Reflection & Views

Pierre-Yves Strub
13 March 2012

MAP INTERNATIONAL SPRING SCHOOL ON FORMALIZATION OF MATHEMATICS 2012

SOPHIA ANTIPOLIS, FRANCE / 12-16 MARCH

Inria informatics mathematics

ihp Institut Henri Poincaré

MICROSOFT RESEARCH INRIA JOINT CENTRE

The `Prop` / `bool` duality

SSReflect slogan:

1. write boolean predicates reflecting decidable propositions,
2. use an interleaving of computation steps and deduction step.

$$\forall x, P x \Leftrightarrow f x = true$$

$$\begin{tikzcd} P t \arrow[r, "\text{reflection}"] & f t = true \arrow[d, "\text{computations}"] \\ & true = true \end{tikzcd}$$

Outline

Changing the point of view (`apply/V`, `case/V`)

Boolean reflection (`reflect`, `apply: (iffP V)`)

Boolean equivalence (`apply/V1/V2`)

Alternate induction principle (`elim/V`)

Changing the point of view

For assumptions

Interpreting an assumption is applying a correspondence lemma before generalizing it or doing a case analysis.

Changing the point of view

For assumptions

Interpreting an assumption is applying a correspondence lemma before generalizing it or doing a case analysis.

```
move=> a h; move: (P2Q _ h); move=> {h} h
```

```
 P : T -> Prop
 Q : T -> Prop
 P2Q : forall x, P x -> Q x
===========================
 forall a, P a -> G
```

Changing the point of view

For assumptions

Interpreting an assumption is applying a correspondence lemma before generalizing it or doing a case analysis.

```
move=> a h; move: (P2Q _ h); move=> {h} h
```

```
 P : T -> Prop
 Q : T -> Prop
 P2Q : forall x, P x -> Q x
 a : T
 h : P a
===========================
 G
```

Changing the point of view

For assumptions

Interpreting an assumption is applying a correspondence lemma before generalizing it or doing a case analysis.

```
move=> a h; move: (P2Q _ h); move=> {h} h
```

```
 P : T -> Prop
 Q : T -> Prop
 P2Q : forall x, P x -> Q x
 a : T
 h : P a
===========================
 Q a -> G
```

Changing the point of view

For assumptions

Interpreting an assumption is applying a correspondence lemma before generalizing it or doing a case analysis.

```
move=> a h; move: (P2Q _ h); move=> {h} h
```

```
 P : T -> Prop
 Q : T -> Prop
 P2Q : forall x, P x -> Q x
 a : T
 h : Q a
===========================
 G
```

Changing the point of view

For assumptions

Interpreting an assumption is applying a correspondence lemma before generalizing it or doing a case analysis.

The tactic `move/V: h => h` does the same: it applies the correspondence lemma `V` (a view lemma) to assumption `h`.

Changing the point of view

For assumptions

Interpreting an assumption is applying a correspondence lemma before generalizing it or doing a case analysis.

The tactic `move/V: h => h` does the same: it applies the correspondence lemma `V` (a view lemma) to assumption `h`.

```
move/P2Q: h => h
```

```
  ...
  a : T
  h : P a
=================
  G
```

Changing the point of view

For assumptions

Interpreting an assumption is applying a correspondence lemma before generalizing it or doing a case analysis.

The tactic `move/V: h => h` does the same: it applies the correspondence lemma `V` (a view lemma) to assumption `h`.

```
move/P2Q: h => h
```

```
  ...
  a : T
  h : P a
=================
  G
```

```
  ...
  a : T
  h : Q a
=================
  G
```

Changing the point of view

For assumptions

Views are applied modulo view hints. For instance, for double implications, the two following view hints are predefined:

- `iffLR : forall P Q, (P <-> Q)-> P -> Q`
- `iffRL : forall P Q, (P <-> Q)-> Q -> P`

Changing the point of view

For assumptions

Views are applied modulo view hints. For instance, for double implications, the two following view hints are predefined:

- `iffLR : forall P Q, (P <-> Q)-> P -> Q`
- `iffRL : forall P Q, (P <-> Q)-> Q -> P`

```
move/P2Q: h => h
```

```
 ...
 P2Q :
  forall x, P x <-> Q x
 a : T
 h : P a
============================
 G
```

Changing the point of view

For assumptions

Views are applied modulo view hints. For instance, for double implications, the two following view hints are predefined:

- `iffLR : forall P Q, (P <-> Q)-> P -> Q`
- `iffRL : forall P Q, (P <-> Q)-> Q -> P`

```
move/P2Q: h => h [move/(iffLR (P2Q _))]
```

```
 ...
 P2Q :
  forall x, P x <-> Q x
 a : T
 h : P a
========================
 G
```

```
 ...
 P2Q :
  forall x, P x <-> Q x
 a : T
 h : Q a
========================
 G
```

Changing the point of view

For assumptions

Views can be applied before doing a case analysis.

Changing the point of view

For assumptions

Views can be applied before doing a case analysis.

```
case/V: h
```

```
  ...
  P : nat -> Prop
  Q : nat -> Prop
  V : forall n, Q n ->
         (P n) \/ (P n.+1)
  a : nat
  h : Q a
 ========================
  G
```

Changing the point of view

For assumptions

Views can be applied before doing a case analysis.

```
case/V: h
```

```
...
P : nat -> Prop
Q : nat -> Prop
V : forall n, Q n ->
      (P n) \/ (P n.+1)
a : nat
h : Q a
=========================
G
```

```
...
h : Q a
================
P a -> G
```

```
...
h : Q a
=================
P a.+1 -> G
```

Changing the point of view

For goals

Finally, a view can be applied to a goal using the `apply/V` tactic.

Changing the point of view

For goals

Finally, a view can be applied to a goal using the apply/V tactic.

```
apply/PQ

  P : Prop
  Q : Prop
  P2Q : P <-> Q
 =================
  P
```

Changing the point of view

For goals

Finally, a view can be applied to a goal using the `apply/V` tactic.

```
apply/PQ [apply: (iffRL P2Q)]
```

```
P : Prop                    P : Prop
Q : Prop                    Q : Prop
P2Q : P <-> Q               P2Q : P <-> Q
=================          =================
P                           Q
```

Changing the point of view

Specialization

`move/(_ x1 .. xn)=> h` specializes the introduced hypothesis by applying `x1 .. xn` to it.

Changing the point of view

Specialization

`move/(_ x1 .. xn)=> h` specializes the introduced hypothesis by applying `x1 .. xn` to it.

```
move/(_ x)=> h

  P : nat -> Prop
  x : nat
 ==========================
  (forall n, P (2 * n)) -> G
```

Changing the point of view

Specialization

`move/(_ x1 .. xn)=> h` specializes the introduced hypothesis by applying `x1 .. xn` to it.

```
move/(_ x)=> h
```

```
P : nat -> Prop
x : nat
============================
(forall n, P (2 * n)) -> G
```

```
P : nat -> Prop
x : nat
h : P (2 * x)
==================
G
```

Outline

Changing the point of view (`apply/V`, `case/V`)

Boolean reflection (`reflect`, apply: `(iffP V)`)

Boolean equivalence (`apply/V1/V2`)

Alternate induction principle (`elim/V`)

Boolean reflection

Double implication

Boolean reflection is an equivalence property between a predicate (in `Prop`) and a boolean predicate:

```
Lemma andE:
  forall b1 b2, (b1 /\ b2) <-> (b1 && b2).
```

Boolean reflection

Double implication

Boolean reflection is an equivalence property between a predicate (in `Prop`) and a boolean predicate:

```
Lemma andE:
  forall b1 b2, (b1 /\ b2) <-> (b1 && b2).
```

SSReflect uses a dedicated predicate for boolean reflection:

```
Inductive reflect (P : Prop) : bool -> Type :=
| ReflectT :   P -> reflect P true
| ReflectF : ~ P -> reflect P false
```

Boolean reflection

The `reflect` predicate

```
Inductive reflect (P : Prop) : bool -> Type :=
| ReflectT :   P -> reflect P true
| ReflectF : ~ P -> reflect P false
```

Boolean reflection

The `reflect` predicate

```
Inductive reflect (P : Prop) : bool -> Type :=
| ReflectT :   P -> reflect P true
| ReflectF : ~ P -> reflect P false
```

`reflect P b` states that `P` is logically equivalent to `is_true b`: an inhabitant of `reflect P b` is either:

- `ReflectT p` with `p : P` and `b = true`, or
- `ReflectF p` with `p : ~P` and `b = false`.

Boolean reflection for conjunction is expressed as:

```
Lemma andP: forall b1 b2,
  reflect (b1 /\ b2) (b1 && b2).
```

Boolean reflection

The `reflect` predicate

```
Inductive reflect (P : Prop) : bool -> Type :=
| ReflectT :   P -> reflect P true
| ReflectF : ~ P -> reflect P false
```

```
case: (andP b1 b2)

 b1 : bool
 b2 : bool
==============================
 if b1 && b2 then G1 else G2
```

Boolean reflection

The `reflect` predicate

```
Inductive reflect (P : Prop) : bool -> Type :=
| ReflectT :   P -> reflect P true
| ReflectF : ~ P -> reflect P false
```

```
case: (andP b1 b2) [P = b1 /\ b2]
```

```
 b1 : bool
 b2 : bool
======================
 (b1 /\ b2) -> G1
```

```
 b1 : bool
 b2 : bool
======================
 ~(b1 /\ b2) -> G2
```

```
[b1 && b2 = true]
```

```
[b1 && b2 = false]
```

Boolean reflection

`reflect` and views

The `reflect` predicate is compatible with views.
The view mechanism guesses with direction to use.

`case/andP`

```
  a : bool
  b : bool
 =============
  a && b -> G
```

Boolean reflection

`reflect` and views

The `reflect` predicate is compatible with views.
The view mechanism guesses with direction to use.

```
case/andP

  a : bool                      a : bool
  b : bool                      b : bool
 =============                 =============
  a && b -> G                   a -> b -> G
```

Boolean reflection

`reflect` and views

The `reflect` predicate is compatible with views.
The view mechanism guesses with direction to use.

```
apply/andP

  a : bool
  b : bool
 =============
  a /\ b
```

Boolean reflection

`reflect` and views

The `reflect` predicate is compatible with views.
The view mechanism guesses with direction to use.

`apply/andP`

```
 a : bool
 b : bool
==============
 a /\ b
```

```
 a : bool
 b : bool
==============
 a && b
```

Boolean reflection

Some `reflect` statements

For logical operators:

```
orP   : reflect (b1 \/ b2)(b1 || b2)
andP  : reflect (b1 /\ b2)(b1 && b2)
negP  : reflect (~ b)(~~ b)

negPf : reflect (b = false)(~~ b)
norP  : reflect (~~ b1 \/ ~~ b2)(~~ (b1 && b2))
nandP : reflect (~~ b1 /\ ~~ b2)(~~ (b1 || b2))
```

Boolean reflection

Some `reflect` statements

For equalities:

```
Fixpoint eqn m n :=
  match m, n with
  | 0 , 0 => true
  | m'.+1 , n'.+1 => eqn m' n'
  | _ , _ => false
  end.

Lemma eqnP :
  forall (n m : nat), reflect (n = m) (eqn n m).
```

Boolean reflection

Some `reflect` statements

For equalities (with `eqType`). SSReflect comes with a predefined type for types having a decidable equality.

A type `T : eqType` comes with boolean predicate `eq_op T` of type `T -> T -> bool` (and written `_ == _`), along with a proof of reflection:

```
eqP T : forall (x y : T), reflect (x = y)(x == y)
```

Boolean reflection

Some `reflect` statements

```
Variable (T : eqType) (p : T -> bool).

Fixpoint all (s : seq T) :=
  if s is x :: s' then a x && all s' else true.

Fixpoint seqmem (z : T) (s : seq T) :=
  if s is x :: s'
    then (x == z) || (seqmem z s')
    else false.

Lemma allP:
  reflect (forall x, seqmem x s -> p x) (all s).
```

Boolean reflection

Proving `reflect P b`

How to prove `reflect P b` ?

1. By case analysis on B.

```
Lemma idP: reflect b b.
Proof.
  case: b.
    apply ReflectT.
    apply ReflectF.
Qed.
```

Boolean reflection

Proving `reflect P b`

How to prove `reflect P b` ?

2. Using `iffP`

```
Lemma iffP:
  reflect P b -> (P -> Q) -> (Q -> P)
    -> reflect Q b.
```

```
apply: (iffP idP)

  P : Prop
  Q : Prop
  b : bool
 ============
  reflect P b
```

Boolean reflection

Proving `reflect P b`

How to prove `reflect P b` ?

2. Using `iffP`

```
Lemma iffP:
  reflect P b -> (P -> Q) -> (Q -> P)
    -> reflect Q b.
```

```
apply: (iffP idP)

  P : Prop
  Q : Prop
  b : bool                ...           ...
 ============          =========     =========
  reflect P b           b -> P        P -> b
```

Boolean reflection

`reflect` as a function

SSReflect provides a coercion `elimT` from a boolean `b` to a proposition `P`, provided that `reflect P b`.

Boolean reflection

`reflect` as a function

SSReflect provides a coercion `elimT` from a boolean `b` to a proposition `P`, provided that `reflect P b`.

```
rewrite (eqP h)

  b1 : bool
  b2 : bool
  h : b1 == b2
 ============
  G b1
```

Boolean reflection

`reflect` as a function

SSReflect provides a coercion `elimT` from a boolean `b` to a proposition `P`, provided that `reflect P b`.

`rewrite (eqP h)` [`rewrite (elimT eqP) h`]

```
b1 : bool
b2 : bool
h : b1 == b2
=============
G b1
```

```
b1 : bool
b2 : bool
h : b1 == b2
=============
G b2
```

Outline

Changing the point of view (`apply/V`, `case/V`)

Boolean reflection (`reflect`, `apply: (iffP V)`)

Boolean equivalence (`apply/V1/V2`)

Alternate induction principle (`elim/V`)

Interpreting equivalences

`apply/V1/V2`

From a logical point of view, there is no difference between

- being equal boolean values (`b1 = b2`), and
- being equivalent (coerced) boolean values (`b1 <-> b2`).

In practice, using equalities instead of double implication is preferable as it allows the use of `rewrite`:

Interpreting equivalences

apply/V1/V2

From a logical point of view, there is no difference between

- being equal boolean values (`b1 = b2`), and
- being equivalent (coerced) boolean values (`b1 <-> b2`).

In practice, using equalities instead of double implication is preferable as it allows the use of `rewrite`:

```
leq_eqVlt :
  forall m n, (m <= n) = (m == n) || (m < n)

rewrite leq_eqVlt

   m : nat
   n : nat
  =================
   G && (m <= n)
```

Interpreting equivalences

apply/V1/V2

From a logical point of view, there is no difference between

- being equal boolean values (`b1 = b2`), and
- being equivalent (coerced) boolean values (`b1 <-> b2`).

In practice, using equalities instead of double implication is preferable as it allows the use of `rewrite`:

```
leq_eqVlt :
  forall m n, (m <= n) = (m == n) || (m < n)
```

```
rewrite leq_eqVlt
```

```
  m : nat
  n : nat
=================
  G && (m <= n)
```

```
m : nat
n : nat
=============================
G && ((m == n) || (m < n))
```

Interpreting equivalences

`apply/V1/V2`:

- applies the goals of the form `p1 = p2`
  with `p1`, `p2` boolean expressions,
- transforms the goal into a double implication,
- applies view `V1` (resp. `V2`) to `p1` (resp. `p2`).

Interpreting equivalences

`apply/V1/V2`:

- applies the goals of the form `p1 = p2`
  with `p1`, `p2` boolean expressions,
- transforms the goal into a double implication,
- applies view `V1` (resp. `V2`) to `p1` (resp. `p2`).

`apply/idP/idP`

```
==========
 p1 = p2
```

Interpreting equivalences

`apply/V1/V2`:

- applies the goals of the form `p1 = p2`
  with `p1`, `p2` boolean expressions,
- transforms the goal into a double implication,
- applies view `V1` (resp. `V2`) to `p1` (resp. `p2`).

`apply/idP/idP`

```
==========          ==========          ==========
 p1 = p2             p1 -> p2            p2 -> p1
```

Interpreting equivalences

apply/V1/V2

apply/norP/idP

```
 b1 b2 b3 : bool
==================
~~(b1 || b2) = b3
```

Interpreting equivalences

apply/V1/V2

```
apply/norP/idP
```

```
  b1 b2 b3 : bool
==================
 ~~(b1 || b2) = b3
```

```
 b1 b2 b3 : bool
==================
 ~~b1 /\ ~~b2 -> b3
```

```
 b1 b2 b3 : bool
==================
 b3 -> ~~b1 /\ ~~b2
```

Outline

Changing the point of view (`apply/V`, `case/V`)

Boolean reflection (`reflect`, `apply: (iffP V)`)

Boolean equivalence (`apply/V1/V2`)

Alternate induction principle (`elim/V`)

Interpreting eliminations

`elim/V`

`elim/V` allows to specify an alternative induction principle.

Interpreting eliminations

elim/V

elim/V allows to specify an alternative induction principle.

Standard (uninterpreted) elimination uses the generated induction principle, derived from the inductive definition.

```
forall (P : nat -> Prop),
  P 0 -> (forall n, P n -> P n.+1) ->
    forall n, P n.
```

Interpreting eliminations

elim/V

elim/V allows to specify an alternative induction principle.

Standard (uninterpreted) elimination uses the generated induction principle, derived from the inductive definition.

```
Variables P : nat -> Prop.

elim: n

  n : nat
 ============

  P n
```

Interpreting eliminations

elim/V

elim/V allows to specify an alternative induction principle.

Standard (uninterpreted) elimination uses the generated induction principle, derived from the inductive definition.

```
Variables P : nat -> Prop.

elim: n

  n : nat
 ============      ============      ============
                                      forall n,
  P n               P 0                 P n -> P n.+1
```

Interpreting eliminations

`elim/V`

Stronger induction principle are derivable:

```
forall (P : nat -> Prop),
  (forall n, (forall p, p < n -> P p) -> P n) ->
    forall n, P n.
```

Interpreting eliminations

elim/V

Stronger induction principle are derivable:

```
forall (P : nat -> Prop),
  (forall n, (forall p, p < n -> P p) -> P n) ->
    forall n, P n.
```

```
Variables P : nat -> Prop.

elim/nat_sind: n

   n : nat
  ============
   P n
```

Interpreting eliminations

elim/V

Stronger induction principle are derivable:

```
forall (P : nat -> Prop),
  (forall n, (forall p, p < n -> P p) -> P n) ->
    forall n, P n.
```

```
Variables P : nat -> Prop.

elim/nat_sind: n

   n : nat
  ============    ===========================
   P n                forall n,
                        (forall p, p < n -> P p) ->
                          P n
```