ppsimpl: a reflexive Coq tactic for canonising goals

Frédéric Besson
Inria Rennes

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, ppsimpl, aiming at canonising goals so that the scope of existing tactics can be broaden 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 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 omega and lia for linear (integer) arithmetic; lra and fourier for linear real arithmetic; ring for solving ring equations; field for rational expressions; congruence for the logic of unintepreted functions and constructors; tauto and 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 type positive that is *almost* an integer. Likewise, dealing with booleans can be cumbersome whereas booleans are *almost* propositions. Moreover, tactics usually do not handle functions even when their specification fall in the relevant logic fragment. For (integer) arithmetic, this is particularly striking: functions such as max, division, modulo, square root can be fully specified using only addition and multiplication. To alleviate these problems, the Ltac tactic zify[1] tries hard to map different numeric types towards Z and ssreflect (Gonthier et al. 2008) makes a systematic link between booleans and propositions using so-called *small scale reflection*. Yet, zify is not easily extensible and ssreflect does not aim at *large scale reflection*.

```
Goal ∀ x y: bool, x && y = true → y = true.
Proof. intros x y; ppsimpl; tauto. Qed.

Goal ∀ x: nat, (x > 0 → sqrt x > x → x = 0).
Proof. intros; ppsimpl; nia. Qed.
```

Figure 1. ppsimpl at work

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 (Sozeau and Oury 2008) 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 https://scm.gforge.inria.fr/anonscm/git/ppsimpl/ppsimpl.git and checking out the ppsimpl-8.5 branch.

2. The tactic from the user perspective

The 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 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 TypeDecl.t 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 nat of Peano integers, the canonical default value is 0; the equivalence relation is Leinitz equality (=) over nat and the predicate states that all natural numbers are positive fun (x:nat) => x >= 0.

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

[1] Tough undocumented, zify is a quite useful tactic developed by P. Letouzey.

`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` = T_1 `-> ... ->` T_n by providing an instance of type `Inj.t Ty F`. The instance contains a function `F'` of type `Ty'` = T'_1 `-> ... ->` T'_n such that T'_i is the injection of type T_i (*i.e.,* we have an instance of type `DeclInj.t` T_i `_` T'_i `_` for each i); a proof that `inj (F` $x_1 \ldots x_n$`) == F' (inj` x_1`) ...(inj` x_n`)` and a proof that `F'` is a morphism for the relevant equivalences. For example, to define of an instance of type `Inj.t (nat -> nat) plus` we would provide the function `Z.add`; prove that `forall x y, Z.of_nat (plus x y) = Z.add (Z.of_nat x) (Z.of_nat y)` and that `forall x x' y y', x = x' /\y = y' -> Z.add x y = Z.add x' y'`. The definition of the type-class `Inj.t` is complicated 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` = T_1 `-> ... ->` T_n. This object is automatically constructed using type-class resolution using instances that ensure in particular that all the T_i and T'_i have been declared (there is an instance of `TypeDecl.t` T_i and `TypeDecl.t` T'_i).

The user can also 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`, we would provide the function `fun x => x + 1` and prove that `forall x, Zsucc x = x + 1`.

Eventually, the user can provide a predicate specifying the input-output relation of a function `F:` T_1 `->...->` T_{n-1} `->` T_n by declaring an instance of class `Abstract.t F`. The instance contains a predicate `P :` T_n `->` T_1 `... ->` T_{n-1} `-> Prop` and a proof that it correctly models the function *i.e.*, `forall r` $a_1 \ldots a_{n-1}$`, F` $a_1 \ldots a_n$ `= r -> P r` $a_1 \ldots a_{n-1}$. For example, to define an instance of type `Abstract.t Z.max`, we would provide the predicate `fun r x y => not (x < y) /\r = x \/ x < y /\r = y` and prove `r = Z.max x y -> not (x < y) / r = x \/ x < y /\r = y`. Again, the correctness lemma is generated by induction over the structure of the type of `F` using type classes.

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 extendable. We do not follow this path but explore 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 lemma[2].

The compilation done, the `ppsimpl` 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, `ppsimpl` 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 \/ ~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 `x` is named `e`.

```
x : bool
y : bool
==================================================
 True →
 (Is_true y ∨ ~ Is_true y) ∧ (Is_true x ∨ ~ Is_true x) →
 (Is_true x ∧ Is_true y ↔ True) → (Is_true y ↔ True)

x : nat
e : Z
==================================================
 ((0 ≤ Z.of_nat x) → (e ∗ e ≤ Z.of_nat x < (e + 1) ∗ (e + 1))) →
 (0 ≤ Z.of_nat x) →
 (0 < Z.of_nat x) → (Z.of_nat x < e) → Z.of_nat x = 0
```

Figure 2. Result of `ppsimpl` for the goals of Figure 1

3. Conclusion

Proof by reflection (Bertot and Castéran 2004, Chapter 17) 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 `lia` or `nia` to discharge goals using, for instance, Euclidean division or square root operators.

As future work, we consider binding `ppsimpl` 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

M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

T. Braibant and D. Pous. Tactics for reasoning modulo AC in coq. In *CPP*, volume 7086 of *LNCS*, pages 167–182. Springer, 2011.

G. Gonthier, A. Mahboubi, and E. Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

C. Keller and B. Werner. Importing HOL Light into Coq. In *ITP*, volume 6172 of *LNCS*, pages 307–322. Springer, 2010.

M. Sozeau and N. Oury. First-Class Type Classes. In *TPHOLs*, volume 5170 of *LNCS*, pages 278–293. Springer, 2008.

[2] For each of them, the user needs to explicitly call a dedicated fully automatic tactic.