HAL open science

Compositional pre-processing for automated reasoning in dependent type theory

Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, Assia Mahboubi, Pierre Vial

To cite this version:

Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, et al.. Compositional pre-processing for automated reasoning in dependent type theory. CPP 2023 - Certified Programs and Proofs, Jan 2023, Boston, United States. pp.1-15, 10.1145/3573105.3575676 . hal-03901019v2

HAL Id: hal-03901019

https://inria.hal.science/hal-03901019v2

Submitted on 18 Dec 2022 (v2), last revised 20 Feb 2024 (v3)

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.

Compositional pre-processing for automated reasoning in dependent type theory

Valentin Blot
LMF, Inria, Université Paris-Saclay

Denis Cousineau
Mitsubishi Electric R&D Centre Europe

Enzo Crance
LS2N, Inria, Nantes Université
Mitsubishi Electric R&D Centre Europe

Louise Dubois de Prisque
LMF, Inria, Université Paris-Saclay

Chantal Keller
LMF, Université Paris-Saclay

Assia Mahboubi
LS2N, Inria, Nantes Université

Pierre Vial
LMF, Inria, Université Paris-Saclay

Abstract

In the context of interactive theorem provers based on a dependent type theory, automation tactics (dedicated decision procedures, call of automated solvers, ...) are often limited to goals which are exactly in some expected logical fragment. This very often prevents users from applying these tactics in other contexts, even similar ones.

This paper discusses the design and the implementation of pre-processing operations for automating formal proofs in the Coq proof assistant. It presents the implementation of a wide variety of predictible, atomic goal transformations, which can be composed in various ways to target different backends. A gallery of examples illustrates how it helps to expand significantly the power of automation engines.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Higher order logic**; **Type theory**.

Keywords: interactive theorem proving, Calculus of Inductive Constructions, Coq, automated reasoning, arithmetic, pre-processing

ACM Reference Format:
Valentin Blot, Denis Cousineau, Enzo Crance, Louise Dubois de Prisque, Chantal Keller, Assia Mahboubi, and Pierre Vial. 2023. Compositional pre-processing for automated reasoning in dependent type theory. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '23), January 16–17, 2023, Boston, MA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3573105.3575676

This work is licensed under a Creative Commons Attribution 4.0 International License.
CPP '23, January 16–17, 2023, Boston, MA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0026-2/23/01.
https://doi.org/10.1145/3573105.3575676

1 Introduction

Mundane parts of formal proofs are best automated. But for users of the Coq interactive theorem prover [35], finding the appropriate plugin for doing so is not as simple as one may wish for. First, they have to find their way in the jungle of available tools, with usually not much guarantee that it will be robust to future irrelevant changes in the libraries. Moreover, some slight changes in a statement, seemingly innocuous for a human, may cause such a tool to fail. Generally speaking, it is hard to predict failures or efficiency of a given automated reasoning proof command.

Actually, various powerful automated reasoning techniques have been made available to interactive theorem proving. But once the user has done all the high level reasoning (which can hardly be automated) and is left with a goal that should be handled automatically, the bottleneck does not lie so much in the power of automated reasoning *per se*, but rather concerns the pre-processing phase that aligns the formula to be proved with the scope of the core automation engine. The latter is indeed often tool-specific, hardly customizable and little compositional. The lack of off-the-shelf goal transformations for addressing this prerequisite actually significantly hampers proof automation. The present paper discusses how to better design such pre-processing phases, for improved formal proof automation.

The core idea is to put a suite of *independent*, *atomic* goal transformations at the service of an automation engine. In this model, a suitable orchestration of small-scale transformations of various natures (inductive reasoning, theory-specific translations, etc.) shall expand the skills of the automation backend, while retaining predictability and robustness of the latter. This model applies to a variety of backend automation engines, and to an extensible collection of small-scale transformations.

Why pre-processing? Proof commands, also called tactics, implementing formal proof producing automated reasoning essentially fall in two camps, which both involve preparing user goals before resorting to a core automation

engine. The first one consists of formal-proof-producing implementations of decision procedures, *e.g.*, for the equational theory of commutative rings [16], for the first-order theory of real numbers [24], of integers [4] etc. These standalone formal proof automation tactics typically target implementation-specific choices of data structures or definitions, that are not be directly compatible with users' choices. Goals involving data structures from external libraries have thus to undergo a preliminary translation phase before being passed to the core tactic [29].

The second family targets a fragment of first-order logic, and is often based on the integration of external automated theorem provers, such as first-order provers, satisfiability (SAT) provers, or satisfiability modulo theories (SMT) provers. In this case, the untrusted output of these external provers is used to guide the (re)construction of a formal proof. For instance, *hammers* [14], as well as SMTCoq [15] fall in this category. But interactive provers and automated provers usually do not speak the same logic: most interactive theorem provers implement a flavor of higher-order logic with inductive types, a strict superset of the fragments of first-order logic handled by automated theorem provers. As a consequence, hammers typically include a translation heuristic from the logic of the interactive prover to that of the automated prover, coupled with a formal proof reconstruction mechanism. For instance, a general encoding of the Calculus of Inductive Constructions into a target dialect of first order logic [11] grounds the implementation of a hammer for the Coq proof assistant.

Structure. The paper is organized as follows. Section 2 motivates this work by illustrating the zoo of tactics available for automated reasoning in the Coq prover, and their limits. Section 3 presents our main contribution: a collection of atomic, small-scale transformations for pre-processing goals. For instance, these transformations can explicity axiomatize inductive data types or ease theory-specific automation. Section 4 validates how such small-scale transformations, or a combination thereof, do enhance various automation backends. Section 5 provides some concluding remarks.

The source code of the examples and the Sniper plugin can be found at https://github.com/smtcoq/sniper/releases/tag/cpp23. In particular, it contains a file `examples/paper_examples.v` which presents all the examples of this paper, in the same order; it is designed to be executed throughout the reading of the paper. A `README.md` explains how to build and execute the code. The source code of the Trakt plugin can be found at https://github.com/ecranceMERCE/trakt/releases/tag/1.2%2B8.13.

2 Context and motivating examples

Technical yet uninteresting proof steps are the daily bread of program verification. Fortunately, many elementary statements like the following fact:

```
Lemma length_rev_app : forall B (l l' : list B),
  length (rev (l ++ l')) = length l + length l'.
```

about the length of a reversed appending of two lists are easily solved by modern automated provers. The corresponding *formal* proof steps can in turn be automated, *e.g.*, using *hammers*, a powerful architecture for connecting external automated theorem provers with formal interactive proof environments. The CoqHammer [12] plugin equips Coq with an instance of hammer, inspired by the Isabelle/HOL pioneering instance, but adapted to CIC. This plugin provides a `hammer` tactic, which combines heuristics with calls to external provers for first-order logic, so as to obtain hopefully sufficient hints, including relevant lemmas from the current context, for proving the goal. The actual formal proof is then reconstructed from these hints thanks to variants of the `sauto` tactic and `hammer` outputs a corresponding robust and oracle-independent proof script. For instance, it can produce the script:

```
Proof. scongruence use: app_length, rev_length. Qed.
```

for proving the `length_rev_app` example, using the `app_length` and `rev_length` auxiliary lemmas from the loaded standard library about lists. Yet, as of version 1.3.2, CoqHammer is not designed to exploit any theory-specific reasoning, and thus cannot prove this slight variant, where `(b :: l')` replaces `l'`:

```
Lemma length_rev_app_cons :
  forall B (l l' : list B) (b : B),
    length (rev (l ++ (b::l'))) =
    (length l) + (length l') + 1.
```

because it lacks arithmetical features. In this case, users may resort to the SMTCoq plugin [15], which implements a certificate checker for proof witnesses output by SMT solvers. The latter automated provers are indeed tailored for finding proofs combining propositional reasoning, congruence and theory-specific decision procedures, *e.g.*, for linear arithmetic. However, none of CoqHammer or SMTCoq can in general reason by case analysis or induction. A variant of the `sauto` tactic can prove the following fact about appending though:

```
Lemma app_nilI : forall B (l l' : list B),
  l ++ l' = [] -> l = [] /\ l' = [].
```

But neither CoqHammer nor SMTCoq can prove the following variant, where the first list is reversed:

```
Lemma app_nil_rev : forall B (l l' : list B),
  (rev l) ++ l' = [] -> l = [] /\ l' = [].
```

The SMTCoq plugin can be used to prove properties of linear integer arithmetic, but only when they are stated using the type `Z` of integers from Coq's standard library:

```
Lemma eZ : forall (z : Z), z >= 0 -> z < 1 -> z = 0.
```

Up to version 2.0, SMTCoq is however clueless about any alternative instance of integer arithmetic, *e.g.*, the type `int` of unary integers included in the Mathematical Components (or MathComp) library [21]:

```
Lemma eint : forall (z : int), z >= 0 -> z < 1 -> z = 0.
```

Fortunately, Coq distributes the `lia` tactic [4], specialized to linear integer arithmetic, which can actually also prove lemmas such as `eZ`. Moreover, `lia` can be customized to a user-defined instance of arithmetic thanks to the `zify` dedicated pre-processing [5]. Once correctly configured [28] for type `int`, `lia` is equally powerful on type `Z` or type `int` and proves both `eZ` and `eint`. However, as powerful as it may be on integer linear arithmetic, the `lia` tactic is by nature unaware of the theory of equality. Hence, although it can prove equality `eintC`, it is unable to prove the variant `cong_eintC`, because the latter involves a congruence with the `(_ :: nil)` operation, alien to the theory of linear integer arithmetic:

```
Lemma eintC : forall (z : int), z + 1 = 1 + z.
Lemma congr_eintC : forall (z : int),
  (z + 1) :: nil = (1 + z) :: nil.
```

Proving the property expressed by `congr_eintC` requires *combining* different theories, in this case integer arithmetic and the theory of equality, as SMT solvers do. Yet, in this case as well, the `SMTCoq` plugin cannot help, because the statement of this fact is phrased using type `int` instead of `Z`. The recent `itauto` SAT solver [6], implemented in Coq, provides an alternate take on formally verified satisfiability modulo theory, and organizes the cooperation between the independent tactics `lia`, for integer arithmetic, and `congruence`, for equality. As a consequence, the `smt` tactic built on top of `itauto` can benefit from `lia`'s pre-processing facilities. For instance, as soon as `lia`'s pre-processing is correctly configured for type `int`, the `smt` tactic is able to prove lemma `congr_eintC`. However, `lia`'s pre-processing facilities are not known to the rest of the SMT decision procedure. Thus, although the following goal is solved by the latter `smt` tactic:

```
Lemma eintCb : forall (z : int), (z + 1 == 1 + z) = true.
```

because `lia` has been informed of the Boolean equality test `(_ == _)` available on type `int`, the same tactic fails on the `congr_eintCb` variant, featuring an uninterpreted symbol:

```
Lemma congr_eintCb : forall (f : int -> int) (z : int),
  (f (z + 1) == f (1 + z)) = true.
```

As it turns out, although a variety of tactics implementing automated reasoning is available to the users of the Coq proof assistant, finding the appropriate weapon for attacking a given goal remains challenging. It is often quite difficult to anticipate the exact competence of tactics based on first-order automated reasoning, and to interpret failure. As a consequence, large-scale formalization endeavors may end up developing their own specific automation tools, like the `list_solve` tactic in the Verified Software Toolchain [2][1, Chapters 64, 65], for automating reasoning about lists and arithmetic, which makes the number of available tactics multiply even more, often redundantly.

3 Pre-Processing components

3.1 The role of small-scale transformations

The pitfalls illustrated in § 2 pertain, generally, to the distance between the standards of automated reasoning and the practice of interactive theorem proving. These pitfalls are probably even more acute in provers based on dependent type theory, and particularly so in Coq. For instance, in a prover like Coq, or Agda, or Lean, decidable theories may live in an ambient constructive logic, and excluded middle for the corresponding formulas then follows from this decidability result, rather than from a global axiom. Moreover, Coq in particular has a decentralized ecosystem of external libraries, taking benefit from the versatility of the specification language, and thus featuring different data structures, automation tools, etc. Importantly, modern Coq libraries often make use of *typeclass mechanisms* [20, 31] for defining mathematical notations, which creates formulas with even more syntactically different, but convertible, variants of a same constant, *e.g.*, the addition on integers. And all these variants should nonetheless be understood as the same symbol for automation to succeed.

As these issues apply to any general-purpose automation plugin, the corresponding solutions should remain as *independent* as possible from the targeted backend. What we refer to as *pre-processing* is the process which transforms a Coq goal $C \vdash G$, with its local context C, into a new goal $C' \vdash G'$ easier to handle for an automation backend[1]. We expect it to be *proof-producing*, *i.e.*, to produce not only $C' \vdash G'$ but also a Coq proof that the new goal entails the former one, to be checked by Coq's kernel.

Pre-processing should be *predictable*, in the sense that it should not hamper the interpretation of failures (raised by the backend). A way to preserve predictability is to avoid a silent use of the global context, and sophisticated, heuristic-based proof-search. Pre-processing transformations should also be well-specified, so as to avoid unjustified discrepancies in the behavior of automation, which may resemble unpredictability from the user's standpoint (*e.g.*, `app_nilI` vs `app_nil_rev`, or `eintCb` vs `congr_eintCb`). These precise specifications are a consequence of the *small-scale* and compositional nature of our transformations[2].

Proof-producing, small-scale pre-processing transformations mostly come in two flavors. The first one works by *enriching the local context* C of the initial goal into a larger C', so as to add facts which may help the backend. This is specially useful to interpret the features of the richer logic implemented by the interactive prover, into the weaker fragment understood by the automated prover. The second one works by *translating the goal* G into a possibly different G', so as to align the definitions used in the goal with those used

[1]We omit here the global context, as it is not affected by pre-processing.
[2]The compositionality of our transformations also makes the development process incremental and more robust.

by the backend. This essentially involves casting types and signatures into equivalent ones.

The rest of the section presents a suite of such proof-producing, predictable, small-scale goal transformations, respectively concerned with inductive types, with first-order logic and with symbol interpretation. These transformations are summed-up in **Table 1**. They are clearly not exhaustive, but represent an essential basis of transformations to handle Coq goals, since they deal with everyday constructions. Future work includes extending this suite to handle more and more goals; the compositionality of the approach eases this extension.

The last subsection (3.5) details the meta-programming tools that we have used to develop these transformations.

3.2 Axiomatizing inductive types

All the transformations of this subsection add statements in the proof context. As they should be applied to several terms, they come in two versions:

1. the *elementary* one taking a term as parameter and applying the transformation to it;
2. the one which scans the proof context and calls 1. on all terms of the suitable form. We call it the *context-handling* tactic, but we should emphasize the fact that it acts only on the local context and not on the global one.

Inversion principle of inductive relations. An inductive relation is a Coq inductive type whose codomain is in `Prop`. Let us consider an inductive relation `R` with arguments `x1 ... xn`. In user-written Coq proofs, the tactic `inversion H` is used to retrieve how a hypothesis `H` of type `R x1 ... xn` could have been obtained. This is a consequence of the semantics of the inductive type `R`. But some automation tactics will not perform `inversion` or will not find how to use inversion lemmas properly. In addition, intuitionistic external tools may need information about the symbol `R` in order to use it effectively. This is the reason why we implemented a tactic which generates and proves these inversion lemmas. As an example, let us consider the simple relation `add`, which holds between three natural numbers when the third argument is the sum of the first one and the second one:

```
Inductive add : nat -> nat -> nat -> Prop :=
  | add0 : forall n, add 0 n n
  | addS : forall n m k, add n m k -> add (S n) m (S k).
```

After calling the elementary tactic on `add`, this new statement is added to the local context:

```
forall (n m k : nat), add n m k <->
  (exists (n' : nat), n = 0 /\ m = n' /\ k = n') \/
  (exists (n' m' k' : nat),
    add n' m' k' /\ n = S n' /\ m = m' /\ k = S k')
```

The context-handling tactic generates this principle for all inductive relations encountered in the local context and in the goal, and it is called `inv_principle_all`. It is used in example 4.3.

Interpretation of algebraic data types. Algebraic data types are inductive types with possibly prenex polymorphism (parameters) and no other type dependency (*e.g.*, indices). This transformation takes an inductive type T, generates three families of statements and proves them:

- D_T: the direct images of the constructors of T are *pairwise disjoint* (no-confusion property);
- I_T: the constructors of T are *injective*;
- G_T: each term of type T is *generated* by one of the constructors (generation principle).

These three statements axiomatize T in a first-order logic with prenex polymorphism.

Example 3.1. In order to illustrate the transformation, we consider the case of the `list` inductive type and we obtain[3]:

```
D_list : forall A (l : list A) (x : A), [] <> x :: l.
I_list : forall A (l l' : list A) (x x' : A),
  x :: l = x' :: l' -> x = x' /\ l = l'.
G_list : forall A (l : list A),
  l = [] \/ exists (x : A) (l' : list A), l = x :: l'.
```

Without these statements, many external backends would consider `list` as an uninterpreted data type and, for instance, would not be able to perform case analysis on a list.

Generation statement for inductive types. This transformation provides an alternative generation statement for inductive types, avoiding existential quantifiers, as it can be an obstacle to some automated backends. To achieve this, the idea is to introduce new definitions in the local context for *projections*. In general, the transformation introduces one such projection per argument of each constructor of a given inductive type. Then, projections are used to describe how the terms of the inductive type can be generated from its constructors. This is best illustrated with an example, such as (once again) the type `list`. When given a type `A`, the transformation `get_projs` generates and proves this statement:

```
forall (l : list A) (a : A),
  l = [] \/ l = proj21 A a l :: proj22 A [] l
```

Suppose that our inductive type features n constructors $C_1, \ldots, C_n$ and each constructor C_i has k_i arguments (excluding parameters). The function for the i-th constructor, and the j-th argument (with $j < k_i$), performs a pattern matching on the inductive term. If this term corresponds to the i-th constructor, the function $\texttt{proj}_{i,j}$ returns its j-th projection, *i.e.*, $\texttt{proj}_{i,j}\ (C_i\ \ldots\ x_j\ \ldots) = x_j$. Otherwise, it returns a default term of the expected type, *i.e.*, $\texttt{proj}_{i,j}\ (C_{i'}\ \ldots) = \texttt{default}$. This default term may be found automatically by an auxiliary tactic, or left as a subgoal to the user.

Thus, $k_1 + \ldots + k_n$ projections are generated. In the case of `list`, we have:

[3] $G_{\texttt{list}}$ can be treated slighly differently, see in 3.2.

Category	Specification	Tactic name (when relevant for the article)
Inductive types (3.2)	Inversion principle	`inv_principle_all`
	Algebraic data types	-
	Generation statement	`get_gen_statement_for_variables_in_context`
	Pattern matching	-
Going first-order (3.3)	Equalities	-
	Monomorphization	-
Symbols (3.4)	Constants and fixpoints (3.4.1)	-
	Types and logical connectives (3.4.2)	`trakt` • •

Table 1. List of transformations

```
proj21 := fun (A : Type) (default : A) (l : list A) =>
  match l with
  | [] => default
  | x :: xs => x
  end
proj22 := fun (A : Type) (default : list A) (l : list A) =>
  match l with
  | [] => default
  | x :: xs => xs
  end
```

These projections are uninterpreted symbols for the external solver, but they are defined in Coq in order to prove the correctness of the application of the transformation. An automated backend can perform the case disjunction over a term once the statement (such as the one above) has been generated and *without* knowing the definitions of the projections.

To sum up, the elementary transformation works in three steps:

1. It generates the projections and poses them in the local context;
2. It generates and proves a first generation statement which abstracts over the default terms;
3. Whenever it is possible, it finds an inhabitant for the considered inductive type and generates a statement with no quantification over default terms.

The context-handling one comes in two versions : either it produces and proves the generation statement for all algebraic datatypes `I` such that there is a statement `x : I` in the local context with `x` a variable, or it specializes the statement for each such variables, generating as many corresponding instances. In this case, we eliminate the first universal quantifier and this can help some automated backends, as in example 4.4, and the tactic is called `get_gen_statements_for_variables_in_context`.

Elimination of pattern matching. Whenever a hypothesis contains pattern matching, this transformation states and proves one statement for each pattern. For instance, if we have a hypothesis stating the definition of the `nth_default` function from Coq's standard library:

```
forall A d l n, @nth_default A d l n =
  match nth_error l n with
  | Some x => x
  | None => d
  end
```

it is replaced with the following two hypotheses:

```
forall A d l n,
  nth_error l n = Some x -> nth_default d l n = x.
forall A d l n,
  nth_error l n = None -> nth_default d l n = d.
```

3.3 Going first order

Some tactics of our pre-processing tool are designed to transform or eliminate constructions or features of CIC that an automated solver may not be able to interpret. As we want to be small-scale, such transformations should deal with one particular aspect of Coq logic at a time. We currently have two transformations, to deal with higher-order equalities and prenex polymorphism.

Higher-order equalities. Whenever a hypothesis in the local context of the proof is of the form `f = g`, this transformation replaces it by quantifying over the arguments:

```
forall x1 ... xn, f x1 ... xn = g x1 ... xn
```

This is particularly helpful to clarify a hypothesis which recovers the definition of a function (as will be presented in § 3.4.1). For instance, from an hypothesis which simply expands the definition of the `hd_default` function:

```
hd_default = fun (A : Type) (l :list A) (default : A) =>
  match l with
  | [] => default
  | x :: _ => x
  end
```

the tactic generates and proves:

```
forall (A : Type) (l : list A) (default : A),
  @hd_default A l default =
    match l with
    | [] => default
    | x :: _ => x
    end
```

This transformation weakens the hypothesis in a Coq point of view, but goals whose proof require the higher-order equality would need further transformations to be discharged to backend automation engines based on first-order logic anyway.

Monomorphization. Most automated provers do not handle polymorphism at all. This is a problem as Coq lemmas are often polymorphic and used later on monomorphic instances. For this reason, we implemented a tactic which instantiates all the polymorphic hypotheses given to the tactic and present in the local context with chosen ground types.

The chosen instances depend on each polymorphic hypothesis, using the following heuristic. Suppose that the hypothesis `H` quantifies over n type variables $A_1, \ldots, A_n$, and `H` contains as a subterm an applied inductive of the form $I\ B_{I_1}\ \ldots\ B_{I_k}$ with codomain `Type` and with k parameters of type `Type`. Then for each B_{I_i} being among the $A_1, \ldots, A_n$ (say that $B_{I_i} = A_j$) we search in the goal or in the other (monomorphic) hypotheses for a subterm of the form $I\ t_{I_1}\ \ldots\ t_{I_k}$ or $C\ t_{I_1}\ \ldots\ t_{I_k}$ if C is a constructor of I. The i-th argument of I which is t_{I_i} will be an instance for the variable A_j. We scan all the monomorphic hypotheses and the goal in the same way to find all the instances.

Example 3.2. If we consider the following proof context:

```
H : forall (A B : Type) (x1 x2 : A) (y1 y2 : B),
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
____________________________________________________
forall (x1 x2 : option Z) (y1 y2 : list unit),
  (x1, y1) = (x2, y2) -> x1 = x2 /\ y1 = y2
```

the hypothesis contains one type that takes polymorphic parameters: `*` (the non dependent product type). Thus, our tactic should look at how the parameters of `*` are instantiated in the other hypotheses or in the goal. Here, there is no other hypotheses so we look at the goal. `A` can only be instantiated by `option Z` and `B` by `list unit`.

We also use the version of the tactic implemented in [7], which is more exhaustive and faster when the context contains a reasonable number of potential instances (two or three) but can be slower when the instances are numerous (exponential in the number of instances). Bobot and Paskevich proposed a complete transformation to encompass polymorphism [8]. We may also implement it in next versions, but found in practice our implementation of monomorphization to be efficient enough, and it has the advantage of avoiding obfuscating the goal.

3.4 Giving meaning to symbols

The very rich type system of Coq allows representing theory-specific values, operations, and predicates with a wide range of data structures. On the contrary, theory-based automated provers often associate the signatures of the theories they can process with a few data structures along with the related values and operations (called *symbols*), restricting the set of goals that can be mapped without loss to input statements for the automated provers. In this subsection, we introduce two pre-processing transformations aiming to bridge this gap, *i.e.*, make automation tactics available to Coq users regardless of the representation they use for theory-specific data in their proofs.

3.4.1 Definitions of functions and constants.

Expanding constants. This simple transformation scans the local context and the goal and adds the definitions of all the functions or constants it encounters. Indeed, one needs to let automated theorem provers have access to the meaning of Coq symbols in order to reason about it. For instance, if the term considered is the ternary relation `in_int`, defined in Coq's standard library `Arith` and stating that the third integer is the interval defined by the first one (included) and the second one (excluded), this tactic adds to the local context a lemma expanding the definition as an equality:

```
in_int = fun p q r => p <= r /\ r < q.
```

This tactic is modular in the sense that it takes as parameters the symbols we do not want to interpret: for instance, we do not want the inductive definition of addition to be unfolded in the above fashion, because most of the backends (*e.g.*, SMT solvers) know about arithmetic.

Anonymous fixpoints. The previous transformation can introduce an anonymous fixpoint when the constant is recursive. Let us consider the `length` function. The transformation states and proves:

```
length = (fix length_anon := fun (A : Type) (l : list A) =>
  match l with
  | [] => 0
  | _ :: l' => S (length_anon l')
  end)
```

For this reason, another transformation replaces the anonymous function `length_anon` with its definition (`length`). Once the higher-order equality is eliminated thanks to the transformation in § 3.3, the hypothesis is finally transformed into:

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

3.4.2 Exploiting the notion of equivalence.

The next transformation related to symbols is `Trakt`, a general goal-rewriting tool powered by user declarations. The plugin exports commands for the user to declare *translation tuples* (*i.e.*, proved associations between source and target symbols), as well as a tactic harnessing these translation tuples during a traversal of the current goal to rewrite it into a pre-processed

goal. The modifications applied to the goal are casting theory-specific values into a designated target type for the theory, and translating logic (connectives, predicates, and relations) from `Prop` to `bool` or the other way around. The tactic is certifying, therefore it does not leave proof obligations, and an automation tool can be called right after pre-processing.

In this presentation of `Trakt`, we shall illustrate the tool used in association with an SMT-like prover with support for arithmetic, logic, and uninterpreted symbols. Indeed, this combination of theories being often found in day-to-day `Coq` proofs, it is a realistic example.

Equivalent types. To translate values in a goal from a type T to a type T', the user needs to declare both types as *equivalent types* in `Trakt`. The required declaration for a type equivalence is a translation tuple $(T, T', e, \bar{e}, \mathrm{id}_1, \mathrm{id}_2)$, where T is a source type, T' is a target type, $e : T \to T'$ and $\bar{e} : T' \to T$ are embedding functions (*i.e.*, explicit casts) in both ways, and id_1 and id_2 are proofs that both of their compositions are identities.

Thanks to this declaration, we can embed every value living in T, or any simple functional type containing T (*e.g.*, $T \to$ `bool`), into the corresponding target type (*e.g.*, T into T', and $T \to$ `bool` into $T' \to$ `bool`). In particular, `Trakt` is able to process uninterpreted symbols and universally quantified variables in a type being the source in an equivalence declaration.

Example 3.3. Consider the following goal:

```
forall (P : int -> Prop) (x : int), P x <-> P x
```

It contains an uninterpreted predicate `P` and two quantifiers. After the declaration of an equivalence between `int` to `Z`, `Trakt` can pre-process this goal into the one below:

```
forall (P' : Z -> Prop) (x' : Z), P' x' <-> P' x'
```

To complete proofs based on a theory, we also need to recognize the *signature* of this theory. That is, we must be able to translate various operators and values into the ones that the proof automation tool run by the user after `Trakt` is able to process, so that they do not remain uninterpreted, while remaining general and independent from the theory. These operators and values can be declared as *symbols* with a translation tuple (s, s', p), where s and s' are two symbols, and p is an embedding property:

$$\frac{\begin{array}{c} s : T_1 \to \cdots \to T_n \to T_o \\ s' : T_1' \to \cdots \to T_n' \to T_o' \end{array}}{\begin{array}{c} p : \forall (t_1 : T_1)\ \cdots\ (t_n : T_n), \\ e_o^?\ (s\ t_1\ \cdots\ t_n) = s'\ (e_1^?\ t_1)\ \cdots\ (e_n^?\ t_n) \end{array}}$$

Here, $e_i^?$ denotes an optional embedding function from T_i to T_i' (provided that the user declared the equivalence before declaring the symbol), present only if $T_i' \neq T_i$. This embedding property allows replacing the symbols with their counterparts, introducing or moving embedding functions to preserve typing.

Example 3.4. Consider the following goal:

```
forall (f : int -> int -> int) (x y : int),
  f x (y + 0) = f (x + 0) y
```

It features universal quantifiers and an uninterpreted function, as in Example 3.3, but also addition and zero on the `int` type. Both symbols can be mapped to their counterparts in `Z` after proving their embedding properties[4]:

```
Lemma add_embedding_property : forall (x y : int),
  Z_of_int (x + y) = (Z_of_int x + Z_of_int y)%Z.
Lemma zero_embedding_property : Z_of_int 0 = 0%Z.
```

Once all the declarations are made, `Trakt` can pre-process the goal into the one below:

```
forall (f' : Z -> Z -> Z) (x' y' : Z),
  f' x' (y' + 0)%Z = f' (x' + 0)%Z y'
```

Logic. In an SMT-like goal, theory-based subterms are logical atoms linked together with connectors, equalities, and various other n-ary predicates. In order to fully process these goals, in addition to theory-based pre-processing, `Trakt` is also able to translate these logical values into their Boolean equivalents, and *vice versa*. They can be declared as *relations* by providing a translation tuple (R, R', p), where R and R' are the two associated relations, and p is a proof of equivalence:

$$\frac{\begin{array}{c} R : T_1 \to \cdots \to T_n \to L \\ R' : T_1' \to \cdots \to T_n' \to L' \end{array}}{\begin{array}{c} p : \forall (t_1 : T_1)\ \cdots\ (t_n : T_n), \\ R\ t_1\ \cdots\ t_n \sim_{L,L'} R'\ (e_1^?\ t_1)\ \cdots\ (e_n^?\ t_n) \end{array}}$$

L	L'	$\sim_{L,L'}$
`bool`	`bool`	$\lambda b.\ \lambda b'.\ b = b'$
`Prop`	`bool`	$\lambda P.\ \lambda b.\ P \leftrightarrow b = \mathbf{true}$
`Prop`	`Prop`	$\lambda P.\ \lambda Q.\ P \leftrightarrow Q$

Here, L and L' are logical types (*i.e.*, either `Prop` or `bool`) and $\sim_{L,L'}$ is a way to express equivalence depending on these logical types. $e_i^?$ denotes optional embedding functions, as used above for symbols.

Example 3.5. Consider the following goal:

```
forall (f : int -> int) (x : int), f x + 0 = f x
```

In addition to the features shown in Examples 3.3 and 3.4, this one features equality on `int`, that we might want to turn into a Boolean equality on `Z`. The proof of equivalence in that case is the following:

```
Lemma eq_int_equivalence_property : forall (x y : int),
  x = y <-> (Z_of_int x =? Z_of_int y)%Z = true.
```

Once the declaration is made, `Trakt` can pre-process the goal into the one below:

[4]The scope `Z_scope` allows using on `Z` the same notations for addition, zero, etc., as for `nat`. We disambiguate by expliciting the scope with a postfix `%Z`.

```
forall (f' : Z -> Z) (x' : Z),
  (f' x' + 0 =? f' x')%Z = true
```

Beyond type equivalence. Trakt also allows declaring *partial* embeddings (*i.e.*, non-surjective embeddings), where $e \circ \bar{e}$ is not always an identity. The declaration for type equivalences can be enriched with an alternative tuple $(T, T', e, \bar{e}, C, p_C, \mathrm{id}_1, \mathrm{id}_{2C})$, where C is a restricting predicate on the embedded values (which we call an *embedding condition*), p_C is a proof that it is true on every embedded value, and id_{2C} is id_2 restricted to the condition C. When an embedding is partial, every embedding function inserted also adds a condition to the output formula, thus modifying the structure of the goal.

Example 3.6. Consider the goal of Example 3.5, with `int` replaced with `nat`:

```
forall (f : nat -> nat) (n : nat), f n + 0 = f n
```

Here are the lemmas the user needs to prove to declare an embedding from `nat` to `Z`:

```
Lemma pC_nat : forall (n : nat), (0 <= Z.of_nat n)%Z.
Lemma id2C_nat : forall (z : Z),
  (0 <= z)%Z -> Z.of_nat (Z.to_nat z) = z.
```

Once all the declarations are made, Trakt can be run to pre-process the goal into the one below (assuming the logical target is `Prop`):

```
forall (f' : Z -> Z),
  ((forall (x : Z), 0 <= x -> 0 <= f x) ->
    forall (n' : Z), 0 <= n' -> f' n' + 0 = f' n')%Z
```

As we can see, hypotheses were added right after the quantifiers for `f'` and `n'`, to restrict their domain so that they match the original quantifiers in `nat`.

Knowledge database and user API. Before making proofs, the user can communicate information to Trakt through four Coq commands, one for each kind of information declared: integer types, relations, symbols, and terms that can trigger conversion. Each of these commands is associated to a Coq-Elpi database with a predicate, and every call to the command adds an instance of the associated predicate to the database. This allows the user to statically fill the database and then freely call the `trakt` tactic that will harness this added knowledge by performing lookups at runtime.

Let us show a simplified syntax of the four available commands. Integer types are declared through one of the following commands:

```
Trakt Add Embedding T Z e e' id1 id2.
Trakt Add Embedding T Z e e' id1 id2C pC.
```

The variable names copy the ones above, except for `e'` to replace $\bar{e}$. Notice that the embedding condition is missing. It is due to the fact that this type can actually be inferred from p_C or id_{2C}. This kind of simplification is performed everytime it is possible, to relieve the user from useless repetitions.

For instance, using the lemmas stated in Example 3.6, we can declare an embedding from `nat` to `Z` with this command:

```
Trakt Add Embedding
  nat Z Z.of_nat Z.to_nat id1_nat id2C_nat pC_nat.
```

Relations and symbols follow a similar model (relations need the arity to be specified because inference is more complex). The user may also add terms that will explicitely trigger conversion, in order to avoid being blocked by redefinitions of constant.

```
Trakt Add Symbol s s' p.
Trakt Add Relation n R R' p.
Trakt Add Conversion t.
```

3.5 Meta-programming

Coq offers various approaches to *meta-programming*, that is, to implement programs operating on the syntax of arbitrary Coq terms. In addition to Ltac [35], the default tactic language available in Coq, used to glue the various transformations together, the present contribution combines two such meta-languages, which offer in particular different levels of control on the syntax for terms: MetaCoq and Coq-Elpi.

3.5.1 MetaCoq. MetaCoq [30] is a plugin which provides a quoted syntax of Coq terms, defined as a Coq inductive type. This plugin also provides a tactic `quote_term`, which turns a Gallina term into its quoted counterpart, and an analogous anti-quotation tactic. MetaCoq is useful for performing a very fine-grained analysis on the syntax of Coq terms and provides information about the global environment: we use it to create new statements from a Coq term (such as in 3.2). The technicalities of de Bruijn indices and the lack of pretty-printing or notation tools sometimes limit its usability.

3.5.2 Coq-Elpi. To build a tool able to perform the Trakt transformation presented in 3.4.2, two technical challenges arise: term traversal (*i.e.*, recursively inspecting a Coq term and building a new term at the same time) and user knowledge management (*i.e.*, a way to declare translation tuples, a database to store them, and a mechanism to perform lookups during pre-processing). For the implementation, we used Coq-Elpi [34], an implementation of λProlog (called Elpi) coupled with an API that connects it to Coq internals, allowing to make new declarations, commands, and tactics. Let us now give a few implementation details about the plugin and explain how this meta-language meets our needs.

Term traversal and reconstruction. In Trakt, the pre-processing algorithm takes the shape of a Coq-Elpi predicate with various cases according to the shape of the inspected term. These cases are based on an inductive type in the meta-language representing Coq terms in *Higher-Order Abstract Syntax* [25] (HOAS), meaning that terms under binders are represented with meta-functions.

For instance, universal quantifiers are represented with `prod N T F`, where `N` is the name of the bound variable, `T` is its type, and `F` is a function (*i.e.*, of type `term -> term` at the meta level) encoding the quantified term. To traverse this term, we introduce a locally bound variable `x` (called a *universal constant*) thanks to the `pi` keyword, then we can recursively traverse `F x`. In particular, this way of binding Coq variables directly frees us from having to handle de Bruijn indices in the meta-program. In addition, when using a universal constant `x`, we express the outputs of the subsequent calls as functions of `x`, ensuring by design that no output term will make the variable escape its scope. Finally, thanks to Prolog-like implication, it is possible to assume information about universal constants, adding this information to the scope of subsequent calls, thus eliminating the need to maintain any context in the pre-processing predicate.

To show all these features in a real use case, let us give a simplified snippet taken from the code of Trakt. Consider a predicate `preprocess` which takes the input goal and outputs the pre-processed goal along with the certification of the goal substitution. The base cases of the predicate should correspond to transformation tuples, but here is the case for universal quantifiers:

```
preprocess (prod N T F) Out Proof :- !,
  pi x\ decl x _ T => preprocess (F x) (F' x) (ProofF x)
  % ...
```

As explained above, we introduce a universal constant `x` to make a recursive call on `F x`, and by using the implication, we introduce `decl x _ T` into the scope of the recursive call, thus assuming that the type of `x` is `T`. The proof on the subterm is represented by a meta-function `ProofF`, so it does not contain `x`. The rest of the code for this case of the predicate focuses on lifting this proof to a proof on the quantified term.

Coq-Elpi also features a quotation and anti-quotation mechanism, to ease the expression of the output term and the corresponding Coq proof. Indeed, it is possible to use Coq syntax to express a term by surrounding the term with brackets, and it is possible to mention Elpi terms thanks to the `lp:` prefix. For example, `{{ lp:A -> lp:B }}` denotes a Coq term embedded in Elpi. It is the type of Coq functions from a type expressed in the Elpi variable `A` to another expressed in the variable `B`.

Management of user knowledge. The Coq-Elpi meta-language suits perfectly the way the Trakt plugin works. Indeed, it offers a way to declare databases and fill them with terms extracted from commands typed by the user, and have a tactic use these databases afterwards. Trakt requires the user to register embeddings, symbols and relations through the available commands, and the `trakt` tactic performs lookups on this database to pre-process the goal. To our knowledge, the other common meta-languages for Coq do not allow for such a direct implementation of this information accumulation procedure, and the related user commands.

4 From single to composed pre-processing and applications

This section revisits the examples discussed in § 3 and illustrates the benefits brought by the various tactics presented in § 3, to various automated backends. These tactics can be used in isolation, for a single-component preprocessing, or combined according to a specific strategy. Section 4.2 presents an example of such a combination, designed for the SMTCoq backend.

4.1 Single-component pre-processing

4.1.1 Theories. The association of the Trakt tactic with the `itauto` plugin [6] for satisfiability modulo theory, overcomes the limitations of the latter discussed in § 2.

Example 4.1. Here is an example of goal successfully proved by the association of Trakt and `itauto`, phrased using the `int` type from library MathComp:

```
Goal forall (f : int -> int) (x : int),
  (f (2%:Z * x) <= f (x + x))%R = true.
Proof. trakt Z Prop. smt. Qed.
```

This goal features arithmetic symbols (addition, multiplication), a Boolean comparison relation, an uninterpreted function `f`, and values in the `int` type from MathComp. Notations are moreover generic: `+` (resp. `*`) refers to the law of an instance of commutative group (resp. the product of a ring) and `<=` is a generic notation for an order relation. Note how the translation is configured so as to target the `Prop` sort, so that the `itauto` solver is eventually able to prove the goal automatically.

Generic notations like the ones of example 4.1 are implemented using the form of *ad hoc* polymorphism available in Coq via a combination of advanced inference, canonical structures [20] or typeclasses [31], and notations. Dealing with this overloading requires some care in the implementation. Indeed, by default, the right case of the pre-processing predicate is selected by performing a Prolog-like unification between the input term and the pattern in the head of the clause (*e.g.*, `prod N T F` in the case above). But in this case, purely syntactic matching does not suffices.

Example 4.2. Consider again Example 3.5, where notations `+` and `0` are instances of notations based on *ad hoc* polymorphism (enabled in the `ring_scope` notation scope denoted by the postfix `%R`):

```
forall (f : int -> int) (x : int), (f x + 0)%R = f x
```

Term `f x + 0` actually unfolds to the term with holes

```
@GRing.add _ (f x) (@GRing.zero _)
```

In this case, holes are filled by Coq with the canonical instance of ring structure available for `int`, so that in the end `f x + 0` is elaborated as:

```
@GRing.add int_ZmodType (f x) (@GRing.zero int_ZmodType)
```

As a consequence, in this goal, the addition on type `int` is represented as `(@GRing.add int_ZmodType)`, a term convertible but not syntactically equal to the defined constant

```
addz : int -> int -> int
```

registered as an equivalent to

```
Z.add : Z -> Z -> Z
```

Trakt supports an additional declaration of terms that can trigger Coq conversion, so that the translation algorithm cannot distinguish the goal above from the one in Example 3.5, thus successfully pre-processing goals featuring such projections of instances of structures. As a result, running `trakt Z bool` on this example gives the expected output goal:

```
forall (f' : Z -> Z) (x' : Z), (f' x' + 0 =? f' x')%Z = true
```

4.1.2 Inversion principle.

The first volume of the Software Foundations textbooks [26] uses the following `ev` inductive predicate to illustrate the *inversion* of inductively defined propositions in the eponymous chapter:

```
Inductive ev : nat -> Prop :=
  | ev_0 : ev 0
  | ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

Example 4.3. A manual proof requires performing inversion twice to prove:

```
Lemma SSSSev_ev : forall (n : nat),
  ev (S (S (S (S n)))) -> ev n.
Proof.
  Fail firstorder congruence.
  inv_principle_all; firstorder congruence.
Qed.
```

Notwithstanding the pedagogical merits of this exercise in iterating inversion, one may wish in realistic situations for a script which does not depend on the number of successor symbols `S` involved in the statement. However, the `firstorder congruence` tactic alone is not able to prove this goal, although the latter falls in the fragment of equality logic with uninterpreted functions decided by this tactic. Generating (automatically) the appropriate inversion lemma, about `ev`, is however enough to help the proof-search procedure.

4.1.3 Generation principle.

As Blazy and Leroy pointed out when reporting on their work about memory models for CompCert [18], most of the related Coq developments fall under first-order reasoning and proofs could be better automated. But the incomplete support for inductive types of the available automation tactics for first-order reasoning is often a showstopper. Automating the generation of the appropriate properties of constructors is however often enough to enable automation.

Example 4.4. In CompCert, the type `memory_chunk` is an enumeration indicating the type, size and signedness of the chunk of memory being accessed. It can be translated into an integer by the function `memory_chunk_to_Z`. It associates to the i-th constructor the integer i. None of the variants of the `sauto` tactic provided by the CoqHammer plugin can prove lemma `memory_chunk_to_Z_eq`, but `sauto` can prove it after suitable principles have been generated.

```
Lemma memory_chunk_to_Z_eq : forall x y,
  x = y <-> memory_chunk_to_Z x = memory_chunk_to_Z y.
Proof.
  Fail sauto.
  get_gen_statement_for_variables_in_context; sauto.
Qed.
```

The same association can prove lemma `app_nilI`, as does the `sauto dep: on` variant of `sauto`:

```
Lemma app_nilI : forall B (l1 l2 : list B),
  l1 ++ l2 = [] -> l1 = [] /\ l2 = [].
  Fail sauto.
  get_gen_statement_for_variables_in_context; sauto.
Qed.
```

4.2 Composite pre-processing

4.2.1 The `snipe` tactic.

The Sniper plugin [7] equips the SMTCoq backend[5] with a pre-processing tactic called `scope`. The general methodology of this plugin is presented in **Figure 1**: the `scope` pre-processing tactic chains small-scale transformations (represented as $T_0 \dots T_6$ in the figure), possibly with various paths depending on the goal and context, then the SMTCoq automated backend is called.

We have significantly improved `scope`, which now incorporates all (but one, specific to intuitionistic logic) of the transformations presented in § 3, and relies on an enhanced strategy for orchestrating their combination. The strategy implemented by this new version of `scope` proceeds as follows. First, Trakt translates each inductive predicate in `Prop` present in the goal into its Boolean fixpoint equivalent, if available in its database. Then, the context is enriched with first-order polymorphic statements about the functions and algebraic data types present in the goal, so as to provide veriT with the needed part of the Coq environment. Then, these statements are instantiated and finally, using Trakt again allows us to target the type `bool` of classical propositions and the type `Z` of relative integers, so as to meet SMTCoq's requirements. Note that the transformation which generates the inversion principles of inductive relations is not used by `scope`, as it is rather designed for backends dealing with intuitionistic logic. This strategy currently freezes an order for the transformations (that is to say a path in the general methodology of **Figure 1**); we live for future work to improve flexibility (see Section 5).

[5] It targets more particularly the `verit` tactic, which calls the external SMT solver veriT [9].

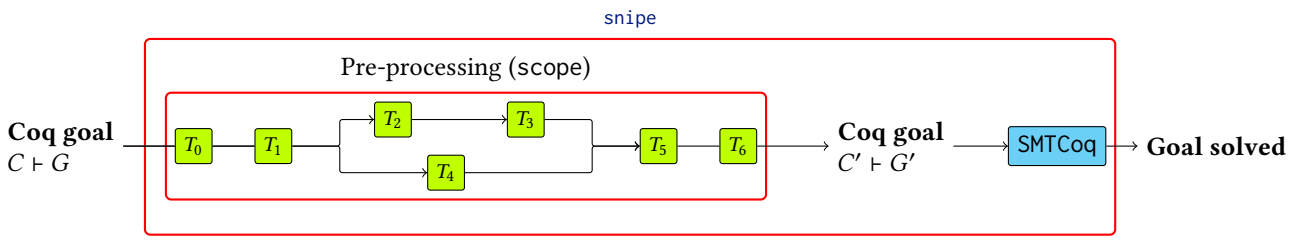

Figure 1. The architecture of the snipe tactic

With this new pre-processing, the snipe tactic automatically solves all the goals of the previous subsection but SSSSev_ev. We now showcase it on two Coq developments: a formalization of interval lists, and a formalization of a λ-calculus based on De Bruijn indices.

4.2.2 Use case of snipe: interval lists.

This use case deals with a formalization in Coq of interval lists [17], representing constraints on integer variables. This example is particularly well-suited for SMT-based automation, as it involves data structures with a decidable theory, and linear arithmetic. This development represents domains by a Coq inductive type:

```
Inductive elt_list :=
  | Nil : elt_list
  | Cons : Z -> Z -> elt_list -> elt_list.
```

where Z is the integer type from the standard library. These intervals come with a well-formedness condition, formalized as an inductive relation, ensuring that the intervals which belong to the list are in ascending order, disjoint and not empty. The first argument of the relation, of type Z, is the lower bound of the interval list.

```
Inductive Inv_elt_list : Z -> elt_list -> Prop :=
  | invNil : forall b, Inv_elt_list b Nil
  | invCons :
    forall (a b j : Z) (q : elt_list),
      j <= a -> a <= b -> Inv_elt_list (b + 2) q ->
      Inv_elt_list j (Cons a b q).
```

It is possible and helpful to write a Boolean equivalent Inv_elt_list_bool of this predicate:

```
Lemma Inv_elt_list_decidable : forall b e,
  Inv_elt_list b e <-> Inv_elt_list_bool b e = true.
```

In particular, this Boolean variant is well-suited for the SMTCoq backend, based on classical logic. At this point, the correspondance between the two equivalent versions of the well-formedness condition can be added to Trakt's database by using the command:

```
Trakt Add Relation
  Inv_elt_list Inv_elt_list_bool Inv_elt_list_decidable.
```

Whenever the trakt tactic targets bool, any goal featuring Inv_elt_list will be translated into a goal featuring its Boolean counterpart Inv_elt_list_bool. Then, the transformations of scope about interpreted symbols can state and prove properties about this Boolean counterpart in an understandable way for a SMT solver, and in particular, for SMTCoq.

Example 4.5. The following monotonicity property:

```
Lemma inv_elt_list_monoton : forall l y z,
  Inv_elt_list y l -> z <= y -> Inv_elt_list z l.
Proof.
  induction l; snipe.
Qed.
```

can be proved by an induction on the list of intervals l, for which sub-cases are solved by the snipe tactic.

The same development introduces the domain of a list of interval, represented as a Coq record:

```
Record t := mk_t
  { domain : elt_list;
    size : Z;
    max : Z;
    min : Z; }.
```

equipped with the following well-formedness condition:

```
Definition Inv_t (d : t) :=
  Inv_elt_list (min d) (domain d) /\
  (min d) = get_min (domain d) min_int /\
  (max d) = process_max (domain d) /\
  (size d) = process_size (domain d).
```

More explicitly, the domain should verify the previous invariant Inv_elt_list, the value corresponding to the field min should be the same as the one computed by the get_min function, and similar conditions apply to the fields size and max. The functions computing the minimum and the maximum of the list of intervals are returning a default value called min_int whenever the domain is empty. The constant min_int is an unconstrained parameter of the formalization.

Notice that the only correct term of type t with an empty domain is the following:

```
Definition empty :=
  {| domain := Nil; size := 0;
     max := min_int; min := min_int |}.
```

After introducing these new definitions, and the Boolean counterpart of the well-formedness condition, snipe is able to prove automatically facts like:

```
Lemma empty_inv : Inv_t empty. Proof. snipe. Qed.

Lemma equiv_empty_Nil : forall d : t,
  Inv_t d -> domain d = Nil <-> d = empty.
Proof. snipe. Qed.
```

4.2.3 Use case of `snipe`: De Bruijn indices.

This example deals with a formalization in Coq of confluence and strong normalization for some λ-calculi [27], based on the MathComp libraries. This formalization involves deeply embeddings of languages with binders, and uses de Bruijn indices to encode bound variables. This guarantees in particular the uniqueness of term representation. The price to pay is the need to prove cumbersome properties on lifting and substitution of variables. Such proofs typically involve a combination of integer arithmetic and propositional reasoning, together with induction with a well-chosen set of generalized variables.

For instance, untyped λ-calculus is defined as:

```
Inductive term : Type :=
  | var of nat
  | app of term & term
  | abs of term.
```

with the lifting and substitution functions being respectively:

```
Fixpoint shift d c t : term :=
  match t with
  | var n => var (if c <= n then n + d else n)
  | app t1 t2 => app (shift d c t1) (shift d c t2)
  | abs t1 => abs (shift d c.+1 t1)
  end.
```

and

```
Notation substitutev ts m n :=
  (shift n 0 (nth (var (m - n - size ts)) ts (m - n))).
Fixpoint substitute n ts t : term :=
  match t with
  | var m => if n <= m then substitutev ts m n else m
  | app t1 t2 =>
    app (substitute n ts t1) (substitute n ts t2)
  | abs t' => abs (substitute n.+1 ts t')
  end.
```

Observe that these definitions make use of the MathComp definitions of addition, subtraction and comparison on natural numbers. By adding them to the database of `Trakt`, an induction on terms followed by the `snipe` tactic is sufficient to prove a number of the aforementioned properties, such as the following[6]:

```
Lemma shift_add d d' c c' t :
c <=? c' -> c' <=? c + d ->
shift d' c' (shift d c t) = shift (d' + d) c t.
(* synonym of: revert d d' c c'; induction t; snipe. *)
Proof. elim: t d d' c c'; snipe. Qed.

Lemma shift_shift_distr d c d' c' t :
c' <=? c ->
shift d' c' (shift d c t) =
                        shift d (d' + c) (shift d' c' t).
(* synonym of: revert d d' c c'; induction t; snipe. *)
Proof. elim: t d d' c c'; snipe. Qed.
```

[6]We display the original scripts, written using the ssreflect tactic language. Comments provide an analogue version in vanilla Coq.

5 Related work and perspectives

The design of user-friendly yet robust automation tools for formal proofs is inherently difficult a problem. For instance, users might not realize that a minor change in a formula might put their problem in an undecidable fragment. As a consequence, some authors like Chlipala advocate a development model for formal libraries in which each new project comes with its own tactic library, with "a different mix of undecidable theories where a different set of heuristics turns out to work well" [10]. The present paper proposes a collection of general-purpose, small-scale, and proof-producing goal transformations, listed in § 3, which can contribute in a compositional and reusable way to such project-specific tactics. In particular, § 4 illustrates how these transformations can be associated with various automated reasoning backends, for applications in various contexts. Targeting more heuristic backends would be just as easy. Because these transformations are small-scale and proof-producing, they do not blur diagnosis in case of failure of the resulting automatic tactic. Because they are small scale and compositional, they can be orchestrated according to different strategies, depending on the project or on the backend. They are independent from other features of hammers, like the ability to select relevant facts from the context, but can greatly help proof reconstruction.

The present work extends and generalizes a previous work [7] that combined some pre-processing tactics with the SMTCoq automation tool. Notably, pre-processing is now completely untied from the backend, which allows for an association with arbitrary tactics, as opposed to just the SMTCoq plugin. Moreover, the pool of pre-processing tactics has been augmented, in particular with the support for translating along declared equivalences of types or symbols [13], and for inductive relations. The support for algebraic data types has also been improved.

The generation of auxiliary principles for inductive types, for helping programs and proofs, automated or manual, is a classic topic [23] and different meta-programming techniques have been studied in particular for the generation of inductive schemes [34].

By contrast, the only work we are aware of about independent, theory-specific pre-processing is the `zify` [5] tactic. For instance, the `mczify` library declares the relevant instances to data structures defined in the MathComp library so as to bring the power of the `lia` tactic, automating linear integer arithmetic, to users of these libraries. As illustrated in § 2, the pre-processing implemented by `zify` is quite specific and

cannot in general traverse uninterpreted symbols. Tools for normalizing coercions and casts [19] can also be seen as pre-processing devices.

Generalizing the latter issue with coercions and casts, different design patterns in the hierarchies of algebraic structures, underlying generic notations, may also limit the usage of the associated decision procedures (e.g., for ring equalities): this problem is discussed and solved by Sakagushi, using an extra layer of reflexion [29].

Univalent foundations are meant to better account for the transfer of properties from one representation of a mathematical concept to another, equivalent one. Homotopy equivalences, at the core of homotopy type theory, make precise the types for which any such property can be transferred. Under the assumption of the univalent axiom, univalent parametricity makes possible to perform this transfer in practice, as partially implemented in the companion prototype to Tanter et al.'s paper [33]. Although illustrating nicely the paper's contributions, the latter prototype is however not usable beyond toy examples. However, a consolidated re-implementation of the same paradigm could provide a more principled implementation of the `trakt` plugin for proofs under the univalence assumption.

Formal proof automation based on the cooperation with a first-order automated reasoning device usually encompasses some notion of pre-processing, although as an indivisible whole. This pre-processing part is then usually tied to the backend(s) used, as, e.g., in the case of `CoqHammer` [12] or of the `F*` [32] programming language. In the case of `F*` however, a tactic language was added to help users target (solely) the SMT backend [22]. In comparison, we use here the `Coq` tactic language and meta-programming tools, and we target various backends.

Backends are willing to evolve. For instance, the efficiency of SMT solvers such as `veriT` and `CVC5` on –certain subsets of– higher-order logic is gaining traction [3]. Our pre-processing approach is well-suited to follow such developments, as small-scale transformations can be easily plugged or unplugged.

Expanding further the pool of small-scale transformations, and their efficiency, is a natural direction of future work. Another direction, may be even more important, is to study more in-depth the appropriate strategies for organizing of the cooperation of small-scale tactics. Ideally, less expertise in meta-programming should be required for implementing the latter strategies. Designing a suitable API would help creating new pre-processing tactics, either by a fine-grained description of the actions, or by triggering automated patterns from an inspection of the goals, or by a combination thereof.

Acknowledgments

The authors would like to thank Kazuhiko Sakaguchi, for his help on the example of Section 4.2.3 and his proofreading of the paper, and Amélie Ledein, for her help on the example of Section 4.2.2.

References

[1] Qinxiang Cao Andrew W. Appel, Lennart Beringer and Josiah Dodds. 2022. *Verifiable C*. https://github.com/PrincetonUniversity/VST/raw/master/doc/VC.pdf.

[2] Andrew W. Appel. 2011. Verified Software Toolchain - (Invited Talk). In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6602)*, Gilles Barthe (Ed.). Springer, 1–17. https://doi.org/10.1007/978-3-642-19718-5_1

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

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

[5] Frédéric Besson. 2017. ppsimpl: a reflexive Coq tactic for canonising goals. In *Coq Workshop on Programming Languages*. https://popl17.sigplan.org/details/main/3/ppsimpl-a-reflexive-Coq-tactic-for-canonising-goals.

[6] Frédéric Besson. 2021. Itauto: An Extensible Intuitionistic SAT Solver. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:18. https://doi.org/10.4230/LIPIcs.ITP.2021.9

[7] Valentin Blot, Louise Dubois de Prisque, Chantal Keller, and Pierre Vial. 2021. General Automation in Coq through Modular Transformations. In *Proceedings Seventh Workshop on Proof eXchange for Theorem Proving, PxTP 2021, Pittsburg, PA, USA, July 11, 2021 (EPTCS, Vol. 336)*, Chantal Keller and Mathias Fleury (Eds.). 24–39. https://doi.org/10.4204/EPTCS.336.3

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

[9] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. 2009. veriT: An Open, Trustable and Efficient SMT-Solver. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5663)*, Renate A. Schmidt (Ed.). Springer, 151–156. https://doi.org/10.1007/978-3-642-02959-2_12

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

[11] Lukasz Czajka. 2018. A Shallow Embedding of Pure Type Systems into First-Order Logic. In *22nd International Conference on Types for Proofs and Programs (TYPES 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 97)*, Silvia Ghilezan, Herman Geuvers, and Jelena Ivetić (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9:1–9:39. https://doi.org/10.4230/LIPIcs.TYPES.2016.9

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

[13] Enzo Crance Denis Cousineau and Assia Mahboubi. 2022. Trakt: a generic pre-processing tactic for theory-based proof automation. (2022). Abstract and talk at the Coq Workshop 2022, https://www.youtube.com/watch?v=dZyylCY1seE.

[14] Martin Desharnais, Petar Vukmirovic, Jasmin Blanchette, and Makarius Wenzel. 2022. Seventeen Provers Under the Hammer. In *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel (LIPIcs, Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:18. https://doi.org/10.4230/LIPIcs.ITP.2022.8

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

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

[17] Amélie Ledein and Catherine Dubois. 2020. FaCiLe en Coq : vérification formelle des listes d'intervalles. In *JFLA'20 : Journées Francophones sur les Langages Applicatifs (Annual Workshop)*. INRIA, INRIA.

[18] Xavier Leroy and Sandrine Blazy. 2008. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. *J. Autom. Reason.* 41, 1 (2008), 1–31. https://doi.org/10.1007/s10817-008-9099-0

[19] Robert Y. Lewis and Paul-Nicolas Madelaine. 2020. Simplifying Casts and Coercions (Extended Abstract). In *Joint Proceedings of the 7th Workshop on Practical Aspects of Automated Reasoning (PAAR) and the 5th Satisfiability Checking and Symbolic Computation Workshop (SC-Square) Workshop, 2020 co-located with the 10th International Joint Conference on Automated Reasoning (IJCAR 2020), Paris, France, June-July, 2020 (Virtual) (CEUR Workshop Proceedings, Vol. 2752)*, Pascal Fontaine, Konstantin Korovin, Ilias S. Kotsireas, Philipp Rümmer, and Sophie Tourret (Eds.). CEUR-WS.org, 53–62. http://ceur-ws.org/Vol-2752/paper4.pdf

[20] Assia Mahboubi and Enrico Tassi. 2013. Canonical Structures for the Working Coq User. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7998)*, Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie (Eds.). Springer, 19–34. https://doi.org/10.1007/978-3-642-39634-2_5

[21] Assia Mahboubi and Enrico Tassi. 2021. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.4457887

[22] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. 2019. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luís Caires (Ed.). Springer, 30–59. https://doi.org/10.1007/978-3-030-17184-1_2

[23] Conor McBride, Healfdene Goguen, and James McKinna. 2004. A Few Constructions on Constructors. In *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 3839)*, Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner (Eds.). Springer, 186–200. https://doi.org/10.1007/11617990_12

[24] Sean McLaughlin and John Harrison. 2005. A Proof-Producing Decision Procedure for Real Arithmetic. In *Automated Deduction - CADE-20, 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3632)*, Robert Nieuwenhuis (Ed.). Springer, 295–314. https://doi.org/10.1007/11532231_22

[25] Dale Miller. 2000. Abstract Syntax for Variable Binders: An Overview. In *Computational Logic - CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1861)*, John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey (Eds.). Springer, 239–253. https://doi.org/10.1007/3-540-44957-4_16

[26] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Logical Foundations*. Electronic textbook. Version 5.5. http://www.cis.upenn.edu/~bcpierce/sf.

[27] Kazuhiko Sakaguchi. 2012-2020. *A Formalization of Typed and Untyped λ-Calculi in Coq and Agda2*. https://github.com/pi8027/lambda-calculus

[28] Kazuhiko Sakaguchi. 2019-2022. *Micromega tactics for Mathematical Components*. https://github.com/math-comp/mczify

[29] Kazuhiko Sakaguchi. 2022. Reflexive Tactics for Algebra, Revisited. In *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel (LIPIcs, Vol. 237)*, June Andronick and Leonardo de Moura (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:22. https://doi.org/10.4230/LIPIcs.ITP.2022.29

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

[31] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 5170)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). Springer, 278–293. https://doi.org/10.1007/978-3-540-71067-7_23

[32] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655

[33] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2021. The Marriage of Univalence and Parametricity. *J. ACM* 68, 1, Article 5 (jan 2021), 44 pages. https://doi.org/10.1145/3429979

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

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

Received 2022-09-21; accepted 2022-11-21