Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin's data types with Coq-Elpi

Benjamin Grégoire, Jean-Christophe Léchenet, Enrico Tassi

Practical and sound equality tests, automatically

Deriving eqType instances for Jasmin's data types with Coq-Elpi

Benjamin Grégoire
Jean-Christophe Léchenet
Enrico Tassi
benjamin.gregoire@inria.fr
jean-christophe.lechenet@inria.fr
enrico.tassi@inria.fr
Université Côte d'Azur, Inria
France

Abstract

In this paper we describe the design and implementation of feqb, a tool that synthesizes sound equality tests for inductive data types in the dependent type theory of the Coq system.
Our procedure scales to large inductive data types, as in hundreds of constructors, since the terms and proofs it synthesizes are linear in the size of the inductive type. Moreover it supports some forms of dependently typed arguments and sigma types pairing data with proofs of decidable properties. Finally feqb handles deeply nested containers without requiring any human intervention.

1 Introduction and motivation

In this paper we lower the amount of manual work one needs to do in order to reuse formal libraries. All modern libraries are organized around interfaces for which the user is expected to provide an instance. For example the `ssreflect` component of the Mathematical Components library is based on the `eqType` interface, that stands for types equipped with an equality test. In order to use that library in the development of the Jasmin compiler [2–4], we need to provide an equality test (and its soundness proof) for each and every data type. As of today most of these `eqType` instances are built by hand.

In this paper we develop the feqb tool which is able to synthesize automatically these instances *in practice*. In order to achieve this, we overcome the following difficulties.

Conference'17, July 2017, Washington, DC, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Tractable space and time. If one looks at the instruction set of a mainstream CPU like x86, one soon discovers that it counts from 1503 to 2034 instructions[1]. Modelling the instruction set as an inductive data type feels natural, but when doing so one immediately incurs in performance problems [10–12]: the `Scheme Equality` procedure was measured to be cubic in the number of constructors! Today Jasmin's backend groups instructions by family (parametrized over the word size) and lacks all floating point instructions, hence the inductive type modelling the instruction set has only 150 constructors. Even so, automatically synthesizing an `eqType` using `Scheme Equality` is slow and won't scale well when we will extend the compiler backend.

Super linear complexity is not completely unexpected, since the *natural code* for the equality test is already not linear. The user facing syntax for inspecting two terms looks linear, but internally Coq stores, and type checks, a term which is quadratic in the number of constructors.

user facing syntax

```
match x, y with
| K1, K1 ⇒ true
| K2, K2 ⇒ true
...
| KN, KN ⇒ true
| _, _ ⇒ false
end
```

internal representation

```
match x with
| K1 ⇒ match y with
       | K1 ⇒ true
       | K2 ⇒ false
       ...
       | KN ⇒ false
       end
| K2 ⇒ match y with
       | K1 ⇒ false
       | K2 ⇒ true
       ...
       | KN ⇒ false
       end
...
```

As we will detail in this paper, the soundness proof about a quadratic piece of code has the same bad complexity as a lower bound.

In order to make "instruction-set large" data types tractable, we show a way to describe in dependent type theory *a double case analysis which is linear in the number of constructors.*

[1]According to different sources and ways of counting families of intructions. See for example https://stefanheule.com/blog/how-many-x86-64-instructions-are-there-anyway/.

Values in types, proofs in terms. In dependent type theory values can occur inside types. The typical example is the type of numbers which are smaller than a bound `m` which is a parameter of their type `ord m`:

```
Inductive ord (m: nat) := mkOrd : ∀n, n < m → ord m.
```

In order to constrain the inhabitants of that type, type theory lets one pair terms with proofs. An inhabitant of `ord m` would be a number `n` paired with a proof of `n < m`. The type of the pairing constructor `mkOrd` uses the dependent function space in an essential way, since the value of the first argument must occur in the type of the second (in the statement of the proof that constrains it).

Types like this are used in Jasmin in many places, for example in the data type for machine words and consequently the data types for values.

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

It is well known that dependent pairs are not the best friend to equality tests, since the proof component may not be unique (there may be many different ways to prove the same property). Luckily there is a large class of properties which have unique (canonical) proofs: all the decidable ones. In that case *the equality test is expected to ignore proof arguments*, and the soundness proof to use the canonicity property of the discarded argument to show that it is sound to ignore it in the first place.

Our tool feqb handles value parameters and the dependent function space, and is hence able to synthesize and prove an equality test for the `word` and `value` types above.

Comfortable use of containers. It is well know to Coq users that the default induction principle generated by Coq for the following data type is too weak to be useful.

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

Such a principle does not give the induction hypothesis on the terms of type `instr` which are inside `cmd` (that is a shorthand for `list (info * instr)`) because of the use of the list and pair containers. It is common practice to manually write (and prove) *deep* principles like this one:

```
Check cmd_rect : ∀Pf Pi Pfi P,
  (* constructor of pair *)
  (∀ (inf: info) (i: instr), Pf inf → Pi i → Pfi (inf,i))
  (* constructors of list *)
  → P [::]
  → (∀ (i: info * instr) (c: cmd), Pfi i → P c → P (i :: c))
  (* constructors of instr *)
  ...
  (* conclusion *)
  → ∀c: cmd, P c
```

All additional predicates and premises are needed in order to propagate the induction property `P` to sub terms, deep inside containers. While this approach "works", in the sense that it lets one prove properties like the soundness of an equality test by induction on the `instr` type, it is not automatically generated by Coq and it is not modular. Unfortunately one needs to prove, at each use, that the pairing constructor and the `cons` one preserve the property. It goes without saying that one would like to prove this kind of result only once for each container, or even better have the system prove it automatically.

It is only recently that Johan and Polonski [13] and Tassi [16] showed a way to synthesize automatically induction principles which are both deep and modular. In this work we follow the schema introduced in [16] and we extend it to types parametrized by values.

1.1 Existing tools and their limits

There are many tools for Coq which can synthesize equality tests and their proofs, but unfortunately none covers all the types we need. In Figure 1 we compare the following tools: Equations [15] is a compiler for dependent pattern matching which also provides a `Derive` command; Deriving [7] is a tool able to synthesize an `eqType` via a reflexive procedure; The Coq system [17] provides the `decide equality` tactic and the `Scheme Equality` command; Coq-Elpi is an extension language for Coq that comes bundled with a `derive` [16] command.

We compare how tools deal with the induction principle which is needed in order to prove the equality test sound. In particular we check if the tool is able to use or generate deep principles and if they are modular. We then check if the tool provides the equality test and its proof as separate concepts (which helps keeping proofs outside statements[2]), if the test is heterogeneous w.r.t. parameters (i.e. lets one state a comparison between words of different sizes) and we try to give a lower bound on the size of the generated equality test and proof given the size of the inductive type. Finally we check which features are supported, in terms of the type of the arguments of the constructors, which may use containers, be dependent or be (canonical) proofs.

We provide some benchmarks in Section 7 on the real time complexity of the synthesis. It is hard to claim the complexity on paper, rather than experimentally, partially because no tool documents it and also because all tools use the Coq type checker as a black box, and type checking time can be, in principle, more than exponential. Of course if a tool generates a term which is quadratic in its input, we can consider n^2 to be a lower bound.

Equations was sometimes difficult to classify, since it defines a lot of tools that one can use interactively to define

[2]To see why this matters read http://gallium.inria.fr/blog/coq-eval/

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

Figure 1. Tools comparison

the equality test and proof. For the comparison, we really focused on the `Derive` command and its ability to generate the equality test automatically.

We use ☝ for positive answers but where some manual intervention is needed. For example, if the data type uses containers then the user has to write by hand a (non modular) deep induction principle, which then Deriving can use.

Scheme equality exposes to the user a single term of type `{ x = y } + { x <> y }`; with (1) we signal that the equality test and its correctness proof are internally separate, so with some extra work the user can access them.

Finally (2) signals that dependent arguments are accepted only if they are used as a parameter of another `eqType`, or inside an irrelevant argument. This limitation of our tool is detailed in Section 3.

1.2 Contributions and paper structure

The contributions of this work are:

- a schema for equality tests and their proofs which is *linear in the size of the inductive type*, that is the number of constructors multiplied by their arity
- the feqb tool that implements the schema and supports *containers* and *dependent records* with *irrelevant arguments*

Section 2 introduces the main idea behind linear equality tests. Section 3 describes the exact class of inductive types feqb can handle. Section 4 analyzes in more details the schema and its complexity. Section 5 describes how the schema deals with dependent arguments, like the ones one finds in sigma types. Section 6 describes how we use deep induction principles to deal with containers. Section 7 details a bit more the implementation and shows a few benchmarks comparing experimentally feqb with the alternatives.

An archive containing the source code of feqb, along with the test bed and raw results of the benchmarks, is available in the supplementary material[3].

[3]Its ongoing integration in Coq-Elpi can be found at https://github.com/LPCIC/coq-elpi/pull/319

2 A schema for linear terms and proofs

The main difficulty in writing a linear-sized decision procedure for equality is the quadratic factor that appears naturally by comparing constructors pairwise. This quadratic factor is amplified by the correctness proof of the equality test.

To illustrate the root of this complexity problem, we look at the equality test on the type `nat` of Peano numbers.

2.1 The complexity problem

The natural code to test the equality of `x` and `y` starts by matching `x` and then, in each branch, it does the same on `y`. This is the root of a quadratic factor.

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

Unfortunately, the two matches on `y` cannot be easily shared since they have very different branches.

The soundness proof for `eqb` is made of two parts, completeness and correctness.

```
Lemma eqb_OK x y : eqb x y ↔ x = y.
Proof. by split ⇒ [|→]; [ apply: eqb_correct | apply: eqb_refl ]. Qed.
```

Remark that we use the `is_true` implicit coercion allowing to cast boolean to proposition. So in the previous lemma, `eqb x y` should be read as `is_true (eqb x y)`, which is equivalent to `eqb x y = true`.

The proof of completeness (or reflexivity) is not problematic. It is linear in the number of constructors, since the induction principle `nat_ind` has one premise per constructor.

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

On the other hand, the proof of correctness is problematic, since it follows the definition of `eqb` which is quadratic.

```
Lemma eqb_correct : forall x y, eqb x y → x = y :=f
fun x : nat ⇒
  nat_ind (fun n : nat ⇒ forall y : nat, eqb n y → n = y)
    (fun y : nat ⇒
    match y as n return (eqb 0 n → 0 = n) with
    | 0 ⇒ fun _ : eqb 0 0 ⇒ eq_refl 0
    | S q ⇒ ... (* absurd : eqb 0 (S q) → false *)
    end)
  (fun p (IH : forall z : nat, eqb p z → p = z) y ⇒
    match y as n return (eqb (S p) n → S p = n) with
    | 0 ⇒ ... (* absurd : eqb (S p) 0 → false *)
    | S q ⇒ fun h : eqb (S p) (S q) ⇒
        (* since : eqb (S p) (S q) ≈ eqb p q *)
        f_equal S (IH q h)
    end)
```

As with the definition of eqb, the two branches for y cannot be shared.

A possible solution to fix the complexity of eqb could be to optimize the internal representation of Coq terms so that the match node can share common branches (the false ones). In particular allowing a default branch would already be enough to solve the problem. Even so, this optimization would not solve the problem in the correctness proof, since the absurd branches have slightly different proofs (p and q are bound in different places).

2.2 The linear equality test

In the definition of eqb, the pattern matches on x and y do two different things at once: they compare the names of the constructors and, if they are equal, they access the fields of the constructor and then compare the sub terms.

The key idea to avoid the quadratic blowup is to keep these two steps separate: we first compare the names of constructors by giving them a first class representation, and only then we access and compare sub terms recursively. Remark that the first step lets us *not generate* the nested match nodes.

We start by defining some auxiliary functions whose signature is given in Figure 2, which provides the generic interface to build our linear test on a type A.

The first requirement is a tag function which associates to each constructor a first class value (a positive number) standing for its name. For nat this can be done by associating 1 to 0 and 2 to S.

```
Definition nat_tag n := match n with 0 ⇒ 1 | S _ ⇒ 2 end.
```

The second function, fields, allows to access the fields of each constructor. Naturally the type of the function is dependent, since each constructor may have different fields. To write this dependent type we define a fields_t function associating to each tag the type of its fields. For the type nat, one obtains:

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

Definition nat_fields n : nat_fields_t (nat_tag n) :=
  match n with
  | 0 ⇒ tt
  | S p ⇒ p
  end.
```

Given these two functions and a function named eqb_fields to compare the fields of a given tag, we can write the generic definition eqb_body (again in Figure 2). It first computes the two tags of x and y; then checks their equality and, if they are equal, it checks the equality of the fields f1 and f2. Remark that we need a type cast on f2 to convince Coq that its type is fields_t t1.

The generic code cannot perform the recursion itself, since Coq does not know yet that A is inductively defined. It is the code specific to the inductive type, nat here, which will perform the recursion.

The eqb_fields function for nat is parametrized by the recursive call (rec) to test the equality of sub terms:

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

Remark that we have been able to share all the branches returning false into one, the right branch of eqb_body (Figure 2). Furthermore all the other branches are defined using nat_eqb_fields, that has one branch per constructor.

To conclude the definition, we would like to tie the knot with a fixpoint like so:

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

Here we step on a well known limitation of the Coq system: syntactic termination checking. Indeed this definition is rejected, since Coq is not able to see that the recursive call will be performed on smaller terms. Remark that the use of size types [1, 5, 6, 9] would completely solve this problem, but Coq does not feature this termination checking technique.

The solution to live in harmony with the current termination checker is to inline a little bit the definition of tag and eqb_body in the definition of the fixpoint itself. To do so, we modify a little the definition of eqb_body: instead of taking two arguments of type A, the first argument, namely x, is given by its (pre computed) tag and fields.

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

Now the following definition is accepted by Coq, since it sees that p is smaller than n1, and, by unfolding the definition of eqb_body and nat_eqb_fields, that the only recursive call to nat_eqb is on p.

```
Fixpoint nat_eqb (n1 n2 : nat) {struct n1} :=
  let body :=
    eqb_body nat nat_tag nat_fields_t nat_fields
     (nat_eqb_fields nat_eqb) in
  match n1 with
  | 0 ⇒ body 1 tt n2
  | S p ⇒ body 2 p n2
  end.
```

2.3 The linear soundness proof

We now explain how the correctness proof of `nat_eqb` can be expressed as a term linear in the size of the inductive. We start by introducing some generic definitions that allow to share part of the proof between different types.

First, we assume a `construct` function that builds an element of type `A` from its tag and its fields. Its specification `constructP` indicates that `fields` and `construct` cancel out.

```
Variable construct : ∀ t, fields_t t ⟶ option A.
Variable constructP : ∀ a, construct (fields a) = Some a.
```

The instantiation for `nat` leads to the following definition and trivial proof:

```
Definition nat_construct t : nat_fields_t t ⟶ option nat :=
  match t with
  | 1 ⟹ fun _ ⟹ Some 0
  | 2 ⟹ fun p ⟹ Some (S p)
  | _ ⟹ fun _ ⟹ None
  end.

Lemma nat_constructP (a: nat) : nat_construct (nat_fields a) = Some a.
Proof. by case. Qed.
```

Remark that the `construct` function is partial. Since tags are encoded as `positive`, there are tags that correspond to no constructor. For those tags, we need to construct a witness of `A`. This can be easily done for `nat`, it is sufficient to take `0`. But in general this is not possible, for example consider the type:

```
Inductive sum (L R : Type) : Type :=
  | inl : L ⟶ sum L R | inr : R ⟶ sum L R.
```

To build a witness using the constructor `inl` (resp. `inr`), we need an element of `L` (resp. `R`). In both cases, we cannot respect the signature. The use of the `option` type allows to solve this problem by returning `None`.

In order to share the part of the proof which talks about the generic definition `eqb_body`, we provide the following lemma.

```
Definition eqb_fields_correct_on (a:A) :=
  ∀ f : fields_t (tag a),
    eqb_fields (fields a) f ⟶ Some a = construct f.

Lemma eqb_body_correct a1 :
  eqb_fields_correct_on a1 ⟶
  ∀ a2,
    eqb_body tag fields_t fields eqb_fields
      (tag a1) (fields a1) a2 ⟶ a1 = a2.
```

```
  Section Core.
  Variable A : Type.

  Variable tag : A ⟶ positive.
  Variable fields_t  : positive ⟶ Type.
  Variable fields : ∀ (a:A), fields_t (tag a).
  Variable eqb_fields : ∀ t, fields_t t ⟶ fields_t t ⟶ bool.

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

Figure 2. Core interface for generic equality

Since the definition of `eqb_body` is parametrized by the function `eqb_fields`, its correctness proof assumes a property on it, namely `eqb_fields_correct_on`, which must be proved for each instantiation of the parameter. The proof is done by case analysis on `Pos.eq_dec` and then relies on `constructP`.

The correctness proof of `nat_eqb` can be built by a simple induction on its first argument.

```
Definition eqb_correct_on (f: A ⟶ A ⟶ bool) (a1: A) :=
  ∀ a2, f a1 a2 ⟶ a1 = a2.

Lemma nat_eqb_correct n1 : eqb_correct_on nat_eqb n1.
Proof.
  pose h :=
    (@eqb_body_correct nat nat_tag nat_fields_t nat_fields
       nat_construct nat_constructP (nat_eqb_fields nat_eqb)).
  elim ⟹ [ | n1 hrec] n2 /=.
  + (* eqb_body nat_fields (nat_eqb_fields nat_eqb) tt n2 ⟶ 0 = n2 *)
    apply (h 0) ⟹ f /=.
    (* true ⟶ Some 0 = Some 0 *)
    done.
  (* eqb_body nat_fields (nat_eqb_fields nat_eqb) n1 n2 ⟶ S n1 = n2 *)
  apply (h (S n1)) ⟹ f /=.
  (* nat_eqb n1 f ⟶ Some (S n1) = Some (S f) *)
  by move⟹ h1; apply (f_equal (fun p ⟹ Some (S p))); apply hrec
Qed.
```

The resulting proof term is linear in the number of constructors.

```
Definition nat_eqb_correct :=
let h := eqb_body_correct nat_constructP
           (eqb_fields:=nat_eqb_fields nat_eqb) in
nat_ind (fun n : nat ⟹ forall n2 : nat, nat_eqb n n2 ⟶ n = n2)
  (h 0 (fun f0 ⟹ (fun _: true ⟹ erefl (Some 0))))
  (fun (n1 : nat) (hrec : forall n2 : nat, nat_eqb n1 n2 ⟶ n1 = n2) ⟹
   h (S n1)
     (fun (fS : nat_fields_t (nat_tag (S n1))) (h1 : nat_eqb n1 fS) ⟹
      f_equal (fun p ⟹ Some (S p)) (hrec fS h1))).
```

We chose to illustrate the schema on a simple type such as `nat` in order to focus on the complexity deriving from the number of constructors. Section 4 will extend this schema to inductive types whose constructors have multiple fields, and show how to keep the size of each proof branch linear in the number of fields.

3 Specification

We build our tool feqb on top of Coq-Elpi which lets one implement Coq commands (or tactics) in Elpi, a dialect of λProlog. This programming language is well suited to manipulate syntax trees which contain binders and provides Higher Order Abstract Syntax (HOAS) data types. Figure 3 shows the HOAS of inductive data types handled by feqb.

$\mathbb{I}$ represents a definition of a Coq inductive type. tparam stands for the typical parameter of containers, like the `A` in `list A`. Since we use HOAS, its argument is a function that binds a type expression (of type $\mathbb{E}$) in the rest of the inductive definition. vparam stands for parameters which are not types but rather values, like `size` in `word size`; similarly it binds that value in the rest of the declaration. The last constructor of $\mathbb{I}$, inductive, encodes the declaration of the constructors. This time the function binds the inductive itself in the declaration

$$\begin{array}{lll}
\mathbb{I} & := & \\
\mid & \text{tparam} & : (\mathbb{E} \to \mathbb{I}) \to \mathbb{I} \\
\mid & \text{vparam} & : \mathbb{E} \to (\mathbb{E} \to \mathbb{I}) \to \mathbb{I} \\
\mid & \text{inductive} & : (\mathbb{E} \to \mathit{list}\ \mathbb{C}) \to \mathbb{I} \\
\mathbb{C} & := & \\
\mid & \text{regular} & : \mathbb{E} \to \mathbb{C} \to \mathbb{C} \\
\mid & \text{dependent} & : \mathbb{E} \to (\mathbb{E} \to \mathbb{C}) \to \mathbb{C} \\
\mid & \text{irrelevant} & : \mathbb{E} \to \mathbb{C} \to \mathbb{C} \\
\mid & \text{stop} & : \mathbb{C} \\
\mathbb{E} & := & \\
\mid & \text{app} & : \mathbb{S} \to \mathit{list}\ \mathbb{E} \to \mathbb{E}
\end{array}$$

```
Inductive list (A : Type) := nil | cons : A → list A → list A.
Record word (nbits: wsize) := mkWord {
  w : Z;
  _ : 0 <=? w && w <? 2^(wsize_size nbits)
}.
```

$$\begin{array}{ll}
\text{"}\mathit{list}\text{"} \triangleq & \text{tparam}\ \lambda a. \\
& \quad \text{inductive}\ \lambda l. \\
& \quad\quad [\text{stop} \\
& \quad\quad ;\text{regular}\ a\ (\text{regular}\ l\ \text{stop})] \\
\text{"}\mathit{word}\text{"} \triangleq & \text{vparam}\ (\text{app}\ \text{"}\mathit{wsize}\text{"}\ [])\ \lambda i. \\
& \quad\quad \text{inductive}\ \lambda t. \\
& \quad\quad\quad [\text{dependent}\ (\text{app}\ \text{"}Z\text{"}\ []) \\
& \quad\quad\quad\quad \lambda n.\text{irrelevant}\ (\ldots)\ \text{stop}]
\end{array}$$

Figure 3. The AST of data types declarations

of the constructors, which can hence mention it to declare recursive data.

A constructor declaration, represented by the node $\mathbb{C}$, carries the "list" of the types of the arguments of the constructor:

- regular stands for the usual, non dependent, argument like the `A` and `list A` arguments of `cons`;
- dependent stands for an argument that can be used in the type of the following arguments, like the `w` of `mkWord`. The second argument of dependent is indeed a function that encodes that dependency.
- irrelevant stands for an argument which can be ignored by the equality test.
- stop stands for the empty list.

All arguments and vparam carry a $\mathbb{E}$ which stands for a Coq type. For simplicity it pairs a symbol name and possibly empty list of arguments, leaving out all non applicative expressions. Remark that some of the excluded type expression can be encoded in $\mathbb{E}$ by using definitions, for example `A → B` can be encoded to `arrow A B` thanks to a suitable definition of `arrow`.

Remark that this syntax does not cover indexed types like vector, since feqb does not handle them directly. If the indexes are in an `eqType`, one can rephrase them in Ford style, with a parameter and an extra equation per constructor [14, Section 3.5], and this class of inductive types is supported by feqb.

Coq inductive declarations are translated into $\mathbb{I}$, during this translation we identify as irrelevant equalities over `bool`[4]. The right part of Figure 3 provides the translation for the Coq inductive declaration of `list` and `word`.

This syntax is further constrained by the predicates depicted in Figure 4. The first predicate, $\Gamma \vdash I \in \mathbb{V}_{\mathbb{I}}$, ensures the validity of object of type $\mathbb{I}$. The context Γ is used to carry assumptions about bound variables. This validation predicate ensures well formedness and that the feqb tool will be able to generate the equality test and the corresponding proofs. For example, in the rule (*tp*), the type parameter x is assumed to be valid, i.e. to be a type with a decidable equality. This assumption comes from the fact that the equality function that the tool will generate for $\lambda x.I$ will be parametrized by an equality function on x. Similarly, the rule (*i*) also adds the assumption but this time this will come from the recursive definition. The assumption is not present in the rule (*vp*), i.e. value parameter does not need to be equipped with a decidable equality.

The second predicate, $\Gamma \vdash K \in \mathbb{V}_{\mathbb{C}}$, is for the well formedness of constructor declarations. It ensures that all type expressions ($\mathbb{E}$) occurring in the type of the constructor are equipped with a decidable equality, rules (*r*) and (*d*), or are irrelevant[5], rule (*ir*).

The third predicate, $\Gamma \vdash E \in \mathbb{V}_{\mathbb{E}}$, is for validity of $\mathbb{E}$. The rule (*ax*) is straightforward. The rule (*e*) ensures that the arguments passed to the inductive i satisfy their requirements ($\Gamma \vdash I \approx A$). The predicate $i \triangleq I$ ensures that the inductive declaration corresponding to the name i has already been processed by the tool and that I is its HOAS representation.

4 Complexity, theory and practice

In this section, we show that terms and proofs can easily grow quadratic in the number of fields when polymorphic containers or lemmas are applied. We explain the solutions we found to avoid this blowup.

4.1 Taming quadratic definitions

The first quadratic factor shows up in the definition of `fields_t` and `fields` functions. The natural idea is to use the product[6] of the types of the arguments of a constructor to encode `fields_t` and the pairing constructor to package the sub terms returned by `fields`. The following code uses this encoding for the `tree` data type:

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

Definition tree_fields_t p :=
```

[4]This can be easily extended to other irrelevant arguments like SProp

[5]Recall that the current implementation is limited to equality over bool.

[6]Dependent product for dependent arguments

$$\frac{\Gamma, x \in \mathbb{V}_{\mathbb{E}} \vdash I \in \mathbb{V}_{\mathbb{I}}}{\Gamma \vdash (\mathsf{tparam}\ \lambda x.I) \in \mathbb{V}_{\mathbb{I}}}\,(tp) \qquad \frac{\Gamma \vdash I \in \mathbb{V}_{\mathbb{I}}}{\Gamma \vdash (\mathsf{vparam}\ \lambda x.I) \in \mathbb{V}_{\mathbb{I}}}\,(vp) \qquad \frac{\Gamma, i \in \mathbb{V}_{\mathbb{E}} \vdash K_j \in \mathbb{V}_{\mathbb{C}} \quad \forall j \in \{1\dots n\}}{\Gamma \vdash (\mathsf{inductive}\ \lambda i.[K_1;\dots;K_n]) \in \mathbb{V}_{\mathbb{I}}}\,(i)$$

$$\frac{\Gamma \vdash E \in \mathbb{V}_{\mathbb{E}} \quad \Gamma \vdash K \in \mathbb{V}_{\mathbb{C}}}{\Gamma \vdash \mathsf{regular}\ E\ K \in \mathbb{V}_{\mathbb{C}}}\,(r) \qquad \frac{\Gamma \vdash E \in \mathbb{V}_{\mathbb{E}} \quad \Gamma \vdash K \in \mathbb{V}_{\mathbb{C}}}{\Gamma \vdash \mathsf{dependent}\ E\ \lambda x.K \in \mathbb{V}_{\mathbb{C}}}\,(d) \qquad \frac{\mathsf{Irrelevant}\ T \quad \Gamma \vdash K \in \mathbb{V}_{\mathbb{C}}}{\Gamma \vdash \mathsf{irrelevant}\ T\ K \in \mathbb{V}_{\mathbb{C}}}\,(ir) \qquad \frac{}{\Gamma \vdash \mathsf{stop} \in \mathbb{V}_{\mathbb{C}}}$$

$$\frac{(x \in \mathbb{V}_{\mathbb{E}}) \in \Gamma}{\Gamma \vdash x \in \mathbb{V}_{\mathbb{E}}}\,(ax) \qquad \frac{i \triangleq I \quad \Gamma \vdash I \approx A}{\Gamma \vdash \mathsf{app}\ i\ A \in \mathbb{V}_{\mathbb{E}}}\,(e)$$

$$\frac{}{\Gamma \vdash \mathsf{inductive}\ \_ \approx \epsilon} \qquad \frac{\Gamma \vdash E \in \mathbb{V}_{\mathbb{E}} \quad \Gamma \vdash I \approx A}{\Gamma \vdash \mathsf{tparam}\ \lambda x.I \approx E :: A} \qquad \frac{\Gamma \vdash I \approx A}{\Gamma \vdash \mathsf{vparam}\ \lambda x.I \approx E :: A}$$

Figure 4. Validity of an eqType

```
  match p with
  | 1 ⇒ unit
  | 2 ⇒ prod (prod nat tree) tree
  | _ ⇒ unit
  end.

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

Remark that in the definition of `tree_fields` the number of occurrences of `tree` grows non linearly. This comes from the fact that the `pair` constructor takes explicitly the type of its two arguments (Coq implements polymorphism à la system-$\mathcal{F}$). We abandon the generic pair container in favor of ad-hoc, monomorphic, containers: for each constructor of the inductive type, we generate a record type gathering its fields. For `tree` this leads to the following declarations:

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

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

Definition tree_fields x :=
  match x with
  | Leaf ⇒ Box_Leaf
  | Node n l r ⇒ Box_Node n l r
  end.

Definition tree_eqb_fields rec t :
  tree_fields_t t → tree_fields_t t → bool :=
  match t with
  | 1 ⇒ fun 'Box_Leaf 'Box_Leaf ⇒ true
  | 2 ⇒ fun '(Box_Node n1 l1 r1) '(Box_Node n2 l2 r2) ⇒
          nat_eqb n1 n2 && rec l1 l2 && rec r1 r2
  | _ ⇒ fun _ _ ⇒ false
  end.
```

4.2 Taming quadratic proofs

A similar phenomenon occurs at proof generation time. We illustrate it on the correctness proof of the `tree` data type. For each constructor `c` of `tree` we build a proof term of type:

```
eqb_fields_correct_on (tree_eqb_fields tree_eqb) c
```

When `c` is `Node n1 l1 r1`, this statement is convertible to:

```
∀ f2: box_for_Node,
  tree_eqb_fields tree_eqb (Box_Node n1 l1 r1) f2 →
    Some (Node n1 l1 r1) = tree_construct f2
```

After destructing `f2` into `Box_Node n2 l2 r2` we have:

```
nat_eqb n1 n2 && rec l1 l2 && rec r1 r2 →
  Some (Node n1 l1 r1) = Some (Node n2 l2 r2)
```

The proof can be easily concluded by chaining these three steps:

1. breaking the conjunction into separate hypotheses
2. turning each hypothesis[7] into an equation, namely `n1 = n2`, `l1 = l2` and `r1 = r2`
3. finally rewriting with these equations

The resulting goal can be trivially proved by reflexivity:

```
Some (Node n2 l2 r2) = Some (Node n2 l2 r2)
```

Unfortunately this proof strategy can lead to a quadratic proof term if the first and third steps are not implemented carefully.

4.2.1 Splitting *n* conjunctions at once.

Let `a`, `b`, and `c` be booleans, `P` a proposition and `h` a proof for `a → b → c → P`. A natural proof term for `a && b && c → P` is:

```
andE (andb a b) c P (andE a b (c → P) h)
```

where `andE`, the standard "eliminator" of `&&`, has type:

```
∀ a b (P: Prop): (a → b → P) → a && b → P
```

It is easy to check that the iterated application of `andE` leads to a quadratic proof term. It is certainly possible to solve the problem using `let` binding to share common sub terms, but we prefer to use a different solution which exploits the ability of type theory to describe statements by recursion on values. The proof terms built this way are simple to synthesize automatically, and this technique will also help solving the other problem arising in the third step.

We first define a recursive function `implies` as follows:

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

Given a list of boolean `[b0; ...; bn]` and a predicate `P`, the term `implies [b0; ...; bn] P` reduces to `b0 → ... → bn → P`.

[7]The first equation comes from the correctness of `nat_eqb`, the two other by induction hypothesis.

Then we define the recursive function `allr` that computes the conjunction of its arguments:

```
Fixpoint allr (l: list bool) :=
  match l with
  | [::] ⇒ true
  | b :: l ⇒ b && allr l
  end.
```

Finally, we can express and prove the elimination lemma for the n-ary conjunction:

```
Lemma impliesP (l: list bool) (P:Prop) : implies l P → allr l → P.
```

The proof term `impliesP [a;b;c] P h` has type `a && b && c → P` and it is linear in the number of conjuncts.

4.2.2 Rewriting *n* equations at once.

The proof step of rewriting one equality is usually encoded in dependent type theory using the standard elimination principle of equality:

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

If we have the following three equations

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

and we want to rewrite all of them, right to left, in order to solve the goal

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

we have to synthesize the following proof term:

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

This proof term is quadratic in the number of arguments of `Node`, since each rewriting step "copies" the entire goal, which is linear in the number of arguments of `Node`.

We solve this problem by writing an n-ary elimination principle for equality, again using the ability of dependent type theory to express types by recursion on values.

```
Fixpoint p_type (T: list Type) :=
  match T with
  | [::] ⇒ Prop
  | A :: T ⇒ A → p_type T
  end.

Fixpoint eq_ind_r_n (T: list Type) : p_type T → p_type T → Prop :=
  match T with
  | [::] ⇒ fun p1 p2 ⇒
      p1 → p2
  | A::T ⇒ fun p1 p2 ⇒
      ∀ (a1 a2:A), a1 = a2 → eq_ind_r_n T (p1 a1) (p2 a2)
  end.

Lemma eq_ind_r_nP (T: list Type) (p: p_type T) : eq_ind_r_n T p p.
```

Given a list of types such as `[:: nat; tree; tree]`, `p_type` reduces to the following type

```
p_type [:: nat; tree; tree] = nat → tree → tree → Prop
```

and in turn `eq_ind_r_n` reduces to this statement

```
eq_ind_r_n [:: nat; tree; tree] =
  fun p1 p2 : nat → tree → tree → Prop ⇒
    ∀ (a1 a2 : nat), a1 = a2 →
    ∀ (b1 b2 : tree), b1 = b2 →
    ∀ (c1 c2 : tree), c1 = c2 →
      p1 a1 b1 c1 → p2 a2 b2 c2
```

which is a 3-way rewriting principle specialized on the types of the sub terms of `Node`. Using this principle, we can synthesize this linear proof term:

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

Note that this time the goal is copied only once, and it is put under a number of lambda abstractions which is linear in the number of arguments of `Node`.

The `eq_ind_r_nP` proof device is similar to the `nary_congruence` one which is part of the `ssreflect` library, although simpler to explain since it recurses on the list of types, rather than their number.

5 Dealing with dependent types

In order to simplify the synthesis of the equality test for dependent arguments, we systematically generate equality tests which are *heterogeneous* on parameters. For example the equality test for `word` has the following shape:

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

As expected the specification constrains `s1` and `s2` to be the same[8]:

```
word_eqb_OK : ∀s (w1 w2 : word s), word_eqb s s w1 w2 ↔ w1 = w2
```

This way of doing lets one synthesize a simpler equality test for the `value` data type

```
Definition value_eqb_fields rec (t: tag) :
  value_fields_t t → value_fields_t t
:= ...
  match t with
  ...
  | 3 (* word *) ⇒ fun '(Box_Vword s1 w1) ;(Box_Vword s2 w2) ⇒
     wsize_eqb s1 s2 && word_eqb s1 s2 w1 w2
  ...
```

The alternative would have been to replace the boolean conjunction with a pattern matching over a correctness proof and then cast the type of `w2` as follows:

```
Lemma adaptor f x y :
  (∀x y, f x y ↔ x = y) → { x = y } + { x <> y }.

Definition value_eqb_fields (t: tag) :=
  match t with
  ...
  | 3 (* word *) ⇒ fun '(Box_Vword s1 w1) '(Box_Vword s2 w2) ⇒
    match adaptor wsize_eqb s1 s2 wsize_eqb_OK with
    | left H  ⇒ word_eqb s1 w1 (case H with eq_refl ⇒ w2 end)
    | right _ ⇒ false
    end
  ...
```

This way of writing the equality test is not only more complex, but also puts the *proof* `wsize_eqb_OK` in the path of Coq's evaluation mechanisms: `value_eqb` would only reduce if the proof `wsize_eqb_OK` was kept transparent.

With our definition the proof `wsize_eqb_OK` is only used in the proof of `value_eqb_OK`, not in the definition of the equality test itself.

During the proof generation of `value_eqb_correct`, the dependency between arguments adds some complications to the proof synthesis. It is necessary to substitute all occurrences of `s2` by `s1`, and then invoke `word_eqb_correct`, which requires the size of the two words to match. To do this it is

[8] `word_eqb_OK` is the composition of `word_eqb_correct` and `word_eqb_refl`.

necessary to perform the bulk rewriting in steps, one level of dependency at a time. To illustrate this, we look at the proof of value_eqb_correct. The case corresponding to the Vword constructor looks like this:

```
eq_ind_r_n [::wsize]
  (fun s ⇒ ∀ w:word s, word_eqb s1 w1 s w →
     Some (Vword s1 w1) = Some (Vword s w))
  s1 s2 (wsize_eqb_correct s1 s2 hs2)
  (fun (w : word s1) (hw : word_eqb s1 w1 s1 w) ⇒
    eq_ind_r_n [::word s1]
      (fun w:word s1 ⇒ Some (Vword s1 w1) = Some (Vword s1 w))
      w1 w (word_eqb_correct w1 w hw))
  w2 hw2
```

in the context where hs2 has type wsize_eqb s1 s2 and hw2 has type word_eqb w1 w2. During the rewriting of s1 = s2, w2 and hw2 need to be generalized, in order to substitute s2 in their types. The parts of the terms involved in this generalization are underlined. The proof concludes with one extra rewriting step. The number of rewriting steps one needs to do is given by the longest chain of dependencies between the fields (typically two), and not by the number of fields.

6 Containers and deep induction principles

We introduce the deep data type (a simplified version of the instr one shown in the introduction) in order to explain the problem we need to overcome and our approach. This type occurs recursively under two containers, namely list and option.

```
Inductive deep := D : list (option deep) → deep.
```

The induction principle generated by Coq for deep is the following one:

```
deep_ind : ∀P: deep → Prop,
  (∀l: list (option deep), P (D l)) →
  ∀d: deep, P d
```

This "induction" principle is too weak, since it clearly lacks any induction hypothesis on l, or better, the subterms of type deep which occur inside l. To overcome this problem, we base our work on the *deep* and modular induction principles synthesized by [16], which we adapt (and simplify in Section 6.1) to better cover our case of interest.

The deep induction principle for deep has the following type, the differences are underlined:

```
deep_induction : ∀P : deep → Prop,
  (∀l, is_list (is_option P) l → P (D l)) →
  ∀d : deep, is_deep d → P d
```

In order to understand the difference, we need to look at the unary parametricity translations of the types involved:

```
Inductive is_option {A} (is_A : A → Type) : option A → Type :=
| is_None : is_option is_A None
| is_Some : ∀a, is_A a → is_option is_A (Some a).

Inductive is_list {A} (is_A : A → Type) : list A → Type :=
| is_nil : is_list is_A nil
| is_cons : ∀a, is_A a → ∀l, is_list is_A l →
              is_list is_A (cons a l).

Inductive is_deep : deep → Type :=
| is_D : ∀l, is_list (is_option is_deep) l → is_deep (D l).
```

The predicate is_option P intuitively means that P holds on the contents of Some; similarly is_list Q means that Q holds on all the elements of a list. Hence is_list (is_option P) l intuitively states that P holds on all the subterms of type deep of l.

The induction principle deep_induction is stronger but also has an extra cost: in order to use it one cannot just provide a term d or type deep, but one also has to provide a "proof" of is_deep d. This proof is automatically synthesized by [16], so in the end there is no extra cost on the user.

This extra argument to the induction principle also plays an important role in the composition of the lemmas which are proved by induction. In our case, if one proves the correctness lemmas for the containers list and option without passing this very last argument, one obtains "auxiliary" lemmas of the following types:

```
list_eqb_correct_aux   : ∀{A} (eqA: A → A → bool) l,
  is_list (eqb_correct_on eqA) l → eqb_correct_on (list_eqb eqA) l

option_eqb_correct_aux : ∀{A} (eqA: A → A → bool) o,
  is_option (eqb_correct_on eqA) o → eqb_correct_on (option_eqb eqA) o

list_eqb   : ∀{A} (eqA: A → A → bool), list A → list A → bool

option_eqb : ∀{A} (eqA: A → A → bool), option A → option A → bool
```

The intuitive reading is that these lemmas let one move the property eqb_correct_on across the container by applying the equality test for the container. For example if the test eqA is correct on all the elements of type A in the list l, then list_eqb eqA is correct on the list l.

The shape of the induction hypothesis in deep_induction is similar to the premise of these lemmas, but not identical. If we set P to eqb_correct_on deep_eqb, then we have the following induction hypothesis:

```
  is_list (is_option (eqb_correct_on deep_eqb)) l
```

The predicate eqb_correct_on is deeper, under is_option, than what the lemma list_eqb_correct_aux expects.

6.1 Functoriality

To the rescue comes the functoriality property of the unary parametricity translation of containers:

```
is_list_functor : ∀{A} (P Q : A → Type),
    (∀x : A, P x → Q x) →
    ∀l, is_list P l → is_list Q l
```

By passing option_eqb_correct_aux as the underlined assumption one obtains a term of type:

```
is_list_functor
  (is_option (eqb_correct_on deep_eqb))
  (eqb_correct_on (option_eqb deep_eqb))
  (option_eqb_correct_aux deep_eqb)
:
  ∀l, is_list (is_option (eqb_correct_on deep_eqb)) l →
        is_list (eqb_correct_on (option_eqb deep_eqb)) l
```

This term can transform the induction hypothesis into the premise of list_eqb_correct_aux.

The feqb tool synthesizes the following term before invoking the synthesis described in the previous sections:

```
fun l (IH : is_list (is_option (eqb_correct_on deep_eqb)) l) ⇒
  let H : eqb_correct_on (list_eqb (option_eqb deep_eqb)) l :=
    list_eqb_correct_aux (option_eqb deep_eqb) l
    (is_list_functor
      (is_option (eqb_correct_on deep_eqb))
      (eqb_correct_on (option_eqb deep_eqb))
      (option_eqb_correct_aux deep_eqb)
      l IH)
  in
   ... (* filled by Sections 4 and 5 *)
```

In practice the deep induction hypotheses provided by [16] are made shallow by using the auxiliary correctness lemmas for the containers. When more than one container needs to be crossed one needs to use functoriality lemmas.

Our contribution to this schema is to make functoriality lemmas about containers featuring a vparam simpler to use. For example the `tuple` type features both a tparam and a vparam.

```
Inductive tuple A i := { l : list A; p : length l == i }.

is_tuple_functor :
  ∀A (P Q : A → Type), (∀x, P x → Q x) →
  ∀(i : nat) (Pi : is_nat i),
  ∀(x : tuple a i), is_tuple A P i Pi x → is_tuple A Q i Pi x
```

The argument `Pi` is required in order to feed `is_tuple`, whose shape is dictated by the parametricity translation, but otherwise serves no purpose, unlike `P`, which is equipped with a map to `Q`. Since `i` is classified as a vparam we synthesize a functoriality lemma which leaves `Pi` untouched, and we require no map for it.

The approach previously used in [16] was to map `Pi` to a `Qi`, but also synthesize a proof that `is_nat i` has a unique inhabitant for each `i`, and use this result in subsequent derivations to identify `Pi` with `Qi` when needed. Since all our derivations work on the same AST which distinguishes type parameters, it is rather natural to use this simpler form of functoriality laws.

6.2 Tiding the knot

Auxiliary lemmas such as `list_eqb_correct_aux` are turned into proper lemmas for the final user by providing their last argument, namely a proof that `is_list (eqb_correct_on eqA) l`. In order to provide this argument, we first synthesize this general lemma, which shows `is_list P` for any predicate `P` which is true on its entire domain.

```
Fixpoint list_is_list {A} P (H : ∀x, P x) l {struct l} : is_list P l :=
  match l with
  | nil ⇒ is_nil P
  | cons x xs ⇒ is_cons P x (H x) xs (list_is_list P H xs)
  end.
```

Then we can prove the correctness lemma as follows:

```
Lemma list_eqb_correct A eqA (HeqA : ∀x, eqb_correct_on eqA x)
:  ∀l, eqb_correct_on (list_eqb eqA) l.
Proof.
  move⇒ l; exact: list_eqb_correct_aux eqA l
                    (list_is_list (eqb_correct_on eqA) HeqA l).
Qed.
```

That is, if `eqA` is a correct equality test on `A`, then `list_eqb eqA` is a correct equality test on `list A`.

7 Implementation and benchmarks

The implementation consists of around 800 lines of Elpi code, organized as follows:

file	size	generates
eqType	151	HOAS (and validation)
tag	55	tag
fields	264	fields_t, fields, construct, constructP
eqb	208	eqb_fields, eqb
eqbcorrect	274	eqb_correct, eqb_reflexive
eqbP	51	eqb_OK
total	803	feqb

Moreover we use the existing code for the generation of deep induction principles (including unary parametricity and functoriality lemmas) which amounts to about 800 lines of Elpi code.

It is out of scope to describe all the code of feqb in the present paper, nevertheless we give a taste of it presenting the rules for synthesizing `eqb_fields` in Figure 5.

The code is divided into two procedures: $\Gamma,\ \sigma \vdash \begin{smallmatrix}\mathtt{i} \mid I\\ \mathtt{j} \mid J\end{smallmatrix} \rightsquigarrow_\lambda \mathtt{B}$ and $\Gamma,\ \sigma \vdash \begin{smallmatrix}\mathtt{X}_s \mid A\\ \mathtt{Y}_s \mid B\end{smallmatrix} \rightsquigarrow_\wedge \mathtt{C}$. The former generate abstractions corresponding to the parameters of the inductive type, while the latter builds the code corresponding to the conjunction of equality tests that compares the fields of constructors.

Γ is a context which contains rules linking two types `t` and `s` to their equality test `f` (of type `t -> s -> bool`) : $\mathtt{t} \overset{?}{=} \mathtt{s} \mapsto \mathtt{f}$. We expect it to contain rules for known `eqTypes`, that is type for which feqb has previously synthesized the equality test, for example:

$$\Gamma \supset \left\{ \begin{array}{l} \mathtt{nat} \overset{?}{=} \mathtt{nat} \mapsto \mathtt{nat\_eqb} \\ \forall n\ m,\ \mathtt{word}\ \ n \overset{?}{=} \mathtt{word}\ \ m \mapsto \mathtt{word\_eqb}\ \ n\ \ m \\ \forall a\ b\ f,\ a \overset{?}{=} b \mapsto f \quad \Longrightarrow \\ \qquad\qquad\qquad \mathtt{list}\ a \overset{?}{=} \mathtt{list}\ b \mapsto \mathtt{list\_eqb}\ \ a\ \ b\ \ f \end{array} \right\}$$

Remember that we build heterogeneous equality tests when the types have value parameters, such as `word`. In the second rule one really needs the two words, their sizes `n` and `m` actually, in order to write their equality test `word_eqb n m`. The third rule has premises: since the type `list` is a container one needs a test for the contained in order to write the test for the container itself.

The predicate[9] $\Gamma \vdash a \overset{?}{=} b \mapsto \mathtt{f}$ allows to build the composition of the rules in Γ. For example, given the Γ above:

$$\Gamma \vdash \mathtt{list\ nat} \overset{?}{=} \mathtt{list\ nat} \mapsto \mathtt{list\_eqb\ nat\ nat\ nat\_eqb}$$

When a type expression in $\mathbb{E}$ needs to be used in a Coq term we invoke the conversion procedure $\lceil . \rceil_\sigma$. This code is straightforward, since $\mathbb{E}$ is a fragment of Coq's syntax. The only caveat is how to map variables bound in the HOAS to Coq terms. We carry σ which is an association between variables crossed in the input $\mathbb{I}$ and Coq terms. For example $\left\lceil \begin{smallmatrix} x \mapsto \mathtt{a} \\ y \mapsto \mathtt{b} \end{smallmatrix} \right\rceil$ extends the map linking x and y (in $\mathbb{E}$) with `a` and `b` respectively.

The rules for $\Gamma,\ \sigma \vdash \begin{smallmatrix}\mathtt{i} \mid I\\ \mathtt{j} \mid J\end{smallmatrix} \rightsquigarrow_\lambda \mathtt{B}$ recursively traverse two copies of the same inductive type declaration $\mathbb{I}$, namely I

[9]Not explained in the figure

$$\frac{\Gamma \cup \{\texttt{a} \overset{?}{=} \texttt{b} \mapsto \texttt{f}\},\ \sigma \circ \begin{bmatrix} x\mapsto\texttt{a} \\ y\mapsto\texttt{b}\end{bmatrix} \vdash \begin{array}{l}\texttt{i}\ \texttt{a} \mid X \\ \texttt{j}\ \texttt{b} \mid Y\end{array} \rightsquigarrow_{\lambda} \texttt{B}}{\Gamma,\ \sigma \vdash \begin{array}{l}\texttt{i} \mid \mathsf{tparam}(\lambda x.X) \\ \texttt{j} \mid \mathsf{tparam}(\lambda y.Y)\end{array} \rightsquigarrow_{\lambda} \texttt{fun (a b : Type) (f : a} \rightarrow \texttt{b} \rightarrow \texttt{bool)} \Rightarrow \texttt{B}}\ (tp)$$

$$\frac{\Gamma,\ \sigma \circ \begin{bmatrix} x\mapsto\texttt{a} \\ y\mapsto\texttt{b}\end{bmatrix} \vdash \begin{array}{l}\texttt{i}\ \texttt{a} \mid X \\ \texttt{j}\ \texttt{b} \mid Y\end{array} \rightsquigarrow_{\lambda} \texttt{B}}{\Gamma,\ \sigma \vdash \begin{array}{l}\texttt{i} \mid \mathsf{vparam}\ T_x\ (\lambda x.X) \\ \texttt{j} \mid \mathsf{vparam}\ T_y\ (\lambda y.Y)\end{array} \rightsquigarrow_{\lambda} \texttt{fun (a : } \lceil T_x \rceil_{\sigma}\texttt{) (b : } \lceil T_y \rceil_{\sigma}\texttt{)} \Rightarrow \texttt{B}}\ (vp)$$

$$\frac{\texttt{B}_w = \texttt{fun '(Box}_w\ \texttt{k}_w\texttt{) '(Box}_w\ \texttt{l}_w\texttt{)} \Rightarrow \texttt{C}_w \quad \Gamma \cup \{\texttt{i} \overset{?}{=} \texttt{j} \mapsto \texttt{rec}\},\ \sigma \circ \begin{bmatrix} x\mapsto\texttt{i} \\ y\mapsto\texttt{j}\end{bmatrix} \vdash \begin{array}{l}\texttt{k}_w \mid K_w \\ \texttt{l}_w \mid L_w\end{array} \rightsquigarrow_{\wedge} \texttt{C}_w \quad \forall w \in \{1 \dots n\}}{\Gamma,\ \sigma \vdash \begin{array}{l}\texttt{i} \mid \mathsf{inductive}\ (\lambda x.[K_1;\dots;K_n]) \\ \texttt{j} \mid \mathsf{inductive}\ (\lambda y.[L_1;\dots;L_n])\end{array} \rightsquigarrow_{\lambda} \begin{array}{l}\texttt{fun (rec : i} \rightarrow \texttt{j} \rightarrow \texttt{bool) (t : positive)} \Rightarrow \\ \texttt{match t with 1} \Rightarrow \texttt{B}_1 \mid \dots \mid \texttt{n} \Rightarrow \texttt{B}_n \mid \texttt{\_} \Rightarrow \texttt{false end}\end{array}}\ (ind)$$

$$\frac{\Gamma \vdash \lceil T_x \rceil_{\sigma} \overset{?}{=} \lceil T_y \rceil_{\sigma} \mapsto \texttt{F} \quad \Gamma,\ \sigma \vdash \begin{array}{l}\texttt{X}_s \mid A \\ \texttt{Y}_s \mid B\end{array} \rightsquigarrow_{\wedge} \texttt{C}}{\Gamma,\ \sigma \vdash \begin{array}{l}\texttt{X :: X}_s \mid \mathsf{regular}\ T_x\ A \\ \texttt{Y :: Y}_s \mid \mathsf{regular}\ T_y\ B\end{array} \rightsquigarrow_{\wedge} \texttt{F X Y \&\& C}} \qquad \frac{\Gamma \vdash \lceil T_x \rceil_{\sigma} \overset{?}{=} \lceil T_y \rceil_{\sigma} \mapsto \texttt{F} \quad \Gamma,\ \sigma \circ \begin{bmatrix} x\mapsto\texttt{X} \\ y\mapsto\texttt{Y}\end{bmatrix} \vdash \begin{array}{l}\texttt{X}_s \mid A \\ \texttt{Y}_s \mid B\end{array} \rightsquigarrow_{\wedge} \texttt{C}}{\Gamma,\ \sigma \vdash \begin{array}{l}\texttt{X :: X}_s \mid \mathsf{dependent}\ T_x\ (\lambda x.A) \\ \texttt{Y :: Y}_s \mid \mathsf{dependent}\ T_y\ (\lambda y.B)\end{array} \rightsquigarrow_{\wedge} \texttt{F X Y \&\& C}}$$

$$\frac{\Gamma,\ \sigma \vdash \begin{array}{l}\texttt{X}_s \mid A \\ \texttt{Y}_s \mid B\end{array} \rightsquigarrow_{\wedge} \texttt{C}}{\Gamma,\ \sigma \vdash \begin{array}{l}\texttt{X :: X}_s \mid \mathsf{irrelevant}\ T_x\ A \\ \texttt{Y :: Y}_s \mid \mathsf{irrelevant}\ T_y\ B\end{array} \rightsquigarrow_{\wedge} \texttt{C}}\ (irr) \qquad \frac{}{\Gamma,\ \sigma \vdash \begin{array}{l}\epsilon \mid \mathsf{stop} \\ \epsilon \mid \mathsf{stop}\end{array} \rightsquigarrow_{\wedge} \texttt{true}}$$

Figure 5. Elpi code for the synthesis of `eqb_fields`

and J. This is necessary since it has to generate heterogeneous tests. During this recursion it carries in `i` and `j` the name of the inductive type applied to the variables which were previously abstracted. Remark how rule (tp) abstracts a function `f` to compare the type parameter, and adds to Γ a rule $\texttt{a} \overset{?}{=} \texttt{b} \mapsto \texttt{f}$ to later know how to compare fields of types, respectively, `a` and `b`.

Once all parameters are abstracted, rule (ind) abstracts one last equality test for the inductive type itself (namely `rec`), and dispatches on the positive `t` all the branches B_w, one per constructor. All branches abstract and destruct the box corresponding to the constructor, and pass all its fields k_w and l_w, together with the corresponding K_w and L_w, to the other function $\rightsquigarrow_{\wedge}$.

The first two rules of $\rightsquigarrow_{\wedge}$ are identical: they compare the field variables `X` and `Y` by looking up a function `F` in Γ using their types. If these arguments have the type of the inductive being currently processed, then the lookup gives `rec`. If their type is one of the type parameters of the inductive, then the test abstracted by rules (tp) is used. If their type is an already known type, possibly a container, we expect Γ to contain a rule for it.

Rule (irr) simply ignores the irrelevant argument, indeed the equality tests we synthesize discard this kind of fields.

Inference rules like the ones in Figure 5 write quite naturally in a logic programming language such as Elpi, which also allows for a trivial implementation of $\Gamma \vdash a \overset{?}{=} b \mapsto \texttt{f}$.

7.1 Benchmarks

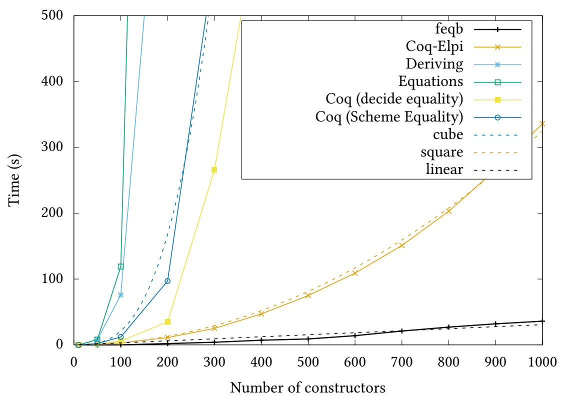
To get an idea of the performance of feqb *in practice*, we performed a few benchmarks comparing feqb to the tools mentioned in Section 1.1.

All the experiments were performed on a HP EliteBook 840 G3 with an Intel Core i7-6600U processor at 2.60 GHz and 16 GB of memory. We used the following versions of the tools: OCaml 4.14.0, Coq 8.16, Coq-Elpi 1.16, Deriving 0.1.0 and Equations 1.3+8.16. Deriving tries by default to simplify the terms it produces, but following the advice in the README, we disabled this costly pass.

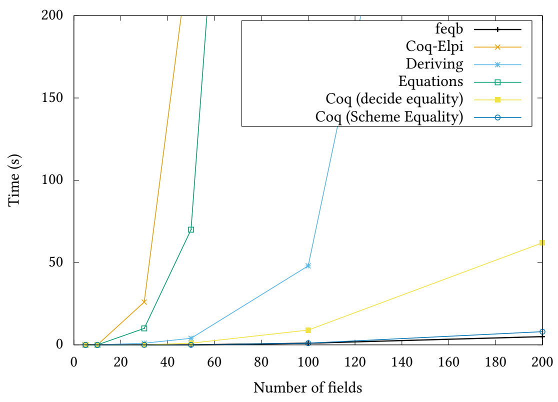
We tested the tools on two families of types. The first one is *variants*, i.e. inductive types where all constructors have no arguments. The second family is *records*, i.e. inductive types with just one constructor but a potentially large number of arguments. The two tests measure how each dimension in the size of the inductive impacts the synthesis time. For each tool, we measured the time taken by the entire generation procedure seen as a black box (which includes synthesis, typechecking, etc. . .).

The results on the first family are shown in Figure 6a. We tested the tools on a few variant types (number of constructors from 10 to 1000), and measured their execution time, with a time out of one hour. The results are shown using solid lines. There are three more curves in the graph, using dashed lines: we tried to make some mathematical functions fit our curves, based on what we anticipated as complexity of the tools (cf. Figure 1). Four tools run out of time when the number of constructors is more than a few hundreds: Equations, Deriving, `Scheme Equality` and `decide equality`. Experimentally they look cubic in the number of constructors. Coq-Elpi scales better and is plausibly quadratic in the number of constructors. Our tool, feqb, outclasses the other approaches and appears to be roughly linear.

The results on the second family are shown in Figure 6b. We tested the tools on a few records (number of fields from 5 to 200). Three tools timeout before reaching 100 fields: Coq-Elpi, Equations and Deriving. `decide equality` behaves much better, but is still outperformed by feqb and `Scheme Equality` whose performances are really similar.



(a) On flat inductive types



(b) On records

Figure 6. Benchmark results

Admittedly, the sizes that we tested in these two benchmarks were big compared to what the daily Coq user needs. But they are not completely unrealistic either, and we wanted to get an idea of the asymptotic complexity.

To be honest, our initial measurements were not reflecting the asymptotic complexity we hoped for feqb. It is only after eliminating unrelated slowdowns from Coq-Elpi (on which feqb is based) that we could sample the real complexity. And we think that there is still a lot of room for improvement in both the code of Coq-Elpi and feqb. This engineering work is an additional contribution of the current paper and also explains why Coq-Elpi's `derive` is, in spite of being quadratic, faster than the alternatives. We believe some engineering work could make the other tools show their actual complexity bound. For example Deriving is a reflexive procedure, it does not really generate an equality test, but it rather provides a program that given a first class description of an inductive data type generates the test, together with its proof. It is totally unclear to us why it does not scale to large inductives.

Last, we want to mention that Elpi is an interpreted language, and in spite of being reasonably efficient [8], it cannot outperform OCaml code. Still, thanks to the better asymptotic complexity, our synthesis is faster than all other tools starting from 50 constructors or 50 fields, even if the other tools are written in OCaml!

Until now, we discussed the performance of the generation of the boolean tests, but not the performance of the boolean tests themselves. We tried to run the tests on objects of type `tree` with 2^{20} nodes to check how they perform on big objects. The fastest approaches are `Scheme Equality` and Coq-Elpi (0.16 s). When we call the simplifying procedure of Deriving, the resulting test is nearly as fast as these two (0.26 s); without the simplifying pass, it becomes much slower (26 s). `decide equality` and feqb are a bit slower (1.2 s), certainly paying for mixing terms and proofs for the first, and for introducing too many indirections for the second. As for the test produced by Equations, it runs out of memory, also paying for mixing terms and proofs, but this time heavily. We consider that the relative slowness of the test synthesized by feqb is not a problem in practice.

The other main goal when designing feqb was to support a certain number of features of Coq inductive types. Figure 7 illustrates on a few concrete types the features supported by feqb and the other tools. On top of types already introduced in this paper, we introduce two new types: `forest` encodes forests (from graph theory) as a mutual inductive type, and `vector` is taken from the standard library of Coq as an example of indexed data type.

```
Inductive forest := | empty | add : tree -> forest -> forest
with tree := | node : nat -> forest -> tree.

Inductive vector A : nat -> Type :=
| nil : vector A 0
| cons : A -> forall n, vector A n -> vector A (S n).
```

While feqb behaves very well, it does not support these two types. It does not support indexed data types such as `vect` like every other tool except Equations. It also inherits from Coq-Elpi the limitation of not covering mutual inductive types such as `forest`, although we believe the schema we presented can be easily extended to that case once mutual inductive types are available in Coq-Elpi.

8 Future work

For future work we look at other places where a naive equality test brings terrible effects on the computation complexity. The first one that comes to mind is the discriminate tactic which internally generates a proof term similar to the one of an equality test proof: when a data type occurs in the index of an inductive relation its equality test is key to discharge impossible branches when reasoning by inversion. We may be able to apply our technique there and reduce the space and time complexity of the inversion and discriminate tactics.

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

Figure 7. Tools coverage of inductive types

References

[1] Andreas Abel. 2007. *Type-based termination: a polymorphic lambda-calculus with sized higher-order types.* Ph.D. Dissertation. Ludwig Maximilians University Munich. https://d-nb.info/984765581

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. https://doi.org/10.1145/3133956.3134078

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020.* IEEE, 965–982. https://doi.org/10.1109/SP40000.2020.00028

[4] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1607–1622. https://doi.org/10.1145/3319535.3363211

[5] Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. 2006. CIC[ˆ( )]: 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 (Lecture Notes in Computer Science)*, Miki Hermann and Andrei Voronkov (Eds.), Vol. 4246. Springer, 257–271. https://doi.org/10.1007/11916277_18

[6] Jonathan Chan and William J. Bowman. 2019. Practical Sized Typing for Coq. *CoRR* abs/1912.05601 (2019). arXiv:1912.05601 http://arxiv.org/abs/1912.05601

[7] Arthur Azevedo de Amorim. 2021. *Deriving.* https://doi.org/10.5281/zenodo.7065501

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

[9] Benjamin Grégoire and Jorge Luis Sacchini. 2010. On Strong Normalization of the Calculus of Constructions with Type-Based Termination. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings (Lecture Notes in Computer Science)*, Christian G. Fermüller and Andrei Voronkov (Eds.), Vol. 6397. Springer, 333–347. https://doi.org/10.1007/978-3-642-16242-8_24

[10] Jason Gross. [n.d.]. Scheme Equality is slow and generates enormous proof terms. https://github.com/coq/coq/issues/8517

[11] Jason Gross and Andres Erbsen. 2022. 10 Years of Superlinear Slowness in Coq. https://jasongross.github.io/papers/2022-superlinear-slowness-coq-workshop.pdf Presented at The Coq Workshop 2022.

[12] Jason S. Gross. 2021. *Performance Engineering of Proof-Based Software Systems at Scale.* PhD Thesis. Massachusetts Institute of Technology. https://jasongross.github.io/papers/2021-JGross-PhD-EECS-Feb2021.pdf

[13] Patricia Johann and Andrew T. Polonsky. 2020. Deep Induction: Induction Rules for (Truly) Nested Types. *Foundations of Software Science and Computation Structures* 12077 (2020), 339 – 358.

[14] Conor McBride. 1999. *Dependently Typed Functional Programs and their Proofs.* Ph.D. Dissertation. University of Edinburgh. Available from http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/.

[15] Matthieu Sozeau and Cyprien Mangin. 2019. Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq. *Proc. ACM Program. Lang.* 3, ICFP, Article 86 (jul 2019), 29 pages. https://doi.org/10.1145/3341690

[16] Enrico Tassi. 2019. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 29:1–29:18. https://doi.org/10.4230/LIPIcs.ITP.2019.29

[17] The Coq Development Team. 2022. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.5846982