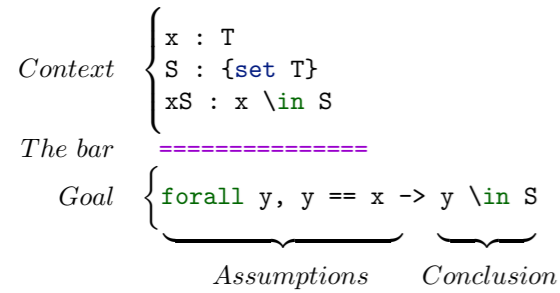


Cheat Sheet

Terminology



Top is the first assumption, y here

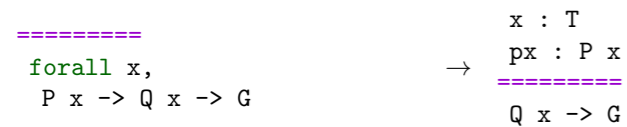
Stack alternative name for the list of *Assumptions*

Popping from the stack

Note: in the following example we assume *cmd* does nothing, exactly like `move`, to focus on the effect of the intro pattern.

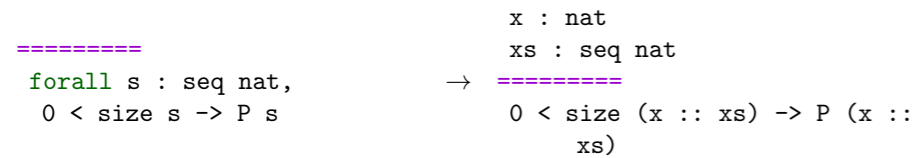
cmd => `x px`

Run *cmd*, then pop **Top**, put it in the context naming it x then pop the new **Top** and names it px in the context



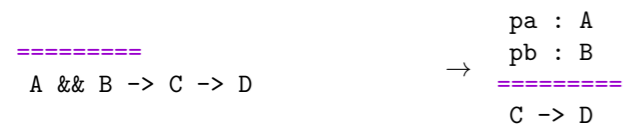
cmd => `[|x xs] //`

Run *cmd*, then reason by cases on **Top**. In the first branch do nothing, in the second one pop two assumptions naming then x and xs . Then get rid of trivial goals. Note that, since only the first branch is trivial, one can write `=> [// | x xs]` too. **caveat**: Immediately after `case` and `elim` it does not perform any case analysis, but can still introduce different names in different branches



cmd => `/andP[pa pb]`

Run *cmd*, then apply the view `andP` to **Top**, then destruct the conjunction and introduce in the context the two parts naming the pa and pb



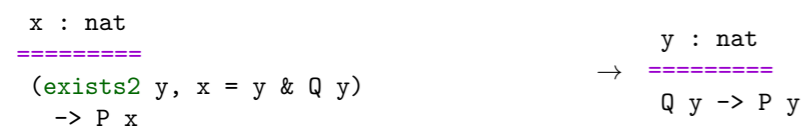
cmd => `/= {px}`

Run *cmd* then simplify the goal then discard px from then context



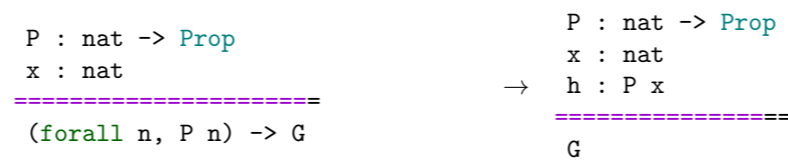
cmd => `[y -> {x}]`

Run *cmd* then destruct the existential, then introduce y , then rewrite with **Top** left to right and discard the equation, then clear x



cmd => `/(_ x) h`

Introduce h specialized to x



Pushing to the stack

Note: in the following *cmd* is not `apply` or `exact`. Moreover we display the goal just before *cmd* is run.

cmd: `(x) y`

Push y then push x on the stack. y is also cleared



cmd: `{-2}x (erefl x)`

Push the type of `(erefl x)`, then push x on the stack binding all but the second occurrence



cmd: `_.+1 {px}`

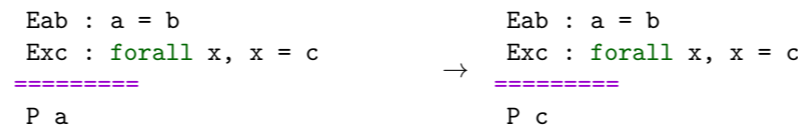
Clear px and generalize the goal with respect to the first match of the pattern `_.+1`



Proof commands

`rewrite Eab (Exc b)`.

Rewrite with Eab left to right, then with Exc by instantiating the first argument with b



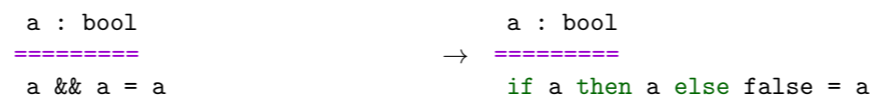
`rewrite -Eab {}Eac`.

Rewrite with Eab right to left then with Eac left to right, finally clear Eac



`rewrite /(_ && _)`.

Unfold the definition of `&&`



`rewrite /= -[a]/(0+a) -/c`.

Simplify the goal, then change a into $0+a$, finally fold back the local definition c



`apply: H`.

Apply H to the current goal



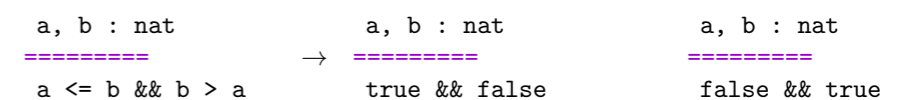
`case: ab`.

Eliminate the conjunction or disjunction



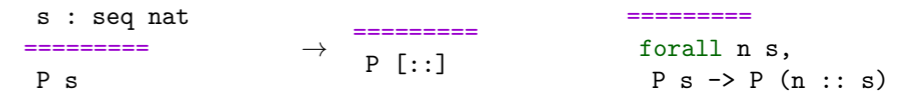
`case: (leqP a b)`.

Reason by cases using the `leqP` spec lemma



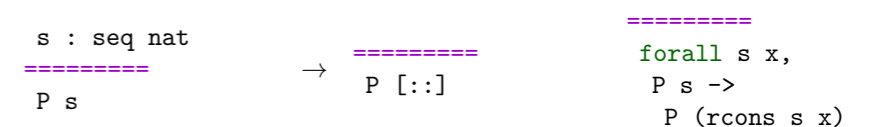
`elim: s`.

Perform an induction on s



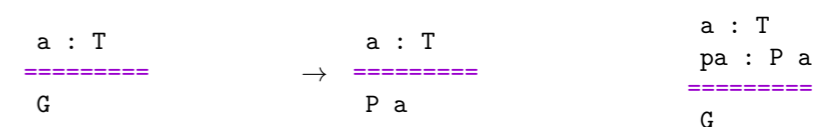
`elim/last_ind: s`

Start an induction on s using the induction principle `last_ind`



`have pa : P a`.

Open a new goal for $P a$. Once resolved introduce a new entry in the context for it named pa



`by []`.

Prove the goal by trivial means, or fail



`exact: H`.

Apply H to the current goal and then assert all remaining goals, if any, are trivial. Equivalent to `by apply: H`.



Reflect and views

reflect P b
States that P is logically equivalent to b

apply: (iffP V)
Proves a reflection goal, applying the view lemma V to the propositional part of **reflect**. E.g. **apply**: (iffP idP)

<pre>P : Prop b : bool ===== reflect P b</pre>	→	<pre>P : Prop b : bool ===== b -> P</pre>	<pre>P : Prop b : bool ===== P -> b</pre>
--	---	--	--

apply/V1/V2
Prove boolean equalities by considering them as logical double implications. The term V1 (resp. V2) is the view lemma applied to the left (resp. right) hand side. E.g. **apply/idP/negP**

<pre>b1 : bool b2 : bool ===== b1 = ~ b2</pre>	→	<pre>b1 : bool b2 : bool ===== b1 -> ~ b2</pre>	<pre>b1 : bool b2 : bool ===== ~ b2 -> b1</pre>
--	---	--	--

rewrite: (eqP Eab)
rewrite with the boolean equality Eab

<pre>Eab : a == b ===== P a</pre>	→	<pre>Eab : a == b ===== P b</pre>
-----------------------------------	---	-----------------------------------

Idioms

case: b => [h1| h2 h3]
Push b, reason by cases, then pop h1 in the first branch and h2 and h3 in the second

have /andP[x /eqP->] : P a && b == c
Open a subgoal for P a && b == c. When proved apply to it the **andP** view, destruct the conjunction, introduce x, apply the view **eqP** to turn b == c into b = c, then rewrite with it and discard the equation

elim: n.+1 {-2}n (ltnSn n)=> {n} // n
General induction over n, note that the first goal has a false assumption **forall** n, n < 0 -> ... and is thus solved by //

<pre>n : nat ===== P n</pre>	→	<pre>n : nat ===== (forall m, m < n -> P m) -> forall m, m < n.+1 -> P m</pre>
------------------------------	---	---

rewrite lem1 ?lem2 //
Use the equation with premises **lem1**, then get rid of the side conditions with **lem2**

Searching

Search _ addn (_ * _) "C" in **ssrnat**
Search for all theorems with no constraints on the main conclusion (conclusion head symbol is the wildcard _), that talk about the **addn** constant, matching anywhere the pattern (_ * _) and having a name containing the string "C" in the module **ssrnat**

Misc notations

```
"f1 \o f2" := (comp f1 f2)
"x \in A" := (in_mem x (mem A))
"x \notin A" := (~ (x \in A))
"[ /\ P1 , P2 & P3 ]" := (and3 P1 P2 P3)
```

```
"[ \/ P1 , P2 | P3 ]" := (or3 P1 P2 P3)
"[ && b1 , b2 , .. , bn & c ]" :=
  (b1 && (b2 && .. (bn && c) .. ))
"[ || b1 , b2 , .. , bn | c ]" :=
  (b1 || (b2 || .. (bn || c) .. ))
"#| A |" := (card (mem A))
"n .-tuple" := (tuple_of n)
"'I_ n" := (ordinal n)
"f1 =1 f2" := (eqfun f1 f2)
"b1 (+) b2" := (addb b1 b2)
```

Notations for natural numbers: nat

```
"n .+1" := (succn n)
"n .-1" := (predn n)
"m + n" := (addn m n)
"m - n" := (subn m n)
"m <= n" := (leq m n)
"m < n" := (m.+1 <= n)
"m <= n <= p" := ((m <= n) && (n <= p))
"m * n" := (muln m n)
"n .*2" := (double n)
"m ^ n" := (expn m n)
"n '! " := (factorial n)
"m %/ d" := (divn m d)
"m %% d" := (modn m d)
"m == n %[mod d]" := (m %% d == n %% d)
"m %| d" := (dvdn m d)
"pi .-nat" := (pnat pi)
```

Notations for lists: seq T

```
"x :: s" := (cons _ x s)
"[ :: ]" := nil
"[ :: x1 ]" := (x1 :: [::])
"[ :: x & s ]" := (x :: s)
"[ :: x1 , x2 , .. , xn & s ]" :=
  (x1 :: x2 :: .. (xn :: s) ..)
"[ :: x1 ; x2 ; .. ; xn ]" :=
  (x1 :: x2 :: .. [:: xn] ..)
"s1 ++ s2" := (cat s1 s2)
```

Notations for iterated operations

```
"\big [ op / idx ]_ i F" :=
"\big [ op / idx ]_ ( i | P ) F" :=
"\big [ op / idx ]_ ( i <- r | P ) F" :=
"\big [ op / idx ]_ ( m <= i < n | P ) F" :=
"\big [ op / idx ]_ ( i < n | P ) F" :=
"\big [ op / idx ]_ ( i \in A | P ) F" :=
"\sum_ i F" :=
"\prod_ i F" :=
"\max_ i F" :=
"\bigcap_ i F" :=
"\bigcup_ i F" :=
```

caveat: in the general form, the iterated operation **op** is displayed in prefix form (not in infix form) **caveat**: the string "big" occurs in every lemma concerning iterated operations

Rewrite patterns

rewrite [pat]lem [in pat2]lem2 [X in pat3]lem3
Rewrite the subterms selected by the pattern **pat** with **lem**. Then in the subterms selected by the pattern **pat2** match the pattern inferred from the left hand side of **lem2** and rewrite the terms selected by it. Last, in the sub terms selected by **pat3** rewrite with **lem3** the sub terms identified by X exactly

rewrite {3}[in X in pat1]lem1
Like in **rewrite** [X in pat1]lem1 but use the pattern inferred from **lem1** to identify the sub terms of X to be rewritten. Of these terms, rewrite only the third one. Example: **rewrite** {3}[in X in f _ X]E.

<pre>E : a = c ===== a + f a (a + a) = f a (a + a) + a</pre>	→	<pre>E : a = c ===== a + f a (a + a) = f a (c + a) + a</pre>
--	---	--

rewrite [e in X in pat1]lem1
Like before, but override the pattern inferred from **lem1** with e

rewrite [e as X in pat1]lem1
Like **rewrite** [X in pat1]lem1 but match **pat1**[X := e] insted of just **pat1**

rewrite /def1 -[pat]/term /=
Unfold all occurrences of **def1**. Then match the goal against **pat** and change all its occurrences into **term** (pure computation). Last simplify the goal

rewrite 3?lem2 // {hyp} => x px
Rewrite from 0 to 3 times with **lem2**, then try to solve with **by** [] all the goals. Finally clear **hyp** and introduce x and px

Pattern matching detailed rules

pattern a term, possibly containing _

key The head symbol of a pattern

The sub terms selected by a pattern:

- the goal is traversed outside in, left to right, looking for verbatim occurrences of the key
- the sub terms whose key matches verbatim are higher order matched (i.e. up to definition unfolding and recursive function computation) against the pattern
- if the matching fails, the next sub term whose key matches is tried
- if the matching succeeds, the sub term is considered to be the (only) instance of the pattern
- the sub terms selected by the pattern are then all the copies of the instance of the pattern
- these copies are searched looking again at the key, and higher order comparing the arguments pairwise

Note: occurrence numbers can be combined with patterns. They refer to the list of sub terms selected by the (last) pattern (i.e. they are processed at the very end).

set n := {2 4}(_ + b)
Put in the context a local definition named n for the second and fourth occurrences of the sub terms selected by the pattern (_ + b)

<pre>===== a + c + (a + b) + (a + b) = a + (a + b) + (0 + a + b) + c</pre>	→	<pre>n := a + b ===== a + c + (a + b) + n = a + (a + b) + n + c</pre>
--	---	---