Cheat Sheet 2

Searching in existing libraries

`Search _ (_ + _) "mul" modn.`
Search lemmas whose name contains the string `mul` and whose statement mentions the infix plus operation and contains the constant `modn`. $\boxed{\text{caveat}}$: always put an underscore after `Search`

Specializing assumptions

`move/(_ x):h => h`
Specialise `h` to `x`

```
P : nat -> Prop                P : nat -> Prop
x : nat                        x : nat
h : forall n, P n        →     h : P x
==================             ================
G                              G
```

`case` and `rewrite` under `->` and `forall`

`case` can be applied when head of stack is a formula `C` of the form `A/\B`, `A\/B`, `(forall x:nat,A)`, or `False`, but also when such a formula is buried under a series of implications (goals will be generated accordingly). Example:
`case:h`

```
h : A -> B/\C
=============   →   ============     ======
D                   B -> C -> D      A
```

`rewrite h` can be applied when `h` labels a formula of the form `x=y`, but also when `x=y` is buried under a series of implications (goals will be generated accordingly) and universal quantifiers (attempts at instantiation will be made). Example:
`rewrite h`

```
y : nat
h : forall x, A -> S x = y
===========================     →
P (S 0)

y : nat                  y : nat
h : forall x, A ->       h : forall x, A ->
   S x = y                  S x = y
==================       ===================
P y                      A
```

Definitions, and new uses of `rewrite`

Rewrite works not only on equalities but also equivalences.

`rewrite h0`
Apply equivalence `h0` to goal (left-to-right).

```
h0: A <-> B          h0: A <-> B
============    →    ============
A/\C                 B/\C
```

`Definition name (x : T): type := body`
Add a function `name` with a parameter `x` of type `T`, producing output of type `type` whose definition is `body`

E.g. `Definition double (a:nat):nat := 2*a`
`rewrite /double`
Unfold the definition of `double`

```
=============         =========
double 3 = 6    →     2*3 = 6
```

Notations and control on `rewrite`

`move:Eab => ->`
does the same as `rewrite Eab`

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

`move:Eab => <-`
does the same as `rewrite -Eab`

`rewrite {2}Eab`
rewrites 2nd occurence found (left-to-right)
can be written `move:Eab => {2}->`

```
Eab : a = b          Eab : a = b
============    →    ============
P a /\ Q a           P a /\ Q b
```

`rewrite ?Eab`
rewrites as many times as possible

`rewrite !Eab`
rewrites as many times as possible, at least once

`rewrite n!Eab`
rewrites exactly `n` times

`rewrite n?Eab`
rewrites at most `n` times

The above can be combined:
`rewrite -2?{1}Eab`
rewrites at most 2 times, from right-to-left, the first occurence that is found (each time)
can be written `move:Eab => 2?{1}<-`

Also works with unfolding of definitions
`rewrite {2}/double`
Unfold the 2nd occurence of the definition of `double`

Higher order proof commands & compact syntax for `case`

cmd$_0$ `;` *cmd*$_1$
Run *cmd*$_0$. Then run *cmd*$_1$ on every goals coming from *cmd*$_0$.

cmd$_0$ `;` [*cmd*$_1$|…|*cmd*$_n$]
Run *cmd*$_0$. Then run *cmd*$_1$,…,*cmd*$_n$ respectively on the n goals coming from *cmd*$_0$.

cmd$_0$ `=> [h | x xs]`
does the same as
cmd$_0$ `; case ; [move => h | move => x xs]`

cmd `in hyp1 |- *`
Synonym for `move: hyp1;`*cmd*`; move=> hyp1`. If `|- *` is not given, then the goal is left untouched

`do ?` *cmd*
Repeat *cmd* as many times as possible

`do !` *cmd*
Repeat *cmd* as many times as possible (at least once)

`do n` *cmd*
Repeat *cmd* exactly `n` times

`do n?` *cmd*
Repeat *cmd* at most `n` times

`by` *cmd*
Run *cmd* and then run `done`

cmd$_1$ `; first` *cmd*$_2$
Run *cmd*$_1$. Run *cmd*$_2$ only on the fist resulting goal

cmd `; last first.`
Run *cmd* then reorder the resulting goals putting the last one first

Examples

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

Reason by cases on Top. In the first branch do nothing, in the second one pop two assumptions naming them `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

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

cmd `=> [x Hx | y] ->`

Perform a case analysis on Top. In the first branch push the first two assumptions on the stack naming them `x` and `Hx`; in the second and last branch push Top naming it `y`. Then rewrite with Top left to right and discard the equation

cmd `=> {2}<- // /= hyp`

Rewrite with Top right to left but affect only its second occurrence. Then try to run `done` on every open goal, then simplify (clean up) the statement of the goal. Finally discard `hyp` removing it from the context