Caveat: Immediately after `cmd` note: in the following example we assume `cmd` `push` `to the stack` alternative name for the list of `Stack` is the first assumption, `cmd` `pop` `two assumptions naming` then `cmd` `Run` `introduce` in the context the two parts naming the `left` to `right` and discard the equation, then clear `x`.

Cheat Sheet

Terminology

Context

* `x : T`
* `s : (nat T)`
* `x < x.+1`

The bar

Goal

* `forall y, y == x -> y \in S`
* `forall x, x : T`

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` `push` `x`

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

`forall x, x : nat`

`false && true`

`forall s : seq nat, 0 < size s -> P s`

`forall s x : seq nat, 0 < size (x :: xs) -> P (x :: xs)`

`forall s, s : seq nat`

`forall n, P n -> G`

Proving a new goal for `P a`.

Slowly open a new goal for `P a`. Once resolved introduce a new entry in the context for it named `pa`.

Rewrite `Eab (Exc b)`.

Rewrite with `Eab` right to left, then with `Exc` by instantiatiing the first argument with `b`.

Rewrite with `Eab` left to right and discard the equation, then clear `x`.

Unfold the definition of `&&`.

Example:

```
rewrite /[_ x] h
Introduce `h` specialized to `x`

P : nat -> Prop
x : nat
------------------
\[\rightarrow h \colon P x \quad G\]

rewrite /-[a]/(x) b
Rewrite with `b` into `0+a`, finally fold back the local definition `c`.

a, b : nat
a, b : nat
a, b : nat
------------------
c := b + 3 : nat
------------------
true && (a == b + 3)
------------------
0 + a == c
```

Proof commands

Rewrite with `Eab` (Exc)ab.

Rewrite with `Eab` left to right, then with `Eab` by instantiatiing the first argument with `b`.

Rewrite with `Eab` right to left then with `Eab` left to right, finally clear `Exc`.

Rewrite with `Eab` left to right and discard the equation, then clear `x`.

Unfold the definition of `&&`.

```
rewrite /(_ x) h
Introduce `h` specialized to `x`

P : nat -> Prop
x : nat
------------------
\[\rightarrow h \colon P x \quad G\]

rewrite /-[a]/(x) b
Rewrite with `b` into `0+a`, finally fold back the local definition `c`.

a, b : nat
a, b : nat
a, b : nat
------------------
c := b + 3 : nat
------------------
true && (a == b + 3)
------------------
0 + a == c
```

Rewrite with `Eab` (Exc)ab.

Rewrite with `Eab` left to right, then with `Eab` by instantiatiing the first argument with `b`.

Rewrite with `Eab` right to left then with `Eab` left to right, finally clear `Exc`.

Rewrite with `Eab` left to right and discard the equation, then clear `x`.

Unfold the definition of `&&`.

```
rewrite /(_ x) h
Introduce `h` specialized to `x`

P : nat -> Prop
x : nat
------------------
\[\rightarrow h \colon P x \quad G\]
```
Reflect and views
reflect P b
States that P is logically equivalent to b
apply: (ifP V)
Proves a reflection goal, applying the view lemma V to the propositional form of reflect. E.g. apply: (ifP idP)
\[
P : \text{Prop} \quad P : \text{Prop} \quad P : \text{Prop}
\]
\[
b : \text{bool} \quad b : \text{bool} \quad b : \text{bool}
\]

\[
\begin{align*}
\text{reflect pat1 lem1} & \Rightarrow \quad \text{rewrite [in X in pat1]lem1} \\
E : a = c & \Rightarrow \quad a + f a : (a + a) = f a (a + a) + a
\end{align*}
\]

rewite [in X in pat1]lem1
Like before, but override the pattern inferred from lem1 with e
rewite [as X in pat1]lem1
Like rewrite [X in pat1]lem1 but match pat1[X := e] instead of just pat1
rewite /def1 /-pat1/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
rewite ?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:
1. the goal is traversed outside in, left to right, looking for verbatim occurrences of the key
2. the sub terms whose key matches verbatim are higher order matched (i.e. up to pure computation).
3. if the matching fails, the next sub term whose key matches is tried
4. if the matching succeeds, the sub term is considered to be the (only) instance of the pattern
5. the sub terms selected by the pattern are then all the copies of the instance of the pattern
6. 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)

Misc notations
"f1 \ o f2" := (comp f1 f2)
"x \in A" := (\langle x,\in A \rangle)
"\[ A, P1, P2, P3 \] := (andP3 P1 P2 P3)