Advanced tactics

Enrico Tassi
13 March

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

Outline

Terminology

The stack

Context

```
x : T
S : {set T}
xS : x \in S
```

The bar `================`

Goal

$$\underbrace{\texttt{forall y, y == x ->}}_{\textit{Assumptions}}\ \underbrace{\texttt{y }\backslash\texttt{in S}}_{\textit{Conclusion}}$$

Top is the first assumption, `y` here

Stack alternative name for the list of *Assumptions*

The real syntax of SSR

The real syntax is not this one:

```
move=> x Hx
move/andP: h => h
case: x
case/andP: x
```

These are compound tactics, the building blocks are:

- `move` and `case` are the tactics acting on Top
- `: gen gen ...` runs *before* the `tactic` to load the goal
- `=> ipat ipat ...` runs *after* the `tactic` to load the context
- `/andP` is a view application on Top

Defective tactics

example

The implicit argument is `Top`:

```
case.


  ====================
   forall b : bool, P b
```

Defective tactics

example

The implicit argument is `Top`:

```
case.
```

```
  ===================                =========      =========
   forall b : bool, P b              P true         P false
```

Equivalent to:

```
move=> Top. case: Top.
```

Loading the goal

simple generalization

Slow motion for:

```
case: ab.
```

```
  ab : A /\ B
===============
  G
```

Loading the goal

simple generalization

Slow motion for:

```
case: ab.
```

```
  ab : A /\ B
===============        =========
  G                      A /\ B -> G
```

Loading the goal

simple generalization

Slow motion for:

```
case: ab.
```

```
  ab : A /\ B
===============        =========          =========
  G                     A /\ B -> G        A -> B -> G
```

Loading the goal

true generalization

We can specify some items of the context that occur in the goal:

```
move: n m.
```

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

Loading the goal

true generalization

We can specify some items of the context that occur in the goal:

```
move: n m.
```

```
  n : nat
  m : nat      n : nat
 =========    ===============
  P n m        forall m, P n m
```

Loading the goal

true generalization

We can specify some items of the context that occur in the goal:

```
move: n m.
```

```
 n : nat
 m : nat       n : nat
=========    ================    ================
 P n m        forall m, P n m     forall n m, P n m
```

Loading the goal

complex generalization

We can specify the occurrences we want to grab, and to keep the context item:

```
move: n.+1 {1}m.
```

```
  n : nat
  m : nat
=========
  P n.+1 m m
```

Loading the goal

complex generalization

We can specify the occurrences we want to grab, and to keep the context item:

```
move: n.+1 {1}m.
```

```
  n : nat          n : nat
  m : nat          m : nat
 =========        ================
  P n.+1 m m       forall m0,
                     P n.+1 m0 m
```

Loading the goal

complex generalization

We can specify the occurrences we want to grab, and to keep the context item:

```
move: n.+1 {1}m.
```

```
  n : nat            n : nat                n : nat
  m : nat            m : nat                m : nat
=========          ===============        ===============
  P n.+1 m m         forall m0,             forall n0 m0,
                       P n.+1 m0 m            P n0 m0 m
```

Loading the goal

lemma generalization

We can generalize a lemma like

```
ltnSn : forall m, m < m.+1
```

```
move: (ltnSn n).
```

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

Loading the goal

lemma generalization

We can generalize a lemma like

```
ltnSn : forall m, m < m.+1
```

```
move: (ltnSn n).
```

```
 n : nat              n : nat
=========            =============
 P n                  n < n.+1 -> P n
```

Views

viewing Top differently

Views applied to Top:

```
case/andP.
```

```
  a : nat
  b : nat
 =========
  P a && P b -> G
```

Views

viewing Top differently

Views applied to Top:

`case/andP.`

```
 a : nat              a : nat
 b : nat              b : nat
=========            =========
 P a && P b -> G      P a /\ P b -> G
```

Views

viewing Top differently

Views applied to Top:

```
case/andP.
```

```
 a : nat                 a : nat                 a : nat
 b : nat                 b : nat                 b : nat
=========               =========               =========
 P a && P b -> G         P a /\ P b -> G         P a -> P b -> G
```

Exception

Custom induction

You have already seen that `elim/view` makes an exception:

`elim/last_ind: s`

What is an elimination principle?

```
last_ind : forall T (P : seq T -> Prop),
  P [::] ->
  (forall s x, P s -> P (rcons s x)) ->
  forall s : seq T, P s
```

Multiple induction

The custom elimination principle can eliminate many items at the same time:

```
my_ind : forall T P,
  P [::] [::] ->
  (forall x xs y ys,
     P xs ys -> P (x :: xs) (y :: ys)) ->
  forall s1 s2 : seq T, size s2 = size s1 -> P s1 s2
```

```
elim/my_ind: s1 / s2.
```

Loading the context

views

Views can be applied in the middle of an intro pattern:

tactic`=> a b /andP pab qa`

```
====================
 forall a b : nat,
 P a && P b -> Q a -> G
```

```
a : nat
b : nat
pab : P a /\ P b
qa : Q a
====================
G
```

Equivalent to:

tactic`=> a b. move/andP=> pab qa.`

Loading the context

destructuring

Case analysis, usually to unpack, can be performed too:

tactic`=> a b /andP[pa pb] qa`

```
====================
 forall a b : nat,
 P a && P b -> Q a -> G
```

```
a  : nat
b  : nat
pa : P a
pb : P b
qa : Q a
====================
G
```

Equivalent to:

tactic`=> a b. case/andP=> pa pb qa.`

Loading the context

case split, two goals at once

Real case analysis can be performed as follows:

tactic`=> a [Pa | Qa]`

```
                          a : nat              a : nat
                          Pa : P a             Qa : Q a
==================        ============         ============
 forall a : nat,          G                    G
   P a \/ Q a -> G
```

Equivalent to:

tactic`=> a. case.`

```
  move=> Pa.
   ...
move=> Qa.
...
```

Loading the context

case split (exception)

When the tactic is `case` or `elim`, brackets just after => do not perform (an additional) case analysis.

```
elim=> [ | x IH]
```

Loading the context

flags and combo

Cleanup flags:

`//`	gets rid of trivial goals
`/=`	simplifies the goals
`//=`	short for `//` and `/=`
`{h}`	clears `h`

Moreover `:` and `=>` can be combined together:

```
elim: n => [ // | x IH] /=.
```

Idiom

General induction

The goal can be prepared to obtain a stronger induction principle:

```
elim: n.+1 {-2}n (ltnSn n) => [// | {n} n IH j le_jn]
```

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

Idiom

General induction

The goal can be prepared to obtain a stronger induction principle:

```
elim: n.+1 {-2}n (ltnSn n) => [// | {n} n IH j le_jn]
```

```
 n : nat
============
 n < n.+1 -> P n
```

Idiom

General induction

The goal can be prepared to obtain a stronger induction principle:

```
elim: n.+1 {-2}n (ltnSn n) => [// | {n} n IH j le_jn]
```

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

Idiom

General induction

The goal can be prepared to obtain a stronger induction principle:

```
elim: n.+1 {-2}n (ltnSn n) => [// | {n} n IH j le_jn]
```

```
  n : nat
 ============
  forall i m, m < i -> P m
```

Idiom

General induction

The goal can be prepared to obtain a stronger induction principle:

```
elim: n.+1 {-2}n (ltnSn n) => [// | {n} n IH j le_jn]
```

```
 n : nat
=============
 forall m,
  m < 0 -> P m
```

```
 n : nat
=============
 forall i,
  (forall m, m < i -> P m) ->
   forall m, m < i.+1 -> P m
```

Idiom

General induction

The goal can be prepared to obtain a stronger induction principle:

```
elim: n.+1 {-2}n (ltnSn n) => [// | {n} n IH j le_jn]
```

```
  n : nat
  IH : forall m, m < n -> P m
  j : nat
  le_jn : j < n.+1
============
   P j
```

Loading the context

substitution

Equations can be substituted on the fly, and unneeded hypotheses cleared

```
case: ex => y [-> yA] {x}
```

```
 x : T
 ex : exists y : T,
      x = f @*^-1 y /\
      y \in A
 ======================
   f @* x \in A
```

Loading the context

substitution

Equations can be substituted on the fly, and unneeded hypotheses cleared

```
case: ex => y [-> yA] {x}
```

```
x : T
ex : exists y : T,
     x = f @*^-1 y /\
     y \in A
=====================
  f @* x \in A
```

```
y : T
yA : y \in A
===============
  f @* (f @*^-1 y) \in A
```

Idioms

Hypotheses refinement & substitution

The `have` tactic accepts the same flags of =>.
The context can be refined and kept clean with `have`:

```
have {hyp1 hyp2} hyp3 : statement
  ...
...
```

Another example is with one shot equations.

```
have /andP[pa /eqP-> {b}] : P a && b == a
  ...
...
```

Outline

Ambiguity

Instantiation and occurrence

```
Lemma addnC x y : x + y = y + x. Proof. ... Qed.
Lemma mulnC x y : x * y = y * x. Proof. ... Qed.

Lemma ex a b : (a + b)^2 = (c + d) * (a + b).
Proof. rewrite addnC.
```

Ambiguity

Instantiation and occurrence

```
Lemma addnC x y : x + y = y + x. Proof. ... Qed.
Lemma mulnC x y : x * y = y * x. Proof. ... Qed.

Lemma ex a b : (a + b)^2 = (c + d) * (a + b).
Proof. rewrite (addnC _ _).
```

The pattern `(_ + _)` has many matches:

Ambiguity

Instantiation and occurrence

```
Lemma addnC x y : x + y = y + x. Proof. ... Qed.
Lemma mulnC x y : x * y = y * x. Proof. ... Qed.

Lemma ex a b : (a + b)^2 = (c + d) * (a + b).
Proof. rewrite (addnC _ _).
```

The pattern `(_ + _)` has many matches:

```
(a + b)^2 = (c + d) * (a + b)
```

Ambiguity

Instantiation and occurrence

```
Lemma addnC x y : x + y = y + x. Proof. ... Qed.
Lemma mulnC x y : x * y = y * x. Proof. ... Qed.

Lemma ex a b : (a + b)^2 = (c + d) * (a + b).
Proof. rewrite (addnC _ _).
```

The pattern `(_ + _)` has many matches:

```
(a + b)^2 = (c + d) * (a + b)
```

```
(a + b)^2 = (c + d) * (a + b)
```

Ambiguity

Instantiation and occurrence

```
Lemma addnC x y : x + y = y + x. Proof. ... Qed.
Lemma mulnC x y : x * y = y * x. Proof. ... Qed.

Lemma ex a b : (a + b)^2 = (c + d) * (a + b).
Proof. rewrite (addnC _ _).
```

The pattern `(_ + _)` has many matches:

```
(a + b)^2 = (c + d) * (a + b)
```

```
(a + b)^2 = (c + d) * (a + b)
```

```
(a + b)^2 = (c + d) * (a + b)
```

Ambiguity

Instantiation and occurrence

```
Lemma addnC x y : x + y = y + x. Proof. ... Qed.
Lemma mulnC x y : x * y = y * x. Proof. ... Qed.

Lemma ex a b : (a + b)^2 = (c + d) * (a + b).
Proof. rewrite (mulnC _ _).
```

The pattern `(_ * _)` has many matches:

Ambiguity

Instantiation and occurrence

```
Lemma addnC x y : x + y = y + x. Proof. ... Qed.
Lemma mulnC x y : x * y = y * x. Proof. ... Qed.

Lemma ex a b : (a + b)^2 = (c + d) * (a + b).
Proof. rewrite (mulnC _ _).
```

The pattern `(_ * _)` has many matches:

`(a + b)^2 = (c + d) * (a + b)`

Ambiguity

Instantiation and occurrence

```
Lemma addnC x y : x + y = y + x. Proof. ... Qed.
Lemma mulnC x y : x * y = y * x. Proof. ... Qed.

Lemma ex a b : (a + b)^2 = (c + d) * (a + b).
Proof. rewrite (mulnC _ _).
```

The pattern `(_ * _)` has many matches:

`(a + b)^2 = (c + d) * (a + b)`

`(a + b)^2 = (c + d) * (a + b)`

The SSR approach

It's all about patterns:

- Inferred looking at the rewrite rule
- Eventually overridden by the user

Matching discipline:

1. Traverse the goal outside in, left to right
2. Look for verbatim copies of the key of the pattern e.g. `(_ + _)`
3. There you match up to computation
4. If the matching fails, try the next occurrence of the key
5. If the matching succeeds, that subterm is the *only* instance of the pattern considered.

Note: the instance of the pattern may occur multiple times

Inferred pattern

Recall the goal:

`(a + b)^2 = (c + d) + (a + b)`

To rewrite there we can decorate the rule with a pattern:

`rewrite addnC`

The first instance of the pattern `(_ + _)` is:

`(a + b)^2 = (c + d) + (a + b)`

That occurs twice:

`(a + b)^2 = (c + d) + (a + b)`

The result is:

`(b + a)^2 = (c + d) + (b + a)`

Simple pattern

Recall the goal and consider the target:

`(a + b)^2 = (c + d) + (a + b)`

To rewrite there we can decorate the rule with a pattern:

```
rewrite [c + _]addnC
```

The pattern `[c + _]` selects:

`(a + b)^2 = (c + d) + (a + b)`

The result is:

`(a + b)^2 = (d + c) + (a + b)`

Simple pattern

forcing unfolding

Recall the goal and our target:

```
Lemma sumn_nseq x n : sumn (nseq n x) = x * n

(a + b)^2 = (c + d) + (a + b)
```

The *key* of the given pattern differs from the inferred one:

```
rewrite -[_^2]sumn_nseq
```

The pattern `[_^2]` selects:

```
(a + b)^2 = (c + d) + (a + b)
```

The result is:

```
sumn (nseq (a + b) (a + b)) = (c + d) + (a + b)
```

Simple contextual pattern

Assume you want to rewrite only the blue subterm:

`(a + b)^2 = (c + d) + (a + b)`

Instead of using the occurrence number `{1}` to identify the occurrence, one can use its context:

```
rewrite [in _^2]addnC
```

The pattern selects:

`(a + b)^2 = (c + d) + (a + b)`

Then all its subterms are matched against the inferred pattern `(_ + _)`:

`(b + a)^2 = (c + d) + (a + b)`

Precise contextual pattern

Assume you want to rewrite only the blue subterm:

```
Lemma addn0 n : n + 0 = n.
```

```
(a + b)^2 = (c + d) + (a + b)
```

We can identify the occurrence of `b` using its context:

```
rewrite -[X in (_ + X)^2]addn0
```

The pattern selects:

```
(a + b)^2 = (c + d) + (a + b)
```

The `X` selects:

```
(a + b)^2 = (c + d) + (a + b)
```

Then the substitution happens exactly there:

```
(a + (b + 0))^2 = (c + d) + (a + b)
```

Extra flags

Rewrite rules can be interleaved with other flags:

- To unfold/fold (local) definitions

  ```
  rewrite /def -/def
  ```

- To simplify (cleanup) the goal or get rid of trivial goals

  ```
  rewrite /= //
  ```

- To iterate 1 or more (or zero or more) times

  ```
  rewrite !lem n?lem
  ```

- To clear unneeded hypotheses

  ```
  rewrite {h}
  ```

Note that `/def` and `/=` can be decorated with patterns to restrict their action to a portion of the goal.

Idiom

Goal readability

Give short names to big expressions:

```
set d := gcd _ _.
```

```
n : nat
m : nat
==========
gcd n m %| n
```

```
n : nat
m : nat
d := gcd n m
==========
d %| n
```

Unfold/fold when needed:

```
rewrite /d
rewrite -/d
```

Idiom

Rules with premises

Consider the goal:

```
 b_gt0 : 0 < b
================
 (a + b) * c %/ (a + b) = c + 0.
```

And the lemmas:

```
mulKn m d : 0 < d -> (d * m) %/ d = m
ltn_addl m n p : m < n -> m < p + n
```

We chain the two rules to kill the side condition:

```
rewrite mulKn ?ltn_addl //.
```

The first rule leaves two active goals:

```
 c = c + 0
 0 < a + b
```

Idiom

Rules with premises

Consider the goal:

```
 b_gt0 : 0 < b
================
 (a + b) * c %/ (a + b) = c + 0.
```

And the lemmas:

```
mulKn m d : 0 < d -> (d * m) %/ d = m
ltn_addl m n p : m < n -> m < p + n
```

We chain the two rules to kill the side condition:

```
rewrite mulKn ?ltn_addl //.
```

The second (optional) rule leaves three active goals:

```
 c = c + 0
 true
 0 < b
```

Idiom

Rules with premises

Consider the goal:

```
 b_gt0 : 0 < b
================
 (a + b) * c %/ (a + b) = c + 0.
```

And the lemmas:

```
mulKn m d : 0 < d -> (d * m) %/ d = m
ltn_addl m n p : m < n -> m < p + n
```

We chain the two rules to kill the side condition:

```
rewrite mulKn ?ltn_addl //.
```

The cleanup switch // leaves only the main goal:

```
 c = c + 0
```

Summary

What you should try to remember

```
move: (t) => /v[ t1 | t2 ].

elim/v: {occ}n n.+1 (ltnSn n) => [ | m IH ] //=.

rewrite lem ?lem !lem [pat]lem [X in pat]lem.
```

Appetizers (for experts, see the manual):

```
rewrite (_ : a = b) [in X in pat]lem -[pat]/def
rewrite [_ a b]lem (lem1,lem2)
rewrite lem in hyp |- *
congr (_ && _)
```