General Automation in Coq through Modular Transformations

Valentin Blot
LMF, Inria, Université Paris-Saclay*
Valentin.Blot@inria.fr

Louise Dubois de Prisque
LMF, Inria, Université Paris-Saclay*
Louise.Dubois-de-Prisque@inria.fr

Chantal Keller
LMF, Université Paris-Saclay*
Chantal.Keller@lri.fr

Pierre Vial
LMF, Inria, Université Paris-Saclay*
Pierre.Vial@inria.fr

Whereas proof assistants based on Higher-Order Logic benefit from external solvers' automation, those based on Type Theory resist automation and thus require more expertise. Indeed, the latter use a more expressive logic which is further away from first-order logic, the logic of most automatic theorem provers. In this article, we develop a methodology to transform a subset of Coq goals into first-order statements that can be automatically discharged by automatic provers. The general idea is to write modular, pairwise independent transformations and combine them. Each of these eliminates a specific aspect of Coq logic towards first-order logic. As a proof of concept, we apply this methodology to a set of simple but crucial transformations which extend the local context with proven first-order assertions that make Coq definitions and algebraic types explicit. They allow users of Coq to solve non-trivial goals automatically. This methodology paves the way towards the definition and combination of more complex transformations, making Coq more accessible.

1 Introduction

Interaction *vs.* automation. The Coq proof assistant allows us to prove theorems *interactively*: that is, given a Coq goal `G` (*i.e.*, a statement to prove), the user has to write a *proof* `P` consisting of inference steps with *ad hoc* premises and conclusion. Then, they can check whether the given proof is correct by writing `Qed`: the proof is then certified (or rejected) by a **logical kernel** implemented in OCaml which type-checks the proof-term constructed by the user. In practice, proof assistants support a limited form of *automation*: in Coq, the user can use keywords called *tactics*, which operate logical transformations on a goal and its hypotheses until solving it, *e.g.*, a tactic may try to apply some inference/typing rules as many times as possible and relieve the user of some bureaucratic work. An important idea to keep in mind is that tactics may use elaborate tools (including plugins, auxiliary softwares, *etc*...) to produce whole proof terms, *including flawed ones*. However, eventually, **any proof term,** whether it is written by the user or produced by a tactic or an external tool, **will be type-checked by the kernel of Coq**, and this is how the trust in Coq lies in the one we have in its kernel.

Anyway, automation in Coq is limited: in most cases (including many trivial proofs), the user has to provide the bigger part of the structure and the steps of the proof: we will give a simple example just below.

This situation is strikingly different from *Automated Theorem Provers*, such as first-order solvers or **SMT solvers**, which find a proof without the user having to find the proof steps if the problem is expressed in a suited way. However, these automated provers have two main limitations:

*This work is funded by a Nomadic Labs-Inria collaboration.

Submitted to:
PxTP 2021

	Automatic provers	Coq
Expressivity	First-order logic	**CIC**
Safety	Trust the whole software	**Kernel for proof checking**
Automation	**Automatic proof**	User-guided proof

Table 1: Pros (bold) and Cons of automatic provers and Coq

- One needs to trust their whole code, and not a small kernel.
- Most of them handle only **first-order logic (FOL)** whereas Coq is based on the **Calculus of Inductive Constructions (CIC)**, which is far richer: (1) CIC is polymorphic, and thus allows quantifying not only over objects (such as integers) but also on types, *e.g.*, in CIC, one may define polymorphic list concatenation `++` whose type is `forall (A : Type), list A →list A →list A` (2) CIC features dependent typing, so that types may depend on terms and other types, *e.g.*, `Vec A n` specifies the type of vectors of length `n` on the carrier `A`, and thus depends on the number `n` and the type `A` (3) CIC enables higher-order computation.

The logical expressiveness of CIC makes it possible to handle complex aspects of programming languages and to prove properties about programs in Coq, in particular fine-grained specifications that an automatic prover based on first-order logic could not understand. For instance, dependent typing allows the specification of equalities beyond decidable datatypes, *e.g.*, in Coq one may directly specify that two *functions* are equal with an equality `f = g`, which is not possible in general in automatic solvers. Yet, it is precisely because most automatic provers only tackle FOL that they admit a lot of powerful automation.

Improving the automation of Coq within first-order logic. To sum up the situation, on the one hand, we have Coq, which is based on CIC and is a rich specification language, but for that reason, it is difficult to automatize its proof search. On the other hand, automatic provers are based on a more limited logic (FOL) but feature extremely efficient proof-search heuristics. Moreover, the trust we have in the former is based on the implementation of a small, isolated kernel. This can be summarized in Table 1.

Such differences are to be expected, since proof assistants and automatic provers do not have the same purposes and uses. However, as it turns out, in Coq, even a *simple proof of first-order logic*, *e.g.*, on decidable datatypes, can be tedious and force the user to provide a lot of input, whereas an automatic prover would automatically find a proof. Introducing more automation in Coq would thus be very useful to spare the user some trivial parts of a proof.

Let us illustrate this with two examples.

Example 1 (Dealing with datatypes)**.** We use the function `hd_error` from the `List` module of Coq Standard Library, of type `forall {A:Type}, list A →option A` (the curly brackets mean that `A` is an implicit argument, that is to say it can be omitted) defined by `hd_error l = Some x` when `l = x :: l0` (*i.e.*, `l` not empty, `x :A`, `l0` a list of elements of type `A`) and `hd_error [] = None`. We then prove:

```
Goal forall l (a:A), hd_error l = Some a → l <> nil.
```

in a context where `A` is a type variable[1]. A typical Coq proof is:

[1]This is handled in Coq thanks to the section mechanism. It allows the user to introduce section-local variables that can be used in other declarations in the section.

```
Proof.
 intros l a H. intro H'. rewrite H' in H. simpl in H. discriminate H.
Qed.
```

The statement is straightforward, its proof relies on the fact that two different constructors always output different values, *e.g.*, `Some x` and `None` cannot be equal, neither `x :: l` and `[]`. Yet, in Coq, the user (especially a non-expert one) has to be highly precise in the way they compose their keywords, even though they would not even bother writing the proof on paper. It may even be more frustrating that this is a statement of first-order logic and as such, it would automatically be dealt with by a first-order prover, provided it knows about (1) the definition of the function `hd_error` and (2) the datatypes `list` and `option`.

Example 2 (Calling lemmas)**.** In this example, we recall some of the annoyances met while using lemmas in Coq. We consider a Boolean search function:

```
Fixpoint search {A : Type} {H: CompDec A} (x : A) l :=
  match l with
  | [] ⇒ false
  | x0 :: l0 ⇒ eqb_of_compdec H x x0 || search x l0
end.
```

Not going into details, the implicit argument `H: CompDec A` specifies that `A` is a *decidable type*, *i.e.*, that equality is decidable on `A`. Boolean equality can then be computed with the function `eqb_of_compdec H`[2].

By induction, we prove `search_app : forall {A: Type} {H : CompDec A} (x: A)(l1 l2: list A), search x (l1 ++ l2)= (search x l1)|| (search x l2).` Now, let us consider a typical Coq proof of the following simple statement:

```
Lemma search_lemma : forall (x: Z) (l1 l2 l3: list Z),
  search x (l1 ++ l2 ++ l3) = search x (l3 ++ l2 ++ l1).
Proof.
  intros x l1 l2 l3. rewrite !search_app.
  rewrite orb_comm with (b1 := search x l3).
  rewrite orb_comm  with (b1 := search x l2) (b2 := search x l1).
  rewrite orb_assoc. reflexivity.
Qed.
```

As expected, the proof uses the commutativity and the associativity of Boolean disjunction `||`, which can be found in the module `Coq.Bool.Bool` of the Standard Library. We start by using the lemma `search_app` 4 times to eliminate `++` from the statement, which can be done automatically with the `!` operator that rewrites as much as needed. However, the order of commutativity/associativity uses must be carefully chosen. Moreover, the instance of the bound variables of the commutativity lemma `forall b1 b2 : bool, b1 || b2 = b2 || b1` must be specified: the same proof without specifying them fails, since Coq would try to rewrite only the leftmost-outermost `||`, which would not work. Actually, although the user recognizes the proof as trivial, they must keep a keen eye at each step on the local proof context to determine which lemma must be used with which instances, and sometimes, they have to print lemmas to identify the names of bound variables they need to instantiate. All this may appear as a nuisance compared to writing a formal proof with a pencil and a paper and deter new users of Coq.

[2]There are multiple ways of representing a decidable equality in Coq; we use the representation from the SMTCoq plugin since we will use it as a back-end (see later).

Our contribution: linking first-order Coq goals with automated provers An interesting observation about Examples 1 and 2 is that their statements and proofs pertain to first-order logic (with decidable equalities). As such, they *should* be automatized.

In this paper, we provide a methodology to reconcile Coq goals with the logic of first-order provers. This methodology consists in

1. implementing pairwise independent logical transformations: each transformation encodes one aspect of Coq logic as formulas in a less expressive logic (until reaching first-order formulas in Coq) and establishes a soundness proof of this encoding;
2. providing strategies to combine these transformations in order to automatically translate Coq goals into fully explicit first-order logic goals.

We also implement this methodology as a new Coq tactic called `snipe` (for the bird known in french as *bécassine des marais*) so that the proofs of the two lemmas presented above become only one single call to this tactic, passing the required lemma `search_app` in the second case.

The tactic `snipe` is two-fold.

1. As presented before, we implemented five small logical transformations, and a strategy that combines them. The transformations presented in this paper deal with definitions, datatypes and polymorphism, and thus the strategy transforms a Coq goal containing these features into a fully first-order goal.
2. Then, we use the SMT solver veriT as a back-end to discharge the obtained first-order goal, available through the SMTCoq[3] plugin[9], which enables safe communication between Coq and SMT solvers.

In step 2, we benefit from the automation provided by the SMT solver veriT, which is able in particular to perform Boolean computation (as in Example 1), Linear Integer Arithmetic (LIA) or relieve the user of the burden of finding the right instantiations of the lemmas. An important observation is that in step 2, the use of SMTCoq could be replaced by any tactic solving first-order logic.

The remainder of the paper is dedicated to explaining the concept of `snipe` in details. In the next section, we explain our methodology. In § 3, we present examples of useful logical transformations together with their implementations. In § 4, we explain the full `snipe` tactic, as well as more examples of its power. We finally present the state of the art before concluding.

The source code can be found at `https://github.com/smtcoq/sniper/releases/tag/pxtp21`.

2 From the logic of Coq to first-order logic: modular transformations

From the Calculs of Inductive Constructions to first-order logic As we saw in the introductory examples, Coq, which is based on CIC, is mostly not automatized even for simple proofs: while it now enjoys very efficient decision procedures for dedicated theories [2, 11], attempts for general automation has not truly succeeded yet (see § 5 for a detailed comparison).

Our contribution is to propose a new approach for general automation, based on (1) small transformations of a CIC goal towards a goal of first-order logic (2) stating and proving first-order properties in the local context of a Coq proof. Then, the first-order goal and the associated first-order properties can be sent to any external automated prover based on first-order logic. As we will see in the next paragraph, the user may choose which transformation they apply.

[3] SMTCoq is available at `https://smtcoq.github.io`.

We choose FOL as our target logic because a lot of work has been done to automatize reasoning in this logic. So, after calling our transformations, the user can choose any way to automatically prove a first-order goal: *e.g.*, an automatic prover certified in Coq, a tool calling external solvers, tactics like `firstorder` or `crush`[6]. In § 4, we provide a fully-automated tactic that applies the transformations presented in this paper and solves the resulting first-order goal using the SMTCoq plugin.

Modular and independent transformations Formally, a logical transformation from the language of Coq to itself is a function f from the terms and formulas of Coq to themselves. Leaving aside the details on the nature of the function f for now, we are interested in **sound transformations**, *i.e.*, transformations f such that, given any Coq statement G in the domain of f, we have $f(G) \Rightarrow G$. This means that it is enough to prove $f(G)$ for G to be valid. More precisely, we are interested with sound transformations f such that $f(G)$ is a first-order formula: indeed, if a Coq goal G is left to prove, we may transform G into $f(G)$ and then send $f(G)$ to an automated theorem prover dealing with first-order. If this succeeds, it means that G is valid.

We are actually also interested in functions g from a subset of Coq formulas such that, given a Coq statement G, $g(G)$ outputs (a list of) *valid* first-order logic statements in Coq that may help proving G. We also call such a function g (which produce auxiliary first-order statements) a sound transformation.

Our approach is to develop modular and independent transformations: each of them encodes one important aspect of CIC. The aim is to tackle only one aspect at a time: it facilitates the proof of soundness whenever we need to write one in Coq. Indeed, as the transformations are simple, they preserve as much as possible the structure of the source formula. We expect that they are easier to implement than a bigger encoding. In addition, this methodology facilitates the debugging and allows us to know precisely which fragment of Coq we can handle. The transformations can be composed in different ways: we can use them separately, combine them all, or only some of them, either by using a default tactic provided for a user who does not want to think about which method they should choose, or by writing them one by one. Moreover, they are independent from the technology used for first-order proving in the end. This is illustrated by the left part of Figure 2.

Certifying and certified transformations There are two ways for writing logical transformations f from Coq to itself. As we will see, both need to resort to the **meta-language** of Coq at some point: they feature functions which cannot be defined in the core language of Coq, but only in extra-layers outside its kernel. However, they do not work the same way from the logical point of view.

1. **Certified transformations.** The function f may be a Coq function. This relies on an internal representation of Coq terms inside Coq, that we call `term` (see Example 3). It comes with two meta-language transformations: the **reification** takes a Coq term as a parameter and outputs its reification (in the type `term`) and the **dereification** is the converse. In this situation, f is a Coq function (*not* a meta-function) of type `term` $\rightarrow$`term` and we may prove that, for all `t` of type `term`, f `t` implies `t` (up to some implicit reification). We call this kind of transformation a *certified* one, because the soundness is proved *once and for all*, *in a Coq statement*.

2. **Certifying transformations.** The function f may be a meta-language function, *i.e.*, f is not a Coq function. In that case, it is *not possible* to write a Coq statement specifying that, for instance, for all G of type `set`, $f(G)$ implies G. However, we may write another meta-language function g such that, for all G, $g(G)$ generates on the fly a proof of $f(G) \rightarrow G$. Such a transformation is said to be *certifying*, because its soundness is not previously established. It takes a local context and

a *specific* goal, and it operates the transformation, which will be type-checked at the end by the kernel (when `Qed` is written) *every time we use it*.

The differences between certified and certifying transformations are summarized in Figure 1. In the work presented in this article, we follow the paradigm of certifying transformations. Indeed, the former require we work only with the reified syntax of the terms of CIC and even for a simple transformation, the proof of soundess is hard (thousands lines of code). But in some situations, it could be useful to use this solution, because it covers all cases.

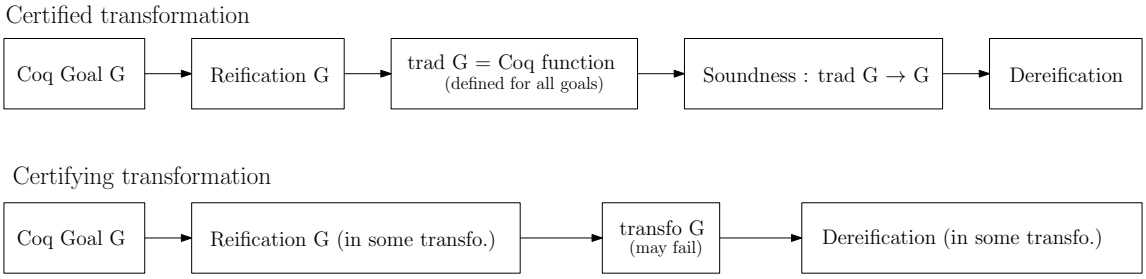

Figure 1: Difference between certified and certifying transformations

Metalanguage tools Most logical transformations analyze the syntax of terms in the source language and produce new statements. But in Coq, we do not directly have access to the syntax of terms. This is the reason why we need meta-programming tools as we saw in the previous paragraph. Many different tools are available now (see § 5), and we used two of them: Ltac [8] and MetaCoq [17], which offer different advantages.

- Ltac is a tactic language in Coq which is based on pattern matching, recursion and backtracking. It allows its users to match on the syntax of Coq terms and also on the local context, for building new goals and terms. It is also possible to execute basic tactics which operate on kernel terms, called tacticals, in a Ltac function. Ltac cannot be seen as a proper meta-programming language as there is no data type which contains the reified syntax of Coq terms. In our transformations, we use Ltac whenever we need to use a particular function on every hypotheses of the context. Once this function is applied, its result is certified in Ltac by a combination of simple Coq tacticals. Sometimes, it is also useful when a superficial access to the syntax of a Coq term is required, but Ltac is less convenient for analyzing the syntax completely.
- MetaCoq enables a more fine-grained analysis on Coq terms. Indeed, this plugin includes an inductive type `term` which corresponds exactly to the Coq counterpart of OCaml kernel terms, and comes with the reification and dereification transformations described above. As it offers an easy access to the syntax of Coq terms, our transformations often use MetaCoq. The problem with MetaCoq terms is that the reified syntax is not very readable: the variables are represented by de Brujin indexes, and there is no notation to reduce the size of the terms. This is the reason why we prefer Ltac to get information about the initial goal, and MetaCoq when we need to build a new term. Such a flexible approach is possible, since we build certifying (not certified) transformations.

Example 3 (Reification)**.** The MetaCoq representation of the Coq term `forall (A:Type), A →A` is `tProd (name "A")(tSort type_reif)(tProd unnamed (tRel 0)(tRel 1))`. The constructor `tProd`

corresponds to the Coq `forall` dependent product binder (and the $\rightarrow$ is simply a notation for a non-dependent product). The type of the variable in this product is `type_reif` (not going into details, this is the MetaCoq reification of `Type`). `tRel 0` and `tRel 1` are the variables, represented by their De Brujin indexes.

3 Examples of transformations

We describe now some of the transformations we have implemented. Most of them are motivated by the following facts.

1. When Coq and an external automated theorem prover communicate, **a lot of symbols defined in Coq will not be intrepreted**. For instance, the function `hd_error` used in Example 1 may be uninterpreted in the external prover, which does not know anything about it except its type. In general, pattern-matching may be uninterpreted. Moreover, algebraic data types and their constructors may also be left uninterpreted. For instance, an external prover may not know that lists have basic properties, *e.g.*, `x1 :: l1 = x2 :: l2` implies `x1 = x2` and `l1 = l2`, or that `[] <> x :: l`, whereas these two *first-order statements* just come from the definition of the type `list`.
2. **Coq handles higher-order objects (in particular, equalities about such objects), whereas first-order provers do not**. For instance, `fun x =>(x + 1)** 2 = fun x =>x ** 2 + 2 x + 1` is an equality between functions (of type `Z` $\rightarrow$`Z`) and cannot be directly interpreted in first-order logic. Yet, it may be sufficient to consider its first-order consequence `forall x, (x + 1)** 2 = x ** 2 + 2 * x + 1`, which is a quantified equality on the type `Z`.

Such transformations, while simple at first glance, are already mandatory to give a bridge between a Coq goal and a first-order prover: the first kind of transformation makes the global context explicit, whereas the second kind encodes higher-order aspects.

As explained in the previous section, we implemented these encodings as certifying transformations (cf. Figure1), using meta-programming. Transformations 3.1 and 3.5 do not need reification while others do. The approach of each transformation is the following: by scanning the goal, it states and proves *in Coq* various auxiliary lemmas of first-order logic about the terms encountered in the goal. The proofs are done easily by applying Coq basic tactics. These lemmas are stored in the local context. It may also perform some transformations on the goal, so that it becomes first-order.

3.1 Definitions

The first transformation[4] makes user-defined terms explicit. For instance, on Example 1, it will add the following assertion to the local context:

```
hd_error_def : hd_error =
  (fun (A : Type) (l : list A) ⇒
    match l with | [] ⇒ None | x :: _ ⇒ Some x end)
```

This assertion is not first-order yet (it will be transformed again by the next two transformations) but it allows one to have access to the definition of the constant.

For its implementation, it mainly uses the Coq tactic `unfold`, which takes an identifier (the name of a previously defined term) as a parameter and replaces it by its definition. This tactic, `get_def`, takes the

[4] See file `definitions.v`.

name of a constant and adds its definition as a proven hypothesis. The example right above is obtained by applying this tactic on the identifier `hd_error`. While `unfold` may cause the goal or the hypotheses to become verbose and not particularly understandable for the user, `get_def` will keep the definitions in separated statements. In the tactic which combines all of our transformations, we apply `get_def` recursively to all the definitions that occur in the hypotheses and in the goal.

3.2 Expansion

Note that in the example for the tactic `get_def`, the added hypothesis pertains to a higher-order function (because of `fun (A : Type)...`). A good way to make the equality deal with first-order objects is to apply the functional definition to an arbitrary argument of its domain[5]. That is, instead of writing $f = \lambda x.\, t(x)$, we write: $\forall x, f(x) = t(x)$ where x is a fresh variable, and t the unfolded definition of f. We did not use a transformation which encodes partial application with an applicative symbol because it may lead to bigger terms and it is not necessary here. We have left it for future work and more complex cases. The tactic takes a hypothesis `H` of the form $t = u$ where $t : A_1 \to \ldots \to A_n \to B$, and asserts and proves automatically the hypothesis

$\forall x_1 : A_1, \ldots, \forall x_n : A_n,\ t(x_1, \ldots, x_n) = u(x_1, \ldots, x_n)$.

If we come back to our running example, once we get the axiom `hd_error_def` in our context, we can apply `expand hd_error_def` to obtain a new assertion in the local context:

```
H0 : forall (A : Type) (l : list A), hd_error l =
  match l with | [] ⇒ None | x :: _ ⇒ Some x end
```

The proof of this statement is really simple in Coq. It suffices to use the tactic `rewrite H`, followed by `reflexivity`, which checks that both members of the equality are convertible in CIC. But the construction of the statement is harder. First, the tactic takes a hypothesis `H` (`hd_error_def` in our case) and finds, thanks to the Ltac value-function `type of`, its type T, which must be an equality $t = u$. In order to have an easy access to the domain and codomain of t and u, we reify T. That is, we call a tactic `quote_term` from the MetaCoq plugin, which takes a Coq term and returns its reified syntax. Then we can perform syntactic operations on this term by implementing auxiliary functions directly in Coq. Our tactic is thus divided in four parts:

- The main tactic `expand` reifies the hypothesis `H` and calls the auxiliary functions.
- The first auxiliary function `list_of_args_and_codomain` finds the common type of t and u: $A_0 \to \ldots \to A_n \to B$ and returns the pair $([A_0; \ldots; A_n], B)$. All the terms involved here are MetaCoq terms.
- The second auxiliary function `gen_eq` constructs the reified equality that we want. It is defined recursively on the list $[A_0; \ldots; A_n]$ by the following equations.
  `gen_eq` $([], B, t, u) \triangleq t =_B u$
  `gen_eq` $([A_0; \ldots; A_n], B, t, u) \triangleq \forall x_0 : A_0,$ `gen_eq`$([A_1; \ldots; A_n], B, t\ x_0, u\ x_0)$ The `gen_eq` function deals with De Brujin indices by lifting these at every recursive call.
- The generated equality is then unquoted and proved by using Coq tactics.

3.3 Elimination of fixpoints

For the sake of simplicity, we have so far taken the example of a non-recursive function (`hd_error`). However, it is very often the case that functions are defined recursively on datatypes. In this section,

[5] See file `expand.v`.

we take the example of the recursive function `length` computing the number of elements in a list. If we expand its definition as presented in the previous sections, we get:

```
H : forall (A : Type) (l : list A), length l =
  (fix length_anon (l : list A) : nat :=
    match l with | [] ⇒ 0 | _ :: l' ⇒ S (length_anon l') end) l
```

As `length` is a fixpoint, its definition is hidden in an anonymous function denoted by `fix`. Again, we transform this expression to make it more intelligible by automatic provers. This is the role of the tactic `eliminate_fix`[6] that replaces the anonymous function with the constant it defines. Here, `eliminate_fix H` asserts and proves the new hypothesis `H0`:

```
H0 : forall (A : Type) (l : list A), length l =
  match l with
    | [] ⇒ 0
    | _ :: l' ⇒ S (length l')
  end
```

Thanks to this tactic, we now have access to the body of the definition of `length` as for non-recursive functions.

3.4 Elimination of pattern matching

Definitions by pattern matching, such as the examples of the previous two subsections, are not understandable for automated provers. More generally, they are not part of the syntax of first-order terms. Thus, instead of getting a function defined by pattern matching, we would like to have one statement for each pattern. We implemented a tactic[7] which does precisely this. In order to present it, let us continue our example from 3.2. The tactic `eliminate_pattern_matching H0` (where `H0` is the hypothesis generated in 3.2), produces two hypotheses:

```
H1 : forall (A : Type), hd_error [] = None
H2 : forall (A : Type) (x : A) (l : list A), hd_error (x::l) = Some x
```

The tactic works in five steps. Note that the formula to which it is applied must be of the form `forall (x0: A0) ... (xi: Ai), E[match xi with ...]` where `E` is an environement with (possibly) free variables.

- The index i of the matched variable is computed with a combination of a dummy subgoal and a metavariable allowing to pass on the result to the main goal.
- The formula is reified and the reified types $A_0, \ldots, A_i$ are retrieved.
- An independent tactic scans the global environment in which the inductive definition of A_i can be found. It returns the list of its reified constructors $[C_0; \ldots; C_j]$ and their reified types $[T_{0,0} \to \ldots \to T_{0,n_0} \to A_i; \ldots; T_{j,0} \to \ldots \to T_{j,n_j} \to A_i]$.
- For each constructor C_k, we construct the following statement:

  ```
  forall (x0: A0) ... (xi−1: Ai−1),
    forall (ak,0: Tk,0) ... (ak,nk : Tk,nk),
    E(match Ck ak,0 ... ak,nk with ...)
  ```

- Each statement is unquoted, asserted and proved by `intros; rewrite H0; reflexivity`.

[6] See file `elimination_fixpoints.v`

[7] See file `elimination_pattern_matching.v`.

3.5 Monomorphization

Most automatic provers do not support polymorphism. In other words, they cannot prove lemmas about functions that can be defined on any type of data. Typically, as we saw in Example 2, the lemma `search_app` is polymorphic, and thus cannot be sent directly to most provers. However, only its instance on `Z` is useful for proving `search_lemma`. This transformation will add the following statement to the local context:

```
search_app_Z : forall (x: Z) (l1 l2: list Z), search x (l1 ++ l2) =
   (search x l1) || (search x l2)
```

There are various ways to handle polymorphism[3][4]. Among them we chose a monomorphization based on instantiating the polymorphic types with chosen ground types from the context.

To write the monomorphization tactic[8], the meta-programming language Ltac was our main tool. Indeed, we did not need a detailed access to the syntax of the terms as MetaCoq provides, but we wanted to apply the same tactic to all the hypotheses in a given context (or to a list of polymorphic lemmas). Ltac allows matching on the local context in a rather simple way, thanks to the tactic `match goal`. In details, the instantiation tactic tries to match all hypotheses in the local context whose type P is a quantified hypothesis: $\forall A : Type, P'$. MetaCoq is useful here, to check that A has type `Type`.

Then, the monomorphization tactic scans the goal and instantiates the variable of type `Type` with all the subterms of type `Type` in the goal. All the generated hypotheses are automatically proven by the tactic `specialize`.

In order to avoid infinite loops, the instantiated hypothesis is not added in the context if it is already present. The tactic can also take parameters (polymorphic lemmas), and they are monomorphized in the same way.

In the future, we will run benchmarks to measure the performance of our tactic. This may help us to develop heuristics for choosing the instances of lemmas efficiently.

3.6 Interpreting Algebraic types

Algebraic datatypes are a special case of inductive types which do not use non-prenex polymorphism or type dependencies. The epitomy of such a type is perhaps `list` that we used in our examples:

```
Inductive list (A : Type) : Type :=
  | [] : list A
  | cons : A → list A → list A.
```

where the constructor `cons` has the infix notation `::`. When one is familiar with inductive types, one knows that this declaration specifies how equality works on the type `list`. For instance, `x1 :: l1 = x2 :: l2` implies `x1 = x2` and `l1 = l2`. Moreover, `[] <> x :: l1`. In general, in an algebraic datatype `I` as defined in Coq:

- Each constructor `C` of `I` is injective, that is:

```
forall (x1 y1: A1) ... (xn yn: An),
  C x1 ... xn = C y1 ... yn → x1 = y1 ∧ ... ∧ xn = yn
```

- If `C` and `C'` are two distinct constructors of `I`, then their direct images are disjoint, that is:

[8] See file `elimination_polymorphism.v`.

```
forall (x1 : A1) ... (xn : An) (x1': A1') ... (xp' : Ap'),
  C x1 ... xn <> C' x1' ... xp'
```

Any inhabitant of `I` is obtained from one of the C_i of `I`, that is:

```
forall (x: I),
((exists x_{1,1} : A_{1,1})...(exists x_{k_1,1} : A_{1,k_1}), x = C_1 x_{1,1}... x_{1,k_1} )
∨...∨((exists x_{n,1} : A_{n,1})...(exists x_{n,k_n} : A_{n,k_n}), x = C_n x_{n,1}... x_{n,k_n} )
```

Note that the above propositions are statements of first-order logic and as such, may be communicated to a first-order automatic theorem prover. However, the third property is written with existential quantifiers which are not treated by the back-end we use in our proof of concept (see Sec 4), so our tactic does not generate this property for the moment.

We define the tactic `interp_alg_types`[9] that finds the algebraic datatypes of Coq (*e.g.*, `list Z`) which occur in the goal and automatically proves that (1) their constructors are injective (2) the direct images of their constructors are disjoint.

If we come back to Example 1, the datatypes `list` and `option` are both made explicit so that the automatic prover is able to conclude.

4 Proof of concept

As explained in § 2 and the left part of Figure 2, the methodology is to combine such transformations, possibly in different ways, then call an automatic solver.

In this section, we provide a proof of concept: all the transformations in the previous section are combined in a fully automatized tactic called `snipe` which applies them and sends the resulting goal and context to the SMT solver veriT through the SMTCoq plugin. This is illustrated by the right part of Figure 2. We now detail this combination and come back to examples illustrating the `snipe` tactic.

4.1 Proof strategy

As presented in Figure 2 we proceed as follows. We first apply a combination of our transformations in a unique tactic called `scope`. Thanks to the SMTCoq plugin, we send the goal and the local context with additional lemmas obtained by the `scope` tactic to the external solver veriT. Let us describe this two-part process more precisely:

- `scope`: This tactic consists of applying first `interp_alg_types` to all algebraic types in the goal and in the context, except types already interpreted by the SMT solver like $\mathbb{Z}$ or the Booleans. Then it calls the `get_definitions` tactic: it adds new definitional hypotheses in the local context, except for the symbols which are part of the built-in theories of veriT. For instance, the definition of the addition in $\mathbb{Z}$ is not needed. Then, the tactics `expand`, `eliminate_fix` and `elimination_pattern_matching` are applied to the new generated hypotheses. Finally, the monomorphization tactic will assert and prove a new proposition for every polymorphic hypothesis applied to a subterm of type `Type` in the goal. A tuple of Coq lemmas choosen by the user can be added as parameters to `snipe`: the tactic will also try to instantiate them if they are polymorphic.

[9] See file `interpretation_algebraic_types.v`.

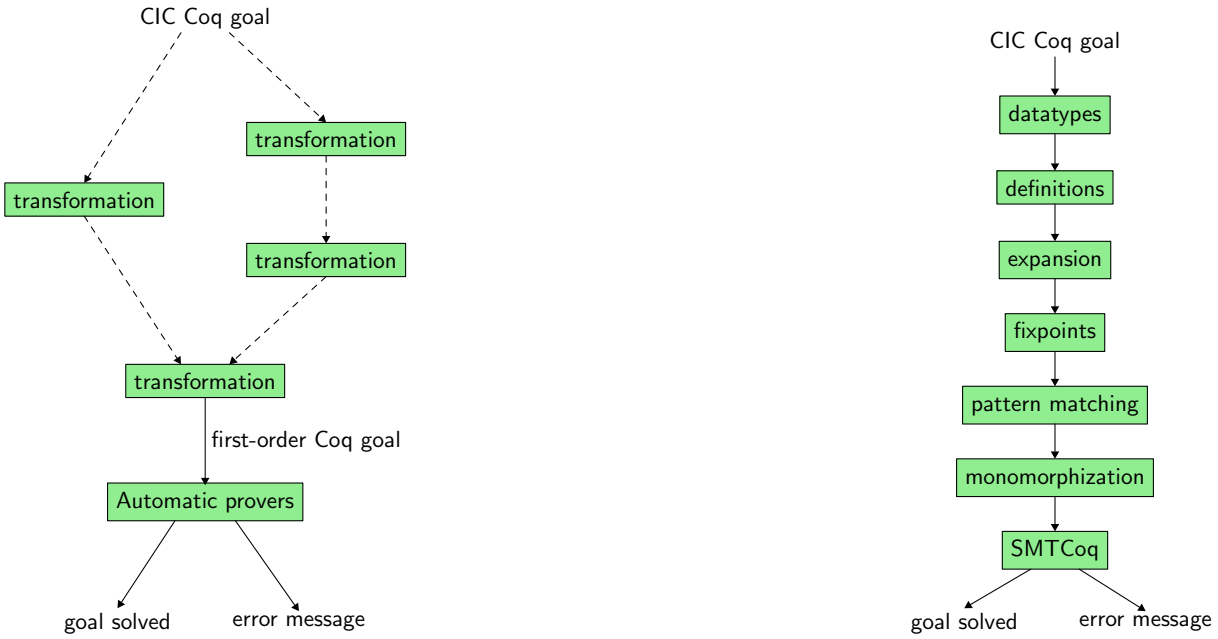

Figure 2: General methodology (left) and proof of concept (right)

- Once `scope` is applied, the tactic `verit` is called. It is an SMTCoq tactic which solves first-order goals in a combination of built-in theories (such as linear arithmetic or congruence) by calling the external SMT solver veriT and reconstructing a Coq proof. Note that:
  - this tactic requires equalities to be decidable (this is the proposition `CompDec A` mentionned in the introduction)
  - its implementation relies on the Coq standard library for machine integers and arrays, which axiomatize these data-structures; that is why the tactic `snipe` also relies on these axioms.

4.2 Examples

Let us go back to our examples[10]. Example 1 is automatically solved with `snipe`. Note however that we need an additional hypothesis `Compdec A`: as previously mentionned the type A has to have a decidable equality for SMTCoq to reason on it. The `scope` part of the tactic adds all the hypotheses about the constructors of types `option` and `list`, and they are instantiated by the variable A. The definition of `hd_error` is also added in the local context. Since it contains a pattern matching, a proposition for every pattern is created and proved. The tactic `verit` transforms the first-order hypotheses into assertions for veriT and solves the goal.

Example 2, about `search`, requires the instantiation of the previous lemma `search_app`. Again, it is solved automatically by the tactic `snipe`, taking this time the lemma `search_app` as a parameter. The proof becomes:

```
Goal forall (A : Type) (H : CompDec A) (x: A) (l1 l2 l3: list A),
  search x (l1 ++ l2 ++ l3) = search x (l3 ++ l2 ++ l1).
Proof. intros A H. snipe @search_app. Qed.
```

[10] See file `examples.v`.

The instantiation of the polymorphic lemma is done by the monomorphization tactic within `scope`. The part of the proof which required a lemma about Booleans and an adequate instantiation of it is automatically handled by the capabilities of veriT to perform propositional reasoning with quantifier instantiation.

More interestingly, we can also prove the intermediate lemma in a very satisfactory way. This lemma is proved by induction on the first list, and induction is currently out of the scope of most automated provers. However, once this induction is done, the user should not worry about proving the sub-cases. This is made possible by our tactic:

```
Lemma search_app : forall {A:Type} {H:CompDec A} x (l1 l2:list A),
   search x (l1 ++ l2) = (search x l1) || (search x l2).
Proof. intros A H x l1 l2. induction l1 as [ | x0 l0 IH]; simpl;
   snipe. Qed.
```

5 State of the art

General automation in proof assistants Improving automation in proof assistants based on Type Theory is a long-standing research topic.

Decision procedures for dedicated theories have met a large success, witnesses the daily used `ring` [11] and `lia` [2] tactics in Coq, deciding equalities respectively in (semi-)ring structures and propositions in the linear integer arithmetic theory. Automatic tactics also exist for propositional and first-order logic, such as `intuition`, `firstorder` or `crush`[6] in Coq. The limitations of these tactics are that they are useless as soon as the reasoning requires handling multiple aspects at one time (such as propositional logic, arithmetic, equalities...), which is very often the case when using an interactive theorem prover.

This is why research in this area moved towards making use of more complex automatic solvers, mainly SMT solvers, which can combine theories with tableau or first-order provers that usually do not natively handle theories but better deal with quantifiers. In this direction, two approaches are usually considered: the *autarkic* approach, which consists in implementing and proving correct the automatic prover in the proof assistant (*e.g.*, the SMT solver `ergo` [13] in Coq, the tableau prover `blast` [14] in Isabelle/HOL, or the first-order solver `metis` in the HOL family [12]), and the *skeptical* approach, which consists in using external provers that output explanations and only checking these explanations at each execution (*e.g.*, SMTCoq [9] in Coq or `smt` [5] in Isabelle/HOL).

However, as we explained, goals in proof assistant usually do not belong to first-order logic, which is the logic handled by these provers. This is why encodings need to be performed. In this direction, the most successful tool is `sledgehammer` [15] for Isabelle/HOL, which mixes both an autarkic and skeptical approches: it encodes a higher-order goal, calls many external solvers in parallel (using lemmas from the global context selected by machine learning), and uses their answers to reconstruct a proof of the original goal, by mixing standard tactics with `smt` or `metis`. This approach was ported to Coq in the `CoqHammer` [7] tool, but with less success.

We identified one main limitation of `CoqHammer` to be proof reconstruction: the tool tries to build a proof of the original goal, but the external solvers proved the encoded goal. This encoded goal is far from the original goal because `CoqHammer` uses a one pass, very complex encoding of CIC into FOL. This was not a problem for Isabelle/HOL whose logic is simpler. One cannot rely on the reconstruction of a proof of the encoded goal, because then it should be proved correct with respect to the original goal, which is very difficult again because of the complexity of the encoding.

This is why we proposed this new approach, where the encoding is a combination of small transformations that can either be certified or output Coq proofs (certifying). Then the resulting goal can

be checked by any approach for first-order proving since we do not need to reconstruct a proof of the original goal. This approach with small, independent transformations was inspired by other tools that use external automatic solvers, in particular `Why3` [10].

As we explained, we are independent of the back-end used to discharge the first-order goal produced by the transformations. We chose SMTCoq in our proof of concept. It restricts us to hypotheses and goals with only universal and prenex quantification, but offers built-in theories, which seems a good trade-off for the kind of goals that are commonly present in Coq. Targeting first-order provers could also be done by providing other strategies that would do less work on quantifiers but encode theories such as linear arithmetic.

We leave for future work a detailed comparison with `CoqHammer`, both in terms of performance and expressivity. For this latter, we are currently theoretically less expressive than `CoqHammer` (since we only handle a small part of Coq logic beyond FOL), but

- we can detail the fragment of Coq logic that we handle, whereas the success of `CoqHammer` is more unpredictable,
- we believe that the approach will scale when implementing more involved transformations.

Recently, automatic provers have pushed towards more expressivity than FOL [1]. Once they can be used in proof assistants such as Coq (currently they do not output certificates for higher-order aspects), it will be very easy to integrate them in our setting: it simply requires unplugging the transformations that deal with the aspects newly handled by these provers. We could also consider using the theory of algebraic datatypes supported by some SMT solvers to handle a sub-part of Coq inductive types.

Meta-programming in Coq The approach by certified/certifying transformations relies on meta-programming. As explained in § 2, we use two meta-programming tools available in Coq: `Ltac` and MetaCoq. These tools are complementary: `Ltac` allows us to handle surface meta-programming very easily, whereas MetaCoq allows us to go deeper into the structure of terms, at the cost of difficult aspects such as De Bruijn indices. We plan for future work to look at other tools that could enjoy both worlds, in particular `Ltac2` [16] and `Coq-Elpi` [18].

6 Conclusion and perspectives

We have developed a methodology to encode some aspects of CIC into FOL by writing independent and modular transformations. This modularity is useful to delimit the features of CIC we can translate, and to combine the transformations in a chosen order. As a proof of concept, we created a Coq tactic `snipe` which performs the transformations described above and calls an external SMT solver. Some Coq proofs are now totally automatized thanks to our tactic.

Now that we have the crucial and basic transformations to extend the local context with first-order hypotheses, we plan to tackle more complex transformations. This future work will help to automatize other aspects of Coq logic. Here is a non exhaustive list of transformations or features we would like to treat:

- **Encoding of higher-order terms.** We may treat them as usual first-order terms and add an applicative symbol @ to the language signature. Thus, $f(x)$ becomes $@(f,x)$.
- **Encoding of inductive predicates.** They are a particular case of dependent typing, and we would like to obtain first-order statements from them as they are very common whenever program specifications are written in Coq.

- **Skolemization.** As we may use external solvers which do not deal with existential quantifiers after applying our tactic `scope`, we would like to be able to encode them and thus perform a skolemization on the goal and the hypotheses.

Another important part of our future work is to improve performance and do a benchmark analysis. This benchmark has to be *qualitative* and *quantitative*, that is, we need to evaluate the efficiency of our tactic and to know how many goals it can solve. In particular, we will compare `snipe` with `CoqHammer`. As previously said, we hope that a more elaborated version of our tactic will do better than `CoqHammer` in the fragment of CIC we chose to deal with.

References

[1] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli & Clark W. Barrett (2019): *Extending SMT Solvers to Higher-Order Logic*. In Pascal Fontaine, editor: *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, *Lecture Notes in Computer Science* 11716, Springer, pp. 35–54, doi:10.1007/978-3-030-29436-6_3.

[2] Frédéric Besson (2006): *Fast Reflexive Arithmetic Tactics the Linear Case and Beyond*. In Thorsten Altenkirch & Conor McBride, editors: *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, *Lecture Notes in Computer Science* 4502, Springer, pp. 48–62, doi:10.1007/978-3-540-74464-1_4.

[3] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu & Nicholas Smallbone (2016): *Encoding Monomorphic and Polymorphic Types*. *Log. Methods Comput. Sci.* 12(4), doi:10.2168/LMCS-12(4:13)2016.

[4] François Bobot & Andrey Paskevich (2011): *Expressing Polymorphic Types in a Many-Sorted Language*. In Cesare Tinelli & Viorica Sofronie-Stokkermans, editors: *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings*, *Lecture Notes in Computer Science* 6989, Springer, pp. 87–102, doi:10.1007/978-3-642-24364-6_7.

[5] Sascha Böhme & Tjark Weber (2010): *Fast LCF-Style Proof Reconstruction for Z3*. In Matt Kaufmann & Lawrence C. Paulson, editors: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, *Lecture Notes in Computer Science* 6172, Springer, pp. 179–194, doi:10.1007/978-3-642-14052-5_14.

[6] Adam Chlipala (2013): *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. Available at `http://mitpress.mit.edu/books/certified-programming-dependent-types`.

[7] Lukasz Czajka & Cezary Kaliszyk (2018): *Hammer for Coq: Automation for Dependent Type Theory*. *J. Autom. Reason.* 61(1-4), pp. 423–453, doi:10.1007/s10817-018-9458-4.

[8] David Delahaye (2000): *A Tactic Language for the System Coq*. In Michel Parigot & Andrei Voronkov, editors: *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, *Lecture Notes in Computer Science* 1955, Springer, pp. 85–95, doi:10.1007/3-540-44404-1_7.

[9] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds & Clark W. Barrett (2017): *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*. In Rupak Majumdar & Viktor Kuncak, editors: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, *Lecture Notes in Computer Science* 10427, Springer, pp. 126–133, doi:10.1007/978-3-319-63390-9_7.

[10] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 - Where Programs Meet Provers*. In Matthias Felleisen & Philippa Gardner, editors: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of*

Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, *Lecture Notes in Computer Science* 7792, Springer, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[11] Benjamin Grégoire & Assia Mahboubi (2005): *Proving Equalities in a Commutative Ring Done Right in Coq*. In Joe Hurd & Thomas F. Melham, editors: *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, *Lecture Notes in Computer Science* 3603, Springer, pp. 98–113, doi:10.1007/11541868_7.

[12] J. Hurd (2005): *System Description: The Metis Proof Tactic*. *Empirically Successful Automated Reasoning in Higher-Order Logic (ESHOL)*, pp. 103–104.

[13] Stéphane Lescuyer (2011): *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq)*. Ph.D. thesis, University of Paris-Sud, Orsay, France. Available at `https://tel.archives-ouvertes.fr/tel-00713668`.

[14] Lawrence C. Paulson (1999): *A Generic Tableau Prover and its Integration with Isabelle*. *J. Univers. Comput. Sci.* 5(3), pp. 73–87, doi:10.3217/jucs-005-03-0073.

[15] Lawrence C. Paulson & Jasmin Christian Blanchette (2010): *Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers*. In Geoff Sutcliffe, Stephan Schulz & Eugenia Ternovska, editors: *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, *EPiC Series in Computing* 2, EasyChair, pp. 1–11. Available at `https://easychair.org/publications/paper/wV`.

[16] Pierre-Marie Pédrot (2019): *Ltac2: tactical warfare*. *CoqPL 2019*.

[17] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau & Théo Winterhalter (2020): *The MetaCoq Project*. *J. Autom. Reason.* 64(5), pp. 947–999, doi:10.1007/s10817-019-09540-0.

[18] Enrico Tassi (2019): *Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq*. In John Harrison, John O'Leary & Andrew Tolmach, editors: *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, *LIPIcs* 141, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 29:1–29:18, doi:10.4230/LIPIcs.ITP.2019.29.