Basic Cheat Sheet

Rewriting

`rewrite Eab (Exc b).`

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

```
 Eab : a = b                    Eab : a = b
 Exc : forall x, x = c     →    Exc : forall x, x = c
=========                      =========
 P a                            P c
```

`rewrite -Eab {}Eac.`

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

```
 Eab : a = b
 Eac : a = c               →    Eab : a = b
=========                      =========
 P b                            P c
```

`rewrite !addnA.`

Rewrite with `addnA`, associativity of addition, as many times as possible.

```
=========                      =========
a + (b + (c + d))          →   a + b + c + d
```

Reasoning by cases or by induction

`case: n => [|p].`

Reson by cases on `n`, name `p` the predecessor

```
 n : nat                                  p : nat
=========        →    =========          =========
 P n                   P 0                P p.+1
```

`elim: n => [|m IHm].`

Perform an induction on `n`

```
                                          m : nat
 n : nat                                  IHm : P m
=========        →    =========          =========
 P n                   P 0                P m.+1
```

`elim: s => // x xs IHxs.`

Get rid of trivial goals, hence no `[ .. | .. ]`

```
                       x : nat
 s : seq nat           xs : seq nat
=========        →     IHxs : P xs
 P s                  =========
                       P (x :: xs)
```

Naming and processing assumptions

`move=> x /lemma px`

Name the first item `x` then view the top item via `lemma` and name the result `qx`. `lemma` has type `forall a, P a -> Q a`, or `reflect P Q`

```
                                          x : T
   =========                              qx : Q x
.    forall x,                      →    =========
      P x -> R x -> G                     R x -> G
```

`move=> /andP[/eqP-> pb]`

Process the top item with the view `andP`, then destruct the resulting conjunction, use `eqP` on the first item and then rewrite with it, finally name the rest `pb`.

```
a, b : nat                                a, b : nat
=========                                 pb : 10 <= b
(a == 7) && 10 <= b -> a + 3 <=     →    =========
    b                                      7 + 3 <= b
```

`move=> /= {pa}`

Simplify the goal, then clear `pa` from the context

```
a : nat
pa : a != 3                               a : nat
=========                           →    =========
(3 == 7) || (10 <= a)                     (10 <= a)
```

Back and Forward chaining

`apply: H.`

Apply `H` to the current goal

```
 H : A -> B
=========                       →    =========
 B                                    A
```

`apply/subsetP.`

Apply the view `subsetP` to the current goal

```
 A, B : {set T}                 A, B : {set T}
=========                   →  =========
 B \subset A                    forall x, x \in B -> x \in A
```

`have pa : P a.`

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

```
                                          a : T
 a : T                 a : T              pa : P a
=========       →     =========          =========
 G                     P a                G
```

`by [].`

Prove the goal by trivial means, or fail

```
=========
 0 <= n                         →
```