Abstract

The Coq proof-assistant provides automation for various logic fragments. However, there is a lack of generic support for extending those tactics. To augment the proof automation, we propose an extensible reflexive tactic, \texttt{ppsimpl}, aiming at canonising goals so that the scope of existing tactics can be broadened at low cost.

The tactic first injects a type, say \( T \), into a canonical type, say \( CT \) and maps function over \( T \) into their counterpart over \( CT \). This transformation allows type \( T \) to benefit from the automation provided for type \( CT \). The tactic also performs another normalisation step which purpose is to restrict the operators of \( CT \) to those that are known to the automated tactics. This is done by either unfolding function definitions or replacing a function by a (partial) specification.

The extensibility of the \texttt{ppsimpl} tactic is obtained through the type-class mechanism which allows to infer and collect all the necessary proof objects. These instances are processed by a Ltac compiler which automatically generate the reification of terms and instantiate the generic correctness proof.

1. Introduction

The Coq proof-assistant comes with a bestiary of proof-procedures. Each of them being specialised for a particular logic fragment. Among the most popular ones are \texttt{omega} and \texttt{lia} for linear (integer) arithmetic; \texttt{ira} and \texttt{fourier} for linear real arithmetic; \texttt{ring} for solving ring equations; \texttt{field} for rational expressions; \texttt{congruence} for the logic of uninterpreted functions and constructors; \texttt{tauto} and \texttt{rtauto} for propositional logic. Sometimes, those tactics fail to solve goals whereas a casual user would expect them to succeed. Hence, mundane proofs get longer; require a deeper knowledge of libraries and are less robust to changes. There are various reasons that limit in practice the applicability of existing Coq tactics. For instance, little support is provided to deal with the low-level numeric types \texttt{positive} and maps function over \texttt{nat} into another type \( Z \) and \texttt{srsreflect} \cite{Gonthier2008} makes a systematic link between booleans and propositions using so-called small scale reflection. Yet, \texttt{zify} is not easily extensible and \texttt{srsreflect} does not aim at large scale reflection.

In this paper, we describe an extensible reflexive tactic to canonise goals in order to extend the scope of current automation. The transformation is specified with type classes \cite{Sozeau2008} and is thus easily extensible. This specification is then compiled into an efficient reflexive tactic. With support for boolean and integers, the tactic solves automatically the goals of Figure 1. A working tactic can be obtained by cloning the git repository \url{https://scm.gforge.inria.fr/anonscm/git/ppsimpl/ppsimpl.git} and checking out the \texttt{ppsimpl-8.5} branch.

2. The tactic from the user perspective

The \texttt{ppsimpl} tactic is compiling ahead of time class instances that are declared by the user. In the following, we review the different type-classes that are required by the \texttt{ppsimpl} compiler. As we will see, certain instances require non-trivial proof-terms. When this is the case, we make sure that the proof-terms are automatically filled in by the type-class resolution mechanism. We also provide tactics dedicated to certain instances so that remaining proof obligations are meaningful from the user standpoint.

2.1 User declarations

The Coq types the tactic operates on is not fixed. A novel type \( T \) is declared by providing an instance of type \texttt{TypeDecl.t}. Such an instance contains a default value of type \( T \), an equivalence over \( T \) and a predicate that is universally true for the values of \( T \). For instance, if we declare the type \texttt{nat} of Peano integers, the canonical default value is \( 0 \); the equivalence relation is Leibniz equality (=) over \( nat \) and the predicate states that all natural numbers are positive fun \( (x:nat) \Rightarrow x >= 0 \).

Once a type \( T \) is defined, the user specifies how to inject \( T \) into another type \( T' \) using an instance of type \texttt{DeclInj.t T (TypeDecl.t T) T' (TypeDecl.t T')}. Such an instance contains an injection function \( inj : T \rightarrow T' \) and a proof that \( inj \) preserves the relevant equivalence. For instance, to inject

\begin{verbatim}
Goal \forall x y : bool, x && y = true \rightarrow y = true.
Proof. intros x y; ppsimpl; tauto. Qed.
Goal \forall x : nat, (x > 0 -> sqrt x > x -> x = 0).
Proof. intros; ppsimpl; nia. Qed.
\end{verbatim}

Figure 1. \texttt{ppsimpl} at work

these problems, the Ltac tactic \texttt{zify} tries hard to map different numeric types towards \( Z \) and \texttt{srsreflect} \cite{Gonthier2008} makes a systematic link between booleans and propositions using so-called small scale reflection. Yet, \texttt{zify} is not easily extensible and \texttt{srsreflect} does not aim at large scale reflection.

\footnote{Tough undocumented, \texttt{zify} is a quite useful tactic developed by P. Letouzey.}

Copyright © 2022 ACM, Inc. All rights reserved.

Permission to make digital or hard copies of part or all of these works for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of these works owned by others than ACM must be honored.

Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
nat toward Z, the injection function is Z.of_nat and we need to prove that forall x y : nat, x = y <-> Z.of_nat x = Z.of_nat y.

After declaring types, the user declares how to inject a function F of type Ty = T₁ -> . . . -> Tₙ -> T₀ by providing an instance of type Inj.t Ty F. The instance contains a function F' of type Ty' = T₁' -> . . . -> Tₙ' such that T₀' is the injection of type T₀ (i.e., we have an instance of TypeDecl.t Inj.t T₀ T₀'). We would provide the function N.add nat => Z.add (Z.of_nat y) (Z.of_nat x) and that forall x x' y (isTrue x = x + y) -> Z.add y x' = Z.add x y'. The definition of the class Inj.t complicates the injected predicate by the fact that the injection and morphism need to be generic w.r.t. the type Ty. For this purpose, the class definition contains additional information that is responsible for constructing the relevant injection and morphism lemma by induction over the structure of the type Ty = T₁ -> . . . -> Tₙ -> T₀. This object is automatically constructed using type-class resolution using instances that ensure in particular that all the T₀ and T₀' have been declared (there is an instance of TypeDecl.t T₀ and TypeDecl.t T₀').

The user can optionally specify that a function F can be unfolded by declaring an instance of class Unfold.t F. The instance contains a function F' and a proof that the results are equivalent. For example, to define an instance of type Unfold.t Zsucc nat => Z.add (Z.of_nat y) (Z.of_nat x) and that forall x x' y (isTrue x = x + y) -> Z.add y x' = Z.add x y'. The definition of the class Unfold.t is complicated by the fact that it contains the injection and morphism need to be generic w.r.t. the type Ty. For this purpose, the class definition contains additional information that is responsible for constructing the relevant injection and morphism lemma by induction over the structure of the type Ty = T₁ -> . . . -> Tₙ -> T₀. This object is automatically constructed using type-class resolution using instances that ensure in particular that all the T₀ and T₀' have been declared (there is an instance of TypeDecl.t T₀ and TypeDecl.t T₀').

### 2.2 Tactic compilation

Given the above type class declarations, it is possible to program a generic rewrite engine recursively traversing terms and applying the appropriate injection lemma. This tactic would have the flavor of the (z)ify tactic with the advantage of being easily extensible. We do not follow this path but instead how to compile declared class instances to obtain a reflexive tactic. Eventually, the compilation would benefit from an OCAML implementation. Currently, it is implemented in Ltac using various tricks which do not deserve advertising. Anyhow, the compilation requires the definition of several functions and lemmas.

The compilation done, the pp(simpl tactic performs a standard proof by reflection. After the tactic, we use a carefully designed conversion in order to de-reify the goal and get a readable result preserving in particular the original variable names. For the examples of Figure 1, pp(simpl generates the goals of Figure 2. Each original variable is given a pre-condition that is generated from type constraints. Here, when declaring TypeDecl.t bool we stated that a boolean can only be true or false i.e., forall b, isTrue b \ /

\neg isTrue b. When declaring TypeDecl.t nat we stated that natural numbers are positive i.e., forall n, 0 <= n. The rest of the goal is obtained by recursively applying injection functions. In Figure 2, the square-root over nat is injected into the square-root over Z. Other introduced pre-conditions are the specifications of functions F that have an Abstract.t F instance. In that case, the call to F is replaced by a fresh variable. For example, in Figure 2, the square-root of is named e.

### 3. Conclusion

Proof by reflection [Bertot and Castéran, 2004] requires a deep-embedding of a specific logic fragment. In order to import HOL-LIGHT proofs into Coq, Keller and Werner [Keller and Werner, 2010] propose a deep-embedding of higher-order logic. The syntax of terms is not typed and therefore the interpretation of terms can therefore fail. Armand et al. [Armand et al., 2011] are using a similar encoding for first-order logic to import SMT proofs into Coq. In this work, we have a syntax for terms that is similar to the AAC tactic [Braibant and Pous, 2011]. This has proved not too hard to work with these (weakly) dependent types. Retrospectively, we fear that it might be responsible for a very noticeable loss of performance. Moreover, it makes extension to polymorphic types and quantifiers more challenging - especially without axiom.

The current tactic is compiled using about one hundred instances and provides additional support for the types nat, positive, Z, comparison and bool. It allows an existing arithmetic tactic such as ring or lia to discharge goals using, for instance, Euclidean division or square root operators.

As future work, we consider binding pp(simpl more tightly with other reflective tactic such as ring or lia thus avoiding to perform a reification twice. For performance (and also to improve error reporting), we also wish to port the Ltac compiler to Ocaml.

### References


T. Braibant and D. Pous. Tactics for reasoning modulo AC in Coq. In this work, we have a syntax for terms that is similar to the AAC tactic [Braibant and Pous, 2011]. This has proved not too hard to work with these (weakly) dependent types. Retrospectively, we fear that it might be responsible for a very noticeable loss of performance. Moreover, it makes extension to polymorphic types and quantifiers more challenging - especially without axiom.

The current tactic is compiled using about one hundred instances and provides additional support for the types nat, positive, Z, comparison and bool. It allows an existing arithmetic tactic such as ring or lia to discharge goals using, for instance, Euclidean division or square root operators.

As future work, we consider binding pp(simpl more tightly with other reflective tactic such as ring or lia thus avoiding to perform a reification twice. For performance (and also to improve error reporting), we also wish to port the Ltac compiler to Ocaml.

### References


