



Boolean reflection via type classes

Benjamin Grégoire, Enrico Tassi

► **To cite this version:**

Benjamin Grégoire, Enrico Tassi. Boolean reflection via type classes. Coq Workshop, Aug 2016, Nancy, France. 2016. <hal-01410530>

HAL Id: hal-01410530

<https://hal.inria.fr/hal-01410530>

Submitted on 6 Dec 2016

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.

Copyright}

Boolean reflection via type classes

Benjamin Gregoire Enrico Tassi

June 21, 2016

Boolean reflection is a formalization technique that represents decidable predicates with their decision procedures and where truth values become booleans. Reflection occurs in the small scale: since conjectures are stated using programs their symbolic execution provides a valuable form of automation. In this approach the user faces the “syntactic” (`bool`) representation of the conjecture and is given tactic-level tools to switch to the “semantic” one (`Prop`) and back.

The `SSReflect` proof language [1] provides the view mechanism to switch from the computational realm of `bool` to the semantic one of `Prop`. To minimize the syntactic noise due to view application `SSReflect` accepts views as annotations of most linguistic constructs. Still a user needs to mention the view name explicitly, even when there is only one view to be applied. We propose a type-class [2] based machinery to attach canonical views to predicates and connectives to relief the `Coq` user from some of the bookkeeping required by the boolean reflection formalization technique.

Let’s take a very simple example. Here the `is_true` constant is used to state the truth of a boolean predicates. Being declared as a coercion is automatically inserted by `Coq` around any boolean value occurring in a context expecting a `Prop`. The support lemmas `andP` and `orP` are views linking the boolean connectives `&&` and `||` to their meaning in `Prop`. The `reflect` predicate simply states that its first argument, in `Prop`, holds if and only its second argument, in `bool`, is equal to `true`.

```
Definition is_true b := b = true.
Coercion is_true : bool -> Sortclass. (* Prop *)
Lemma andP b1 b2 : reflect (b1 /\ b2) (b1 && b2).
Lemma orP b1 b2 : reflect (b1 \/ b2) (b1 || b2).

Lemma example_bool a b : ((a && b) || a) -> a
Proof. by move=> /orP[ /andP[ Ha Hb ] | Ha ]; assumption. Qed.

Lemma example_prop a b : ((a /\ b) \/ a) -> a
Proof. by move=> [ [ Ha Hb ] | Ha ]; assumption. Qed.
```

The `example_bool` proof¹ applies the two views in order to de-structure the assumption. The second proof needs no such bookkeeping, since the conjecture is already stated in `Prop`.

We propose a declarative way of associating canonical views to connectives and predicates and a generic view name `xP` to select the view fitting the current

¹A much simpler proof would be to enumerate truth values as in “by case a; case b”. For the sake of clarity we picked an oversimple example.

context. We provide a recursive variant `lxP` that pushes views recursively along logical connectives, as well as other variants that push views recursively to predicates `rxP` or under binders `rbxP`. The resulting proof script looks as follows.

```
Lemma example_bool a b : ((a && b) || a) -> a
Proof. by move=> /lxP[ [ Ha Hb ] | Ha ]; assumption. Qed.
```

The declaration of the canonical view for the boolean conjunction follows.

```
Instance andV m rm p1 p2 b1 b2 '{Valid Logic m} '{LogicRec m rm}
  '{View rm p1 b1} '{View rm p2 b2} : View m (p1 /\ p2) (b1 && b2).
```

The `m` (mode) and `rm` (recursive mode) variables are used to constrain the view application and relate it to the views that will be recursively applied. The value of `m` is part of the input, for example `m` can signal a one level deep view application (e.g. for `xP`) or a recursive one but limited to logical connectives (for `lxP`). The `LogicRec` relation decides which kind of view is accepted in the recursive calls (only the identity view for `xP`). `Valid` is a relational predicate stating that for the view to be applicable the required mode must be one that includes logical rules.

A more advanced example is the view for the `has` boolean predicate, that asserts that an element of a list validates a given predicate `p`.

```
Instance hasV m rm (T : eqType) (P : T -> Prop) (p : pred T) l
  '{Valid NonLogic m} '{BindRec m rm} '{forall x, View rm (P x) (p x)} :
  View m (exists x, x \in l /\ P x) (has p l).
```

Note how the type class mechanism let us express that `p` and `P` are related by a view under a context augmented with `x`. Here `BindRec` constrains the recursive call to find only the identity view unless the current mode enables pushing views under binders. The `Valid` premise asserts that the view is valid when the views labelled as non-logical are enables (e.g. in `xP` but not in `lxP`).

```
Lemma example_has l : has [pred x | 0 < x <= 7] l.
Proof. apply /rbxP.
```

The application of `/rbxP` pushes views under binders

```
l : list nat
=====
exists x : nat, x \in l /\ 0 < x /\ x <= 7
```

While `/rxP` stops at the binder frontier.

```
l : list nat
=====
exists x : nat, x \in l /\ [pred y | 0 < y <= 7] x
```

The Coq code is available at <http://github.com/gares/autoview>

References

- [1] G. Gonthier, A. Mahboubi, E. Tassi. A Small Scale Reflection Extension for the Coq system. RR-6455. 2015
- [2] M. Sozeau, N. Oury. First-Class Type Classes. TPHOLs, LNCS 5170, pages 278-293, 2008.