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

Benjamin Grégoire, Jean-Christophe Léchenet, Enrico Tassi

To cite this version:
Benjamin Grégoire, Jean-Christophe Léchenet, Enrico Tassi. Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin’s data types with Coq-Elpi. CPP ’23: 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, Jan 2023, Boston MA USA, France. pp.167-181, 10.1145/3573105.3575683. hal-03800154

HAL Id: hal-03800154
https://inria.hal.science/hal-03800154

Submitted on 6 Oct 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License
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.

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. 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.

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 instructions. 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 \( \text{ord} \):

\[
\text{Inductive} \ \text{ord} \ (m \ : \ \text{nat}) := \text{mkord} : \forall n, \ n < m \rightarrow \text{ord} m.
\]

In order to constrain the inhabitants of that type, type theory lets one pair terms with proofs. An inhabitant of \( \text{ord} m \) would be a number \( n \) paired with a proof of \( n < m \). The type of the pairing constructor \( \text{mkord} \) 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.

\[
\text{Definition} \ \text{wsize} := \text{U8} \mid \text{U16} \mid \text{U32} \mid \text{U64} \mid \text{U128} \mid \text{U256}.
\]

\[
\text{Definition} \ \text{wsize}\_\_\text{s} := \text{nat} := \ldots \text{(* omitted *)}
\]

\[
\text{Record} \ \text{word} (\text{mbits} : \text{wsize}) := \text{mkWord} (\ w : \text{Z}; \ n : 0 <=? w && w <? 2^{(\text{wsize}\_\_\text{s} \text{nbits})} \).
\]

\[
\text{Variant} \ \text{value} : \text{Type} :=
\]

\[
\text{Variant} \ \text{instruction} : \text{Type} :=
\]

\[
\text{Definition} \ \text{chk} := \text{forall} \ PI \ \text{Pfi} \ P,
\]

\[
\text{(* constructor of pair *)}
\]

\[
\text{(* conclusion *)}
\]

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 \( \text{instr} \) 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 \( \text{cons} \) 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 \text{Derive} command; Deriving [7] is a tool able to synthesize an \text{eqType} via a reflexive procedure; The Coq system [17] provides the \text{decide equality} tactic and the Scheme Equality \text{command}; Coq-Elpi is an extension language for Coq that comes bundled with a \text{derive} [16] \text{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 \text{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

\[\text{...}(\text{** conclusion **})\]

\[\rightarrow \forall c: \text{cmd}, P c\]

\[\text{To see why this matters read http://gallium.inria.fr/blog/coq-eval/} \]
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 \((x = y) \cup \{x \neq y\}\); 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 equality test, 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 material³.

### 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.

```coq
Fixpoint eqb (x y : nat) :=
  match x with
  | 0 ⇒ match y with
    | 0 ⇒ true
    | S _ ⇒ false
  end
  | S n ⇒ match y with
    | 0 ⇒ 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.

```coq
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.

```coq
Lemma eqb_refl x : eqb x x :=
  fun x : nat ⇒
  nat_ind (fun n : nat ⇒ eqb n n)
  eq_refl (= eqb 0 0 ⇒
    (fun p (IH : eqb p p) ⇒
      IH) (= eqb (S p) (S p) ⇒
      x).
end.
```

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

```coq
Lemma eqb_correct : forall x y, eqb x y → x = y :=
  fun x : nat ⇒
  nat_ind (fun n : nat ⇒ forall y : nat, eqb n n → n = y)
  (fun y : nat ⇒
  nat_ind (fun n : nat ⇒
    (fun p (IH : eqb p p) ⇒
      IH) (= eqb (S p) (S p) ⇒
      x).
end.
```

³Its ongoing integration in Coq-Elpi can be found at https://github.com/LPCIC/coq-elpi/pull/319
The second function, \( \text{nat_tag n :=...} \)

Definition to compare the fields of a given tag, we can write the generic construction. Naturally the type of the function is dependent, to each tag the type of its fields. For the type this dependent type we define a constructor. Since each constructor may have different fields. To write constructors by giving them a first class representation, and only

The key idea to avoid the quadratic blowup is to keep the first step lets us not generate the nested \text{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 \( \lambda \).

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

Definition \text{nat_tag n :=...} \end

The second function, \text{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 \text{fields_t} function associating to each tag the type of its fields. For the type \text{nat}, one obtains:

Definition \text{nat_fields t p :=...} \end

Given these two functions and a function named \text{eqb_fields} to compare the fields of a given tag, we can write the generic definition \text{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 \( f_1 \) and \( f_2 \). Remark that we need a type cast on \( f_2 \) to convince Coq that its type is \text{fields_t t1}.

The generic code cannot perform the recursion itself, since Coq does not know yet that \( \lambda \) is inductively defined. It is the code specific to the inductive type, \text{nat} here, which will perform the recursion.

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

Definition \text{nat_eqb_fields rec t:=...} \end

Remark that we have been able to share all the branches returning \text{false} into one, the \text{right} branch of \text{eqb_body} (Figure 2). Furthermore all the other branches are defined using \text{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 \text{nat_eqb (n1 n2 : nat) [struct n1]} := \text{eqb_body p 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 \text{tag} and \text{eqb_body} in the definition of the fixpoint itself. To do so, we modify a little the definition of \text{eqb_body}: instead of taking two arguments of type \( \lambda \), the first argument, namely \( x \), is given by its (pre computed) tag and fields.

Definition \text{eqb_body t1 (f1 : field_t t1) (y : A)} :=

match \text{Pos.eq_dec t2 t1 with...} \end

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

Fixpoint \text{nat_eqb (n1 n2 : nat) [struct n1]} := \text{eqb_body n1 n2 n1 n2}.

The solution to fix the complexity of \text{eqb} could be to optimize the internal representation of Coq terms so that the match node can share common branches (the \text{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 \text{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 \text{not generate} the nested \text{match} nodes.

We start by defining some auxiliary functions whose signature is on Figure 2, which provides the generic interface to build our linear test on a type \( \lambda \).

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

Definition \text{nat_tag n :=...} \end

The second function, \text{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 \text{fields_t} function associating to each tag the type of its fields. For the type \text{nat}, one obtains:

Definition \text{nat_fields t p :=...} \end

Given these two functions and a function named \text{eqb_fields} to compare the fields of a given tag, we can write the generic definition \text{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 \( f_1 \) and \( f_2 \). Remark that we need a type cast on \( f_2 \) to convince Coq that its type is \text{fields_t t1}.

The generic code cannot perform the recursion itself, since Coq does not know yet that \( \lambda \) is inductively defined. It is the code specific to the inductive type, \text{nat} here, which will perform the recursion.

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

Definition \text{nat_eqb_fields rec t:=...} \end

Remark that we have been able to share all the branches returning \text{false} into one, the \text{right} branch of \text{eqb_body} (Figure 2). Furthermore all the other branches are defined using \text{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 \text{nat_eqb (n1 n2 : nat) [struct n1]} := \text{eqb_body n1 n2 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 \text{tag} and \text{eqb_body} in the definition of the fixpoint itself. To do so, we modify a little the definition of \text{eqb_body}: instead of taking two arguments of type \( \lambda \), the first argument, namely \( x \), is given by its (pre computed) tag and fields.

Definition \text{eqb_body t1 (f1 : field_t t1) (y : A)} :=

match \text{Pos.eq_dec t2 t1 with...} \end

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

Fixpoint \text{nat_eqb (n1 n2 : nat) [struct n1]} := \text{eqb_body n1 n2 n1 n2}.

The solution to fix the complexity of \text{eqb} could be to optimize the internal representation of Coq terms so that the match node can share common branches (the \text{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).
2.3 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 $\text{construct}$ indicates that fields and construct cancel out.

Variable $\text{construct} : \forall t, \text{fields}_t t \rightarrow \text{option } A$.

Variable $\text{construct}_P : \forall a, \text{construct } (\text{fields } a) = \text{Some } a$.

The instantiation for $\text{nat}$ leads to the following definition and trivial proof:

Definition $\text{nat_construct } t : \text{nat_fields}_t t \rightarrow \text{option } \text{nat} :=$

match $t$ with
| 0 $\Rightarrow$ $\text{fun } _ \Rightarrow \text{Some } 0$
| $n \Rightarrow$ $\text{fun } p \Rightarrow \text{Some } (S p)$
| $\_ \Rightarrow$ $\text{fun } _ \Rightarrow \text{None}$.
end.

Lemma $\text{nat_construct}_P (a : \text{nat}) : \text{nat_construct } (\text{nat_fields } a) = \text{Some } a$.
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 $\text{sum } (L R : \text{Type}) : \text{Type} :=$

| $\text{inl } L \rightarrow \text{sum } L R | \text{inr } R \rightarrow \text{sum } L R$.

To build a witness using the constructor $\text{inl}$ (resp. $\text{inr}$), 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 $\text{eqb_body}$, we provide the following lemma.

Definition $\text{eqb_fields}_\text{correct}_P (a : A) :=$

$\forall f : \text{fields }_t (\text{tag } a)$,
$\text{eqb_fields } (\text{fields } a) f \rightarrow \text{Some } a = \text{construct } f$.

Lemma $\text{eqb_body}_\text{correct } a_1 :$
$\text{eqb_fields}_\text{correct}_P \text{ on } a_1 \rightarrow$
$\forall a_2$.

Section Core.

Variable $A : \text{Type}$.

Variable $\text{tag} : A \rightarrow \text{positive}$.

Variable $\text{fields}_t : \text{positive} \rightarrow \text{Type}$.

Variable $\text{eqb_fields} : \forall t, \text{fields}_t t \rightarrow \text{fields}_t t \rightarrow \text{bool}$.

Definition $\text{eqb_body} (x y : A) :=$

let $t_1 := \text{tag } x \text{ in}$
let $t_2 := \text{tag } y \text{ in}$
match $\text{Pos } \text{eq_dec } t_1 t_2 \text{ with }$
| $\text{left } h \rightarrow \text{let } f_1 : \text{fields}_t t_1 \rightarrow \text{fields}_t x \text{ in}$
| $\text{left } h \rightarrow \text{let } f_2 : \text{fields}_t t_2 \rightarrow \text{fields}_y \text{ in}$
| $\text{match } h \text{ with eq_refl } \Rightarrow \text{f2 end}$
| $\text{right } \Rightarrow \text{false}$
| $\end$. End Core.

Figure 2. Core interface for generic equality
of the constructors, which can hence mention it to declare recursive data.

A constructor declaration, represented by the node C, carries the "list" of the types of the arguments of the constructor:

1. regular stands for the usual, non dependent, argument like the A and list A arguments of cons;
2. 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.
3. irrelevant stands for an argument which can be ignored by the equality test.
4. stop stands for the empty list.

All arguments and vparam carry a 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 expressions (E) occurring in the type of the constructor are equipped with a decidable equality, rules (ir) and (a), or are irrelevant, rule (ir).

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 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, \( \Gamma \vdash I \in \mathbb{V}_I \), ensures the validity of object of type I. The context \( \Gamma \) 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, \( \Gamma \vdash K \in \mathbb{V}_C \), is for the well formedness of constructor declarations. It ensures that all type expressions (E) occurring in the type of the constructor are equipped with a decidable equality, rules (ir) and (a), or are irrelevant, rule (ir).

The third predicate, \( \Gamma \vdash E \in \mathbb{V}_E \), is for validity of 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 \( i \vdash I \) 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^5 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 list (A : Type) := nil | cons : A -> list A -> list A.
Record word (nbw: wsize) := mkWord {
  n : Z;
  w : : 0 <=? w && w <? 2^n(wsize nbw)}.

"list" \( \equiv \) tparam \( \lambda a. \)
  inductive \( \lambda l. \)
    stop
  \[\text{regular} \ a \ (\text{regular} \ l \ \text{stop})\]

"word" \( \equiv \) vparam (app "wsize" []) \( \lambda i. \)
  inductive \( \lambda t. \)
    dependent (app "Z" [])
  \( \lambda n. \text{irrelevant} (...) \ \text{stop} \)
```

\( ^4 \) 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.
When c

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:

\[
\text{eQB\_fields\_correct\_on}\ (\text{tree\_eQB\_fields\ tree\_eQB})\ c
\]

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

\[
\forall f2:\ \text{box\_for\_Node},
\text{tree\_eQB\_fields\ tree\_eQB}\ (\text{Box\_Node}\ n1\ l1\ r1)\ f2 ---
\text{Some}\ (\text{Node}\ n1\ l1\ r1) = \text{tree\_construct}\ f2
data:tree
\]

After destructing f2 into Box\_Node n2 l2 r2 we have:

\[
\text{nat\_eQB\ n2\ l2\ r2} ---
\text{Some}\ (\text{Node}\ n1\ l1\ r1) = \text{Some}\ (\text{Node}\ n2\ l2\ r2)
data:tree
\]

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

1. breaking the conjunction into separate hypotheses
2. turning each hypothesis\(^5\) into an equation, namely
   \[n1 = n2, l1 = l2\text{ and } r1 = r2\]
3. finally rewriting with these equations
   The resulting goal can be trivially proved by reflexivity:
   \[
   \text{Some}\ (\text{Node}\ n2\ l2\ r2) = \text{Some}\ (\text{Node}\ n2\ l2\ r2)
data:tree
   \]

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

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:

\[
\text{eQB\_fields\_correct\_on}\ (\text{tree\_eQB\_fields\ tree\_eQB})\ c
\]

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

\[
\forall f2:\ \text{box\_for\_Node},
\text{tree\_eQB\_fields\ tree\_eQB}\ (\text{Box\_Node}\ n1\ l1\ r1)\ f2 ---
\text{Some}\ (\text{Node}\ n1\ l1\ r1) = \text{tree\_construct}\ f2
data:tree
\]

After destructing f2 into Box\_Node n2 l2 r2 we have:

\[
\text{nat\_eQB\ n2\ l2\ r2} ---
\text{Some}\ (\text{Node}\ n1\ l1\ r1) = \text{Some}\ (\text{Node}\ n2\ l2\ r2)
data:tree
\]

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

1. breaking the conjunction into separate hypotheses
2. turning each hypothesis\(^5\) into an equation, namely
   \[n1 = n2, l1 = l2\text{ and } r1 = r2\]
3. finally rewriting with these equations
   The resulting goal can be trivially proved by reflexivity:
   \[
   \text{Some}\ (\text{Node}\ n2\ l2\ r2) = \text{Some}\ (\text{Node}\ n2\ l2\ r2)
data:tree
   \]

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 \(a\) a proof for \(a \rightarrow b \rightarrow c \rightarrow P\).

A natural proof term for \(a \land b \land c \rightarrow P\) is:

\[
\text{and\_E}\ (\text{and\_B}\ a\ b\ C)\ p
\]

where \(\text{and\_E}\), the standard “eliminator” of \(\land\), has type:

\[
\forall a\ b\ (P:\ \text{Prop})\ (a \land b \rightarrow P) \rightarrow a \land b \rightarrow P
\]

It is easy to check that the iterated application of \(\text{and\_E}\) leads to a quadratic proof term. It is certainly possible to solve the problem using \(\text{let}\) 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 \(\text{implies}\) as follows:

\[
\text{Fixpoint}\ \text{implies}\ (1:\ \text{list\ bool})\ (P:\ \text{Prop})\ :\ \text{Prop} :=
\text{match}\ l\ \text{with}
| [] \Rightarrow P
| (b :: l) \Rightarrow b \rightarrow \text{implies}\ l\ P
\text{end}
data:tree
\]

Given a list of boolean \([b0; \ldots; bn]\) and a predicate \(P\), the term \(\text{implies}\ [b0; \ldots; bn] P\) reduces to \(b0 \rightarrow \ldots \rightarrow bn \rightarrow P\).

\(^5\)The first equation comes from the correctness of \(\text{nat\_eQB}\), the two other by induction hypothesis.
Then we define the recursive function \text{allr} that computes the conjunction of its arguments:

\text{Fixpoint allr (l: list bool) :=}
\begin{align*}
\text{match l with} \\
\text{| [] :⇒ true} \\
\text{| [x; y] :⇒ x \land y \land \text{allr l}} \\
\text{end.}
\end{align*}

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

\text{Lemma eq_ind_r_n (eq n l r n2 l2 r2 hr hl) :} \text{eq_ind_r_n P (T: list} \rightarrow \text{Prop P \implies P} \rightarrow \text{eq_ind_r_n P}}.

If we have the following three equations

\begin{align*}
\text{hr} : r1 = r2 \\
\text{hl} : l1 = l2 \\
\text{hn : n1 n2} \\
\end{align*}

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

\text{Node n1 l1 r1 = Node n2 l2 r2}

we have to synthesize the following proof term:

\text{eq_ind_r_nP (T: list} \rightarrow \text{Prop P \implies P} \rightarrow \text{eq_ind_r_n P}}.

This proof term is quadratic in the number of arguments of \text{Node}, since each rewriting step “copies” the entire goal, which is linear in the number of arguments of \text{Node}.

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

\text{Fixpoint p_type (T: list Type) :=}
\begin{align*}
\text{match T with} \\
\text{| [] :⇒ Prop} \\
\text{| [A; T] :⇒ A \rightarrow p_type T} \\
\text{end.}
\end{align*}

\text{Fixpoint eq_ind_r_n (T: list Type) : p_type T \rightarrow p_type T \rightarrow Prop :=}
\begin{align*}
\text{match T with} \\
\text{| [] :⇒ fun p1 p2 \Rightarrow} \\
\text{| [A; T] :⇒ fun p1 p2 \Rightarrow} \\
\text{V (a1 a2; A), a1 = a2 \rightarrow eq_ind_r_n T (p1 a1) (p2 a2)} \\
\text{end.}
\end{align*}

\text{Lemma eq_ind_r_nP (T: list Type) : p_type T \rightarrow p_type T \rightarrow Prop :=}
\begin{align*}
\text{match T with} \\
\text{| [] :⇒ Prop} \\
\text{| [A; T] :⇒ Prop} \\
\text{end.}
\end{align*}

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

\text{p_type [:: nat; tree; tree] = nat \rightarrow tree \rightarrow tree \rightarrow Prop}

and in turn \text{eq_ind_r_n} reduces to this statement

\text{eq_ind_r_n [:: nat; tree; tree] =}
\begin{align*}
\text{fun p1 p2 : nat \rightarrow tree \rightarrow tree \rightarrow Prop \Rightarrow} \\
V (a1 a2 : nat), a1 = a2 \rightarrow \\
V (b1 b2 : tree), b1 = b2 \rightarrow \\
V (c1 c2 : tree), c1 = c2 \rightarrow \\
p1 a1 b1 c1 \rightarrow p2 a2 b2 c2
\end{align*}

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

\text{eq_ind_r_n [:: nat; tree; tree]}
\begin{align*}
\text{(fun n l r \Rightarrow Node n l l r = Node n l r)} \\
n l n2 \text{hn l1 l2 h1 r1 r2 hr refl_equality}
\end{align*}

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 \text{Node}.

The \text{eq_ind_r_nP} proof device is similar to the \text{nary_congruence} one which is part of the \text{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 \text{word} has the following shape:

\text{word_eqb : \forall s1 s2, word s1 \rightarrow word s2 \rightarrow bool}

As expected the specification constrains \text{s1} and \text{s2} to be the same:

\text{word_eqb_OK : \forall s (w1 w2 : word s), word_eqb s s w1 w2 \leftrightarrow w1 = w2}

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

\text{Definition value_eqb_fields rec (t: tag) : value_fields t t :=}
\begin{align*}
\text{match t with} \\
\text{| 3 (* word *) :⇒ fun 'Box_Vword s1 w1) \rightarrow wsize_eqb s1 s1 w1 w1) \rightarrow false) \\
\text{end.}
\end{align*}

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

\text{Lemma adaptor f x y : (\forall x y, f x y \leftrightarrow x = y) \rightarrow \forall x y, f x y) \\
\text{Definition value_eqb_fields (t: tag) :=}
\begin{align*}
\text{match t with} \\
\text{| 3 (* word *) :⇒ fun 'Box_Vword s1 w1) \rightarrow wsize_eqb s1 s2 w1 w2 \rightarrow false) \\
\text{end.}
\end{align*}

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

With our definition the proof \text{wsize_eqb_OK} is only used in the proof of \text{value_eqb_OK}, not in the definition of the equality test itself.

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

\text{word_eqb_OK} is the composition of \text{word_eqb_correct} and \text{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:

\[
\text{eq_ind_r_n :: wsize}\\
\begin{align*}
&\text{(fun s \Rightarrow w:word s, w word_eqb s1 w1 s w \rightarrow} \\
&\text{Some (w word_eqb s1 w1 w s))} \\
&\text{s1 s2 (w size_eqb_correct s1 s2 hw2) \\
&\text{(fun (w : word s1) \Rightarrow hw : word_eqb s1 w1 w1 s w) \Rightarrow} \\
&\text{eq_ind_r_n :: word_eqb w1 w2)} \\
&\text{in the context where hs2 has type wsize_eqb s1 s2 and hw2 has type word_eqb s1 w1 w2. During the rewriting of s1 = s2, s2 and hw2 need to be generalized, in order to substitute s2 in their types. The parts of the terms involved in this generalization are underlined. The proof concludes with one extra rewriting step. The number of rewriting steps one needs to do is given by the longest chain of dependencies between the fields (typically two), and not by the number of fields.}
\]

### 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:

\[
\text{deep_ind : \forall P : deep \rightarrow Prop,} \\
(\forall l, is_list P l \rightarrow \forall d : deep, P d)
\]

This "induction" principle is too weak, since it clearly lacks any induction hypothesis on 1, or better, the subterms of type `deep` which occur inside 1. 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:

\[
\text{deep_induction : \forall P : deep \rightarrow Prop,} \\
(\forall l, is_list (option deep) l \rightarrow \forall d : deep, is_deep d \rightarrow P d)
\]

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

**Inductive** is_option P := option A → Prop,

\[
\text{is_list (is_option P) l \rightarrow P (D l))} \\
\forall d : deep, P d
\]

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

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

\[
\text{fun l (IH : is_list (is_option (eqb_correct_on deep_eqb)) l :=} \\
\text{let H : eqb_correct_on (option_eqb deep_eqb)) l \Rightarrow} \\
\text{list_eqb_correct_aux (option_eqb deep_eqb) l
}\]

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)` intuitively states that `P` holds on all the subterms of type `deep` of 1.

The induction principle `deep_induction` is stronger but also has an extra cost: in order to use it one cannot just provide a term `a` or type `deep`, but one also has to provide a "proof" of `is_deep a`. 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:

\[
\text{list_eqb_correct_aux : \forall (A) (eqA : A \rightarrow A \rightarrow bool) l,} \\
\text{is_list (eqb_correct_on (list_eqb eqA)) l \rightarrow P (D l) \rightarrow Prop,} \\
\text{is_list (is_option (eqb_correct_on deep_eqb)) l \rightarrow P d)
\]

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 1, then `list_eqb_eqb` is correct on the list 1.

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:

\[
is_list (is_option (eqb_correct_on deep_eqb)) l \rightarrow
\]

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

\[
is_list (is_option (eqb_correct_on deep_eqb)) l \rightarrow
\]

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

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

\[
\text{fun l (IH : is_list (is_option (eqb_correct_on deep_eqb)) l \Rightarrow} \\
\text{let H : eqb_correct_on (option_eqb deep_eqb)) l \Rightarrow} \\
\text{list_eqb_correct_aux (option_eqb deep_eqb) l
}\]
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.

\[\text{Inductive tuple } A \times_1 := \{ l : \text{list } A; p : \text{length } l = 1 \}.\]

\[\text{is_tuple functor : }\]
\[
\forall (P \ Q : A \rightarrow \text{Type}), (\forall x, P x \rightarrow Q x) \rightarrow
\forall (i : \text{nat}) (Pi : \text{is nat } i),
\forall (x : \text{tuple } A p), \text{is_tuple } A p i Pi x
\]

The argument \(Pi\) is required in order to feed \(\text{is_tuple}\), 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 \(\text{is nat } i\) 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 \textit{list \_eqb \_correct \_aux} are turned into proper lemmas for the final user by providing their last argument, namely a proof that \(\text{is list } (\text{\_eqb \_correct \_on eqA}) \ l\). In order to provide this argument, we first synthesize this general lemma, which shows \(\text{is list } P\) for any predicate \(P\) which is true on its entire domain.

\[\text{Fixpoint } \text{list is list } (A) P (H : \forall x, P x) l \{ \text{struct } l \} : \text{is list } P \ l :=
\]
\[
\begin{align*}
\text{match } l \text{ with } & \\
| \text{nil} \Rightarrow \text{is list } P \text{ nil} & \\
| \text{cons } x \text{ xs} \Rightarrow \text{is list } P \ x \ (H \ x) \ xs \ (\text{is list } P \ H \ xs) & \end{align*}
\]

Then we can prove the correctness lemma as follows:

\[
\text{Lemma list_eqb_correct } A \ e q A (\text{HeqA} : \forall x, \text{eqb_correct on eqA} x) : \forall l, \text{eqb_correct on } (\text{list_eqb eqA}) \ l.
\]

\[\text{Proof.} \]
\[
\begin{align*}
\text{move} \Rightarrow 1; & \\
\text{exact: list_eqb_correct_aux eqA 1} & \\
(\text{list is list } (\text{eqb_correct on eqA}) \text{ HeqA} 1). & \end{align*}
\]

Qed.

That is, if \(\text{eqA}\) is a correct equality test on \(A\), then \(\text{list_eqb eqA}\) is a correct equality test on \(\text{list } A\).

### 7 Implementation and benchmarks

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

<table>
<thead>
<tr>
<th>file</th>
<th>size</th>
<th>generates</th>
</tr>
</thead>
<tbody>
<tr>
<td>eqType</td>
<td>151</td>
<td>HOAS (and validation)</td>
</tr>
<tr>
<td>tag</td>
<td>55</td>
<td>tag</td>
</tr>
<tr>
<td>fields</td>
<td>264</td>
<td>fields_t, fields, construct, constructP</td>
</tr>
<tr>
<td>eqb</td>
<td>208</td>
<td>eqb_fields, eqb</td>
</tr>
<tr>
<td>eqbcorrect</td>
<td>274</td>
<td>eqb_correct, eqb_reflexive</td>
</tr>
<tr>
<td>eqbP</td>
<td>51</td>
<td>eqb_OK</td>
</tr>
<tr>
<td>total</td>
<td>803</td>
<td>feqb</td>
</tr>
</tbody>
</table>

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 \(\text{feqb}\) in the present paper, nevertheless we give a taste of it presenting the rules for synthesizing \(\text{eqb_fields}\) in Figure 5.

The code is divided into two procedures: \(\Gamma, \sigma \vdash i \mid j \sim_\lambda B\) and \(\Gamma, \sigma \vdash s \mid \text{equality test } f\) where \(\text{equality test } f\) is of type \(\text{t} \rightarrow \text{f} \rightarrow \text{bool}\). We expect it to contain rules for known \textit{eqTypes}, that is type for which \(\text{feqb}\) has previously synthesized the equality test, for example:

\[
\Gamma \supset \left\{ \begin{array}{l}
\forall n m, \text{word } n \approx \text{word } m \mapsto \text{word_eqb } n \ m \\
n \ a \ b \ f, \ a \approx b \mapsto f \implies \text{word_eqb } a \ b \\
\end{array} \right\}
\]

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

The predicate \(\Gamma \vdash a \approx b \mapsto f\) allows to build the composition of the rules in \(\Gamma\). For example, given the \(\Gamma\) above:

\[
\Gamma \vdash \text{list } \text{nat } \approx \text{list_eqb } \text{nat } \text{nat } \text{nat_eqb}
\]

When a type expression in \(\Sigma\) needs to be used in a Coq term we invoke the conversion procedure \([,|\sigma]\). This code is straightforward, since \(\Sigma\) is a fragment of Coq’s syntax.

The only caveat is how to map variables bound in the HOAS to Coq terms. We carry \(\sigma\) which is an association between variables crossed in the input \(I\) and Coq terms. For example \([x \mapsto y]\) extends the map linking \(x\) and \(y\) (in \(\Sigma\)) with \(a\) and \(b\) respectively.

The rules for \(\Gamma, \sigma \vdash i \mid j \sim_\lambda B\) recursively traverse two copies of the same inductive type declaration \(I\), namely \(I \vdash i \mid j \sim_\lambda B\).
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 \( \Gamma \) a rule \( a = b \mapsto f \) 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 \( \sim_A \).

The first two rules of \( \sim_A \) are identical: they compare the field variables \( X \) and \( Y \) by looking up a function \( F \) in \( \Gamma \) using their types. If these arguments have the type of the inductive type being currently processed, then the lookup gives \( \text{rec} \). If their type is one of the type parameters of the inductive type, then the test abstracted by rules \((tp)\) is used. If their type is an already known type, possibly a container, we expect \( \Gamma \) 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 \( \Gamma \vdash a \equiv b \mapsto t \).

7.1 Benchmarks

To get an idea of the performance of \( \text{eqb} \) in practice, we performed a few benchmarks comparing \( \text{eqb} \) 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 \( \text{variants} \), i.e. inductive types where all constructors have no arguments. The second family is \( \text{records} \), 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, \( \text{eqb} \), 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 variant types (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 \( \text{eqb} \) and Scheme Equality where performances are really similar.
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 \texttt{feqb}. It is only after eliminating unrelated slowdowns from Coq-Elpi (on which \texttt{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 \texttt{feqb}. This engineering work is an additional contribution of the current paper and also explains why Coq-Elpi’s \texttt{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 \texttt{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). \texttt{decide equality} and \texttt{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 \texttt{feqb} is not a problem in practice.

The other main goal when designing \texttt{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 \texttt{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 \texttt{vector} is taken from the standard library of Coq as an example of indexed data type.

```
Inductive forest := | empty | add : tree -> forest -> forest

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

While \texttt{feqb} behaves very well, it does not support these two types. It does not support indexed data types such as \texttt{vect} like every other tool except Equations. It also inherits from Coq-Elpi the limitation of not covering mutual inductive types such as \texttt{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.
Figure 7. Tools coverage of inductive types

References


