Contents

`reverse_naive :: [a] -> [a]`

Naive list reversal

```
Theorem (attempt): reverse (reverse xs) = xs

Proof attempt:

IB: let xs = [], then:

  reverse (reverse [])             { []-case }
= reverse []                       { []-case }
= []

IS: let xs = x : xs', then

  reverse (reverse (x : xs))       { :-case }
= reverse (reverse xs ++ [x])      { :-case }
= ...
= bummer
```

Reversal with accumulator:

`reverse_with_acc :: [a] -> [a] -> [a]`

`reverse :: [a] -> [a]`

Now, we can define `reverse` properly.

- Our goal is still: `reverse (reverse xs) = xs`
- Observe (informally): `reverse_with_acc xs ys = (reverse xs) ++ ys`

- This helps to notice that `rev (rwa xs ys) = rwa ys xs`, which is sufficient, because we can then just take `ys = []`, and then, by definition, and by []-case,

  ```
  rev (reve xs) = rev (rwa xs []) = rwa [] xs = rev xs
  ```

So, let us prove `rev (rwa xs ys) = rwa ys xs`, by induction on `xs`

```
IB: xs = [] -->

  rev (rwa [] ys)
= rev ys                      { []-case for rwa }
= rwa ys []                   { def. of rev }

IS: xs = x : xs' -->

  rev (rwa (x : xs') ys)
= rev (rwa xs' (x : ys))      { :-case for rwa }
= rwa (x : ys) xs'            { IH }
= rwa ys (x : xs')            { :-case for rwa }
```