```
(** * Lecture 02 *)

(** Infer some type arguments automatically. *)
Set Implicit Arguments.

(**

  Note that the type constructor for functions (arrow "->")
  associates to the right:
<<
    A -> B -> C = A -> (B -> C)
>>
*)

Inductive list (A: Type) : Type :=
| nil  : list A
| cons : A -> list A -> list A.

Fixpoint length (A: Type) (l: list A) : nat :=
  match l with
  | nil _ => O
  | cons x xs => S (length xs)
  end.

(**

  So far, Coq will not infer the type argument for [nil]:
<<
Check (cons 1 nil).

Error: The term "nil" has type "forall A : Type, list A"
 while it is expected to have type "list nat".
>>
*)

Check (cons 1 (nil nat)).

(** We can tell Coq to always try though: *)
Arguments nil {A}.

Check (cons 1 nil).

(**
  [countdown] is a useful function for testing.
  [countdown n] produces the list:
<<
  n :: (n - 1) :: (n - 2) :: .. :: 1 :: 0 :: nil
>>
*)
Fixpoint countdown (n: nat) :=
  match n with
  | O => cons n nil
  | S m => cons n (countdown m)
  end.

(**
  We can run our [countdown] function on some example inputs:
*)
Eval cbv in (countdown 0).
(**
<<
     = cons 0 nil
     : list nat
>>
*)
Eval cbv in (countdown 3).
(**
<<
     = cons 3 (cons 2 (cons 1 (cons 0 nil)))
```



```
     : list nat
>>
*)
Eval cbv in (countdown 10).
(**
<<
     = cons 10
       (cons 9
        (cons 8
         (cons 7
          (cons 6
           (cons 5
            (cons 4
             (cons 3
              (cons 2
               (cons 1
                (cons 0 nil))))))))))
     : list nat
>>
*)


(**
  [map] takes a function [f] and list [l] and produces a new
  list by applying [f] to each element of [l].

  (Note that because Gallina is a pure functional programming
  language, the original input list is completely unchanged.)
*)
Fixpoint map (A B: Type) (f: A -> B) (l: list A) : list B :=
  match l with
  | nil => nil
  | cons x xs => cons (f x) (map f xs)
  end.

Eval cbv in (map (plus 1) (countdown 3)).
(**
<<
     = cons 4 (cons 3 (cons 2 (cons 1 nil)))
     : list nat
>>
*)
Eval cbv in (map (fun _ => true) (countdown 3)).
(**
<<
     = cons true (cons true (cons true (cons true nil)))
     : list bool
>>
*)

Definition is_zero (n: nat) : bool :=
  match n with
  | O => true
  | S m => false
  end.

Eval cbv in (map is_zero (countdown 3)).
(**
<<
     = cons false (cons false (cons false (cons true nil)))
     : list bool
>>
*)

Fixpoint is_even (n: nat) : bool :=
  match n with
  | O => true
  | S O => false
  | S (S m) => is_even m
```

```
  end.

Eval cbv in (map is_even (countdown 3)).
(**
<<
     = cons false (cons true (cons false (cons true nil)))
     : list bool
>>
*)

(**
  [map] produces an output list which is exactly the
  same length as its input list.

  (Note that this proof uses bullets (+).
   See the course web page for more info.)
*)
Lemma map_length :
  forall (A B : Type) (f : A -> B) (l : list A),
  length (map f l) = length l.
Proof.
  intros.
  induction l.
  + simpl. reflexivity.
  + simpl.
    (** Replace "length (map f l)" with "length l"  *)
    rewrite IHl.
    reflexivity.
Qed.

(**

  To prove properties about all elements of a type, we typically
  use _induction_.

  We do this by proving that the property holds on the "base cases",
  that is, for nonrecursive constructors.

  For example, [(O : nat)] and [(nil : list A)] are base cases for
  [nat] and [list] respectively.

  Then, we prove that the property is _preserved_ by each of the
  recursive constructors, _assuming_ it holds for all the recursive
  arguments to that constructor.

  To prove the inductive case for a property [P] of nats, we need to prove
<<
    forall n, P n -> P (S n)
>>
  For lists, we need to prove
<<
    forall l x, P l -> P (cons x l)
>>
*)

(**
  Function composition, like from math class.
*)
Definition compose
  (A B C : Type)
  (f : B -> C)
  (g : A -> B)
  : A -> C :=
  fun x => f (g x).

(**
  Mapping two functions one after the other over a list
  is the same as just mapping their composition over the list.
*)
```



```
Lemma map_map_compose:
  forall (A B C : Type)
         (g : A -> B) (f : B -> C) (l : list A),
  map f (map g l) = map (compose f g) l.
Proof.
  intros.
  induction l.
  + simpl. reflexivity.
  + simpl. rewrite IHl.
    (** need to "unfold" [compose] so simpl can grind *)
    unfold compose. reflexivity.
Qed.

(**
  Often we'd like to process a list by "crunching" it
  down into a single value.

  [foldr] does this by taking a function [f], a list
  [cons e1 (cons e2 (cons e3 ... (cons eN nil) ...))],
  and an initial "accumulator" or "state" [b] and computing:
<<
  f e1 (f e2 (f e3 ... (f eN b) ...))
>>
*)
Fixpoint foldr (A B : Type) (f : A -> B -> B)
               (l : list A) (b : B) : B :=
  match l with
  | nil => b
  | cons x xs => f x (foldr f xs b)
  end.

(**
  Again, [foldr] works by putting a function [f] in for each [cons]:
<<
  foldr f (cons 1 (cons 2 (cons 3 nil))) x
  -->
  f 1 (f 2 (f 3 x))
>>
  See how [foldr] replaces [cons] with [f] and [nil] with [x].
*)

(**
  [foldr plus] sums a list of [nat]s.
  Let's sum the values from 0 to 10
*)
Eval cbv in (foldr plus (countdown 10) 0).
(**
<<
     = 55
     : nat
>>
*)

(**
  Consider our good friend the factorial function:
*)
Fixpoint fact (n: nat) : nat :=
  match n with
  | O => 1
  | S m => mult n (fact m)
  end.

Eval cbv in (fact 0).
Eval cbv in (fact 1).
Eval cbv in (fact 2).
Eval cbv in (fact 3).
Eval cbv in (fact 4).

(**
```

```
  We can write it slightly differently using [foldr]:
*)
Definition fact' (n: nat) : nat :=
  match n with
  | O => 1
  | S m => foldr mult (map (plus 1) (countdown m)) 1
  end.

Eval cbv in (fact' 0).
Eval cbv in (fact' 1).
Eval cbv in (fact' 2).
Eval cbv in (fact' 3).
Eval cbv in (fact' 4).

(**
  As an exercise, please prove these two versions of
  factorial equivalent:
*)
Lemma fact_fact':
  forall n,
  fact n = fact' n.
Proof.
  (** challenge problem *)
Admitted.

(**
  It turns out we can write many list function
  just in terms of [foldr].  Here's a definition
  of [map] using [foldr]:
*)
Definition map' (A B : Type)
  (f : A -> B) (l : list A) : list B :=
  foldr (fun x acc => cons (f x) acc) l nil.

(**
  We can prove our "foldr" version of map equivalent
  to the direct definition:
*)
Lemma map_map' :
  forall (A B : Type) (f : A -> B) (l : list A),
  map f l = map' f l.
Proof.
  intros.
  induction l.
  + simpl. unfold map'. simpl. reflexivity.
  + simpl. rewrite IHl.
    (** again, need to unfold so simpl can grind *)
    unfold map'. simpl.
    reflexivity.
(**
  _Note_: this proof is very sensitive
  to the order of rewrite and unfold!
*)
Qed.

(**
  We can also define another flavor of fold, called
  [foldl], that starts applying [f] to the first element
  of the list instead of the last.

  What's the difference?
  When would you use [foldl] instead of [foldr]?
*)
Fixpoint foldl (A B : Type)
  (f : A -> B -> B)
  (l : list A) (b : B) : B :=
  match l with
  | nil => b
  | cons x xs => foldl f xs (f x b)
```



```
  end.

(**
  When working with lists, appending one list onto the
  end of another is very useful.  This [app] function
  does exactly that.  Notice how similar it is to adding
  [nat]s.
*)
Fixpoint app (A : Type)
 (l1 : list A) (l2 : list A) : list A :=
  match l1 with
  | nil => l2
  | cons x xs => cons x (app xs l2)
  end.

Eval cbv in (app (cons 1 (cons 2 nil)) (cons 3 nil)).
(**
<<
     = cons 1 (cons 2 (cons 3 nil))
     : list nat
>>
*)

(**
  This is the analog of our lemma about (n + 0) from
  Lecture 01, but for appending [nil] onto a list.

  Notice how similar the proofs are!  We have seen
  this pattern several times already.
*)
Theorem app_nil:
  forall A (l: list A),
  app l nil = l.
Proof.
  intros.
  induction l.
  + simpl. reflexivity.
  + simpl. rewrite IHl. reflexivity.
Qed.

(**
  [app] is associative, meaning we can freely
  re-associate (move parens around).

  There's the same proof pattern again!
*)
Theorem app_assoc:
  forall A (l1 l2 l3: list A),
  app (app l1 l2) l3 = app l1 (app l2 l3).
Proof.
  intros.
  induction l1.
  + simpl. reflexivity.
  + simpl. rewrite IHl1. reflexivity.
Qed.

(**
  Sometimes a list is "backward" from the order
  we would prefer it in.

  Here is a simple but "inefficient" way to reverse
  a list.
*)
Fixpoint rev (A: Type) (l: list A) : list A :=
  match l with
  | nil => nil
  | cons x xs => app (rev xs) (cons x nil)
  end.
```

```
(**
  We say that the version of [rev] above is "inefficient"
  because it is not _tail recursive_.  A function is tail
  recursive when all recursive calls are the final action
  of the function.  You can read more about tail recursion
  here:
    https://en.wikipedia.org/wiki/Tail_call

  Tail recursion is generally faster and leads to less
  stack space consumption, but it is more complicated
  and therefore often trickier to reason about.

  Below we define a tail recursive function to reverse a list.
  We first define a helper function [fast_rev_aux] which takes
  an additional argument [acc] ("acc" is short for "accumulator").
  We "accumulate" the resulting reversed list with each recursive call.

  Note how [fast_rev_aux] only calls itself in tail position,
  i.e., as its result.

  Tail recursion is typically faster because compilers for
  functional programming languages often perform tail-call
  optimization ("TCO"), in which stack frames are re-used
  by recursive calls.
*)
Fixpoint fast_rev_aux (A : Type)
  (l : list A) (acc : list A) : list A :=
  match l with
  | nil => acc
  | cons x xs => fast_rev_aux xs (cons x acc)
  end.

Definition fast_rev (A : Type)
  (l : list A) : list A :=
  fast_rev_aux l nil.

(**
  We can make sure our faster, tail-recursive version
  of reverse is right by proving it equivalent to the
  simpler, non-tail-recursive version.

  However, as we see below, we will not be able to do
  this directly.  We will first need to prove a helper
  lemma with a _stronger_ induction hypothesis.
*)
Theorem rev_ok:
  forall A (l : list A),
  fast_rev l = rev l.
Proof.
  intros.
  induction l.
  + simpl. (** reduces rev, but does nothing to rev_fast *)
    unfold fast_rev. (** unfold fast_rev to fast_rev_aux *)
    simpl. (** now we can simplify the term *)
    reflexivity.
    (** TIP: if simpl doesn't work, try unfolding! *)
  + unfold fast_rev in *.
    (** this looks like it could be trouble... *)
    simpl. rewrite <- IHl.
    (** STUCK! need to know about the rev_aux accumulator (acc) *)
    (** TIP: if your IH seems weak, try proving something more general *)
Abort.

Lemma fast_rev_aux_ok:
  forall A (l1 l2 : list A),
  fast_rev_aux l1 l2 = app (rev l1) l2.
Proof.
  intros.
  induction l1.
```



```
  + simpl. reflexivity.
  + simpl.
    (** STUCK AGAIN! need to know for *any* l2 *)
    (** TIP: if your IH seems weak, only intro up to the induction variable *)
Abort.

Lemma fast_rev_aux_ok:
  forall A (l1 l2: list A),
  fast_rev_aux l1 l2 = app (rev l1) l2.
Proof.
  intros A l1.
  induction l1.
  + intros. simpl. reflexivity.
  +
(**
  Compare the induction hypothesis (IHl1) here with
  the one we had before.  What's different?  Why is
  this called "generalizing" the induction hypothesis?
*)
    intros. simpl.
    rename l2 into foo.
(**
  Note that we can rewrite by IHl1 even though it is
  universally quantified (i.e., there's a [forall]).
  Coq will figure out what to replace [l2] with
  in [IHl1 (cons a foo)].
*)
    rewrite IHl1. rewrite app_assoc.
    simpl.
    reflexivity.
Qed.

(**
  With our stronger induction hypothesis from the lemma,
  we can now prove [rev_ok] as a special case of [rev_aux_ok].
*)
Lemma rev_ok:
  forall A (l: list A),
  fast_rev l = rev l.
Proof.
  intros.
  unfold fast_rev.
  rewrite fast_rev_aux_ok.
  rewrite app_nil.
  reflexivity.
Qed.

(**

Here we'll stop for an in class exercise!

<<

                                   ~-.
          ,,,;           ~-.~-.~-
         (.../          ~-.~-.~-.~-.~-.
         } o~`,        ~-.~-.~-.~-.~-.~-.
         (/    \     ~-.~-.~-.~-.~-.~-.~-.
          ;    \   ~-.~-.~-.~-.~-.~-.~-.
         ;     {_.~-.~-.~-.~-.~-.~-.~
        ;:  .-~`   ~-.~-.~-.~-.~-.
       ;.: :'    ._   ~-.~-.~-.~-.~-
        ;::`-.    '-._ ~-.~-.~-.~-
         ;::. `-.    '-,~-.~-.~-.
          ';::::.`''-.-'
            ';::;;:,:'
               '||"
               / |
```

```
              ~'~"'

>>
*)

(** add an element to the end of a list *)
Fixpoint snoc (A : Type)
  (l : list A) (x : A) : list A :=
  match l with
  | nil => cons x nil
  | cons y ys => cons y (snoc ys x)
  end.

Theorem snoc_app_singleton :
  forall A (l : list A) (x : A),
  snoc l x = app l (cons x nil).
Proof.
  intros.
  induction l.
  + simpl. reflexivity.
  + simpl. rewrite IHl. reflexivity.
Qed.

Theorem app_snoc_l :
  forall A (l1 : list A) (l2 : list A) (x : A),
  app (snoc l1 x) l2 = app l1 (cons x l2).
Proof.
  intros.
  induction l1.
  + simpl. reflexivity.
  + simpl. rewrite IHl1. reflexivity.
Qed.

Theorem app_snoc_r :
  forall A (l1 : list A) (l2 : list A) (x : A),
  app l1 (snoc l2 x) = snoc (app l1 l2) x.
Proof.
  intros.
  induction l1.
  + simpl. reflexivity.
  + simpl. rewrite IHl1. reflexivity.
Qed.

(** Another simple but inefficient way to reverse a list *)
Fixpoint rev_snoc (A : Type) (l : list A) : list A :=
  match l with
  | nil => nil
  | cons x xs => snoc (rev_snoc xs) x
  end.

Lemma fast_rev_aux_ok_snoc:
  forall A (l1 l2 : list A),
  fast_rev_aux l1 l2 = app (rev_snoc l1) l2.
Proof.
  intros A l1.
  induction l1.
  + intros. simpl. reflexivity.
  + intros. simpl.
    rewrite IHl1.
    rewrite app_snoc_l.
    reflexivity.
Qed.

Lemma fast_rev_ok_snoc:
  forall A (l : list A),
  fast_rev l = rev_snoc l.
Proof.
  intros.
  unfold fast_rev.
```



```
  rewrite fast_rev_aux_ok_snoc.
  rewrite app_nil.
  reflexivity.
Qed.

(**
  We'll finish off with some example lemmas
  about [rev] and [length].

  Note how often the same proof pattern keeps emerging!
*)

Lemma length_app:
  forall A (l1 l2 : list A),
  length (app l1 l2) = plus (length l1) (length l2).
Proof.
  intros.
  induction l1.
  + simpl. reflexivity.
  + simpl. rewrite IHl1. reflexivity.
Qed.

Lemma plus_1_S:
  forall n,
  plus n 1 = S n.
Proof.
  intros.
  induction n.
  + simpl. reflexivity.
  + simpl. rewrite IHn. reflexivity.
Qed.

Lemma rev_length:
  forall A (l: list A),
  length (rev l) = length l.
Proof.
  intros.
  induction l.
  + simpl. reflexivity.
  + simpl. rewrite length_app.
    simpl. rewrite plus_1_S.
    rewrite IHl. reflexivity.
Qed.

Lemma rev_app:
  forall A (l1 l2: list A),
  rev (app l1 l2) = app (rev l2) (rev l1).
Proof.
  intros.
  induction l1.
  + simpl. rewrite app_nil. reflexivity.
  + simpl. rewrite IHl1. rewrite app_assoc.
    reflexivity.
Qed.

Lemma rev_involutive:
  forall A (l: list A),
  rev (rev l) = l.
Proof.
  intros.
  induction l.
  + simpl. reflexivity.
  + simpl. rewrite rev_app.
    simpl. rewrite IHl. reflexivity.
Qed.
```