/proof synthesis, in spite of the stunning similarity.

Coq provides two mechanisms strictly related to this work. The `Scheme Equality` command generates for a type `T` the code for the equality test (`T_eqb`) and a proof that equality is decidable on `T`. The proof internally uses the equality test, but its type does not:

```
T_eq_dec : ∀x y : T, {x = y} + {x <> y}
```

By unfolding the proof term, that is transparent, it should be possible to recover the fact that `T_eqb` is a correct equality test. Data types defined using containers are not supported. The `decide equality` tactic requires the user to start a lemma with a statement as the one depicted above. The tactic only performs one (case split) step and has to be iterated by hand. It does not remember which equalities were proved decidable before, it is up to the user to eventually share code. The proof term generated is, in a type theoretic sense, a program even if its code mixes the comparison test with its correctness proof. This proof is fully transparent, and inlines all the contextual reasoning steps such as injection and discrimination. As a result the term is very large and computationally heavy when run within Coq.

In the programming language world derivation is much more developed. The dominant approach is to provide some meta programming facilities, e.g. by providing a syntax to the declaration of types and then use the programming language itself to write derivations [17] that run at compile time as compiler plugins. Our approach is similar in a sense, since we work at the meta level on the syntax of types (and terms), but it is also very different since we pick a different programming language for meta programming. In particular we choose a very high level one that makes our derivations very concise and hides uninteresting details such as the representation of bound variables. The derivation described in the paper is the result of many failed attempts and we believe that the high level nature of the programming language we chose played an important role in the exploratory phase.

The link between the unary parametricity translation, also called predicate lifting, and induction principles was independently remarked by Kaposi and Kovács in [7].

7 Conclusion

We described a technique to derive stronger induction principles for Coq data types built using containers. We use the unary parametricity translation of a data type in order to fuel its induction principle, to thread an invariant on the contained when used as a container and finally to confine the modularity problems stemming from the termination check implemented in Coq. Finally we provide a Coq package deriving correct equality tests for polynomial inductive data types.

It is work in progress to extend the derivation to inductive types with decidable indexes. Preliminary work hints that indexes of base types such as `nat` pose no problem. On the contrary when indexes mention containers, that admit a decidable equality only if their contained does, the *param1P* component gets substantially more complex. In particular some notions of Homotopy Type Theory come in to play. For example the notion of being provable on the entire domain such as `(∀a : A, P a) → (∀t : T A, is_T A P t)` seems to require to be strengthened using the notion of contractibility (that is, the property should hold and its proof be unique), in order for the construction to compose well.

We also look forward to let the user tune the derivation process by annotating the type declarations. For example the user may want to skip certain arguments when generating the equality test, such as the integer describing the length of a sub vector in the `cons` constructor. The resulting equality test surely requires some user intervention in order to be proved correct, but it features a better computational complexity.

Finally, adding other derivations to the package seems appealing. For example the interface next to `eqType` in the hierarchy used in the Mathematical Component library is the one of countable types, i.e. types in bijection with natural numbers. The interface requires, roughly, a serialization function to another countable type, a tedious task that could be made automatic.

We are grateful to Maxime Denes and Cyril Cohen for the many discussions shedding light on the subject. We thank Cyril Cohen for writing the code of *param2* (binary parametricity translation), out of which *param1* was easily obtained. We also thank Damien Rouhling, Laurent Théry and Laurence Rideau for proofreading the paper. Finally we are indebted to 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 Andreas Abel. Type-based termination of generic programs. *Sci. Comput. Program.*, 74(8):550–567, 2009. `doi:10.1016/j.scico.2008.01.004`.

2 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita Interactive Theorem Prover. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23*, pages 64–69, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

3 Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. CICˆ: Type-Based Termination of Recursive Definitions in the Calculus of Inductive Constructions. In *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, pages 257–271, 2006. `doi:10.1007/11916277_18`.

4 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 93–110, Cham, 2014. Springer International Publishing.

5 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

6 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, Embeddable, λProlog Interpreter. In *Proceedings of LPAR*, Suva, Fiji, November 2015. URL: `https://hal.inria.fr/hal-01176856`.

7 Ambrus Kaposi and András Kovács. A Syntax for Higher Inductive-Inductive Types. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, pages 20:1–20:18, 2018. `doi:10.4230/LIPIcs.FSCD.2018.20`.

8 Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In Patrick Cégielski and Arnaud Durand, editors, *CSL - 26th International Workshop/21st Annual Conference of the EACSL - 2012*, volume 16 of *CSL*, pages 381–395, Fontainebleau, France, September 2012. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CSL.2012.399`.

9 Assia Mahboubi and Enrico Tassi. Canonical Structures for the Working Coq User. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, pages 19–34, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

10 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. draft, v1-183-gb37ad7, 2018.

11 Conor McBride, Healfdene Goguen, and James McKinna. A Few Constructions on Constructors. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs*, pages 186–200, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

12 Dale Miller. Abstract Syntax for Variable Binders: An Overview. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz

Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic — CL 2000*, pages 239–253, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

13 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012. doi:10.1017/CBO9781139021326.

14 Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

15 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

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

17 Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002. doi:10.1145/636517.636528.

18 Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag. doi:10.1007/978-3-540-71067-7_23.

19 Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). CoqPL, January 2018. URL: https://hal.inria.fr/hal-01637063.

20 The Coq Development Team. The Coq Proof Assistant, version 8.8.0, April 2018. doi:10.5281/zenodo.1219885.

21 Dmitry Traytel, Andrei Popescu, and Jasmin C. Blanchette. Foundational, Compositional (Co)Datatypes for Higher-Order Logic: Category Theory Applied to Theorem Proving. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 596–605, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/LICS.2012.75.

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