PROOFS ARE PROGRAMS

Lecture 05

POLYMORPHISM, HIGHER-ORDER-FUNCTIONS AND MORE

(for later: http://etc.ch/gBqD)

Back to Lists

```
Inductive natlist : Type :=
  | nat_nil
  | nat_cons (n : nat) (l : natlist).
```

Back to Lists

```
Inductive natlist : Type :=
  | nat_nil
  | nat_cons (n : nat) (l : natlist).
```

```
Definition mylist := cons 1 (cons 2 (cons 3 nil)).
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1;2;3].
```

Back to Lists

```
Inductive natlist : Type :=
  | nat_nil
  | nat_cons (n : nat) (l : natlist).
```

```
Definition mylist := cons 1 (cons 2 (cons 3 nil)).
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1;2;3].
```

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil => nil
  | h :: t => rev t ++ [h]
  end.
```

Back to Lists

```
Inductive natlist : Type :=
  | nat_nil
  | nat_cons (n : nat) (l : natlist).
```

```
Definition mylist := cons 1 (cons 2 (cons 3 nil)).
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1;2;3].
```

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil => nil
  | h :: t => rev t ++ [h]
  end.
```

```
Theorem nil_app : ∀ l : natlist,
  [] ++ l = l.
Proof. reflexivity. Qed.
```

List of Booleans

List of Booleans

```
Definition mylist := [true; true; false]
```

List of Booleans

```
Definition mylist := [true; true; false]
```

```
Inductive natlist : Type :=
  | nat_nil                    (* natlist *)
  | nat_cons (n : nat) (l : natlist).
           (* nat -> natlist -> natlist *)
```

List of Booleans

```
Definition mylist := [true; true; false]
```

```
Inductive natlist : Type :=
  | nat_nil                  (* natlist *)
  | nat_cons (n : nat) (l : natlist).
            (* nat -> natlist -> natlist *)
```

```
Inductive boollist : Type :=
  | bool_nil                   (* boollist *)
  | bool_cons (b : bool) (l : boollist).
            (* bool -> boollist -> boollist *)
```

Make Lists General Again

Make Lists General Again

```
list : Type -> Type

list (bool) -> boollist
list (nat)  -> natlist
```

Make Lists General Again

```
list : Type -> Type

list (bool) -> boollist
list (nat)  -> natlist
```

Make Lists General Again

```
list : Type -> Type

list (bool) -> boollist
list (nat)  -> natlist
```

```
Inductive list : Type :=
| nil          (* list *)
| cons (x : X) (l : list).
         (* X -> list -> list *)
```

Make Lists General Again

```
list : Type -> Type

list (bool) -> boollist
list (nat)  -> natlist
```

```
Inductive list : Type :=
| nil              (* list *)
| cons (x : X) (l : list).
         (* X -> list -> list *)
```

Where does "X" come from?

Should both lists have the same type?

Make Lists General Again

```
Inductive list : Type :=
  | nil             (* list *)
  | cons (x : X) (l : list).
          (* X -> list -> list *)
```

Make Lists General Again

```
Inductive list : Type :=
  | nil            (* list *)
  | cons (x : X) (l : list).
          (* X -> list -> list *)
```

```
Inductive list : Type :=
  | nil            (* list X *)
  | cons (x : X) (l : list X).
          (* X -> list X -> list X *)
```

Make Lists General Again

```
Inductive list : Type :=
  | nil              (* list *)
  | cons (x : X) (l : list).
            (* X -> list -> list *)
```

```
Inductive list : Type :=
  | nil              (* list X *)
  | cons (x : X) (l : list X).
            (* X -> list X -> list X *)
```

```
Inductive list : Type :=
| nil          (* forall X : Type, list X *)
| cons (x : X) (l : list X).
      (* forall X : Type, X -> list X -> list X *)
```

Make Lists General Again

```
Inductive list : Type :=
  | nil             (* list *)
  | cons (x : X) (l : list).
          (* X -> list -> list *)
```

```
Inductive list : Type :=
  | nil             (* list X *)
  | cons (x : X) (l : list X).
          (* X -> list X -> list X *)
```

```
Inductive list : Type :=
| nil          (* forall X : Type, list X *)
| cons (x : X) (l : list X).
      (* forall X : Type, X -> list X -> list X *)
```

```
Inductive list (X : Type) : Type :=
| nil          (* forall X : Type, list X *)
| cons (x : X) (l : list X).
      (* forall X : Type, X -> list X -> list X *)
```

```
Inductive list (X : Type) : Type :=
| nil
| cons (x : X) (l : list X).
```

```
Inductive list (X : Type) : Type :=
| nil
| cons (x : X) (l : list X).
```

```
Check nil : ∀ X : Type, list X.
Check cons : ∀ X : Type, X -> list X -> list X.
Check (nil nat) : list nat.
Check (cons nat 3 (nil nat)) : list nat.
```

```
Inductive list (X : Type) : Type :=
| nil
| cons (x : X) (l : list X).
```

```
Check nil : ∀ X : Type, list X.

Check cons : ∀ X : Type, X -> list X -> list X.

Check (nil nat) : list nat.

Check (cons nat 3 (nil nat)) : list nat.
```

```
Inductive list (X : Type) : Type :=
| nil
| cons (x : X) (l : list X).
```

```
Check nil : ∀ X : Type, list X.

Check cons : ∀ X : Type, X -> list X -> list X.

Check (nil nat) : list nat.

Check (cons nat 3 (nil nat)) : list nat.
```

```
Inductive list (X : Type) : Type :=
| nil
| cons (x : X) (l : list X).
```

```
Check nil : ∀ X : Type, list X.

Check cons : ∀ X : Type, X -> list X -> list X.

Check (nil nat) : list nat.

Check (cons nat 3 (nil nat)) : list nat.
```

```
Inductive list (X : Type) : Type :=
| nil
| cons (x : X) (l : list X).
```

```
Check nil : ∀ X : Type, list X.

Check cons : ∀ X : Type, X -> list X -> list X.

Check (nil nat) : list nat.

Check (cons nat 3 (nil nat)) : list nat.
```

```
Check cons bool true (cons nat 3 (nil nat)). (* ??? *)
```

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | 0 => nil X
  | S count' => cons X x (repeat X x count')
  end.

Example test_repeat1 :
  repeat nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).

Example test_repeat2 :
  repeat bool false 1 = cons bool false (nil bool).
```

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | 0 => nil X
  | S count' => cons X x (repeat X x count')
  end.

Example test_repeat1 :
  repeat nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).

Example test_repeat2 :
  repeat bool false 1 = cons bool false (nil bool).
```

```
Check repeat. (* ??? *)
```

TYPE ANNOTATION INFERENCE

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=




Fixpoint repeat' X x count : list X :=



    .
```

TYPE ARGUMENT SYNTHESIS

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | O => nil X
  | S count' => cons X x (repeat X x count')
  end.

Fixpoint repeat' X x count : list X :=
  match count with
  | O => nil _
  | S count' => cons _ x (repeat'' _ x count')
  end.
```

HIGHER-ORDER-FUNCTIONS

```
Definition inc3times (n : nat) : nat :=
  S (S (S n)).
```

HIGHER-ORDER-FUNCTIONS

```
Definition inc3times (n : nat) : nat :=
  S (S (S n)).
```

```
Definition doit3times (f : nat -> nat) (n : nat) : nat :=
      f (f (f n)).

Check S : nat -> nat.
Check inc : nat -> nat.
```

HIGHER-ORDER-FUNCTIONS

```
Definition inc3times (n : nat) : nat :=
  S (S (S n)).
```

```
Definition doit3times (f : nat -> nat) (n : nat) : nat :=
      f (f (f n)).

Check S : nat -> nat.
Check inc : nat -> nat.
```

HIGHER-ORDER-FUNCTIONS

```
Definition inc3times (n : nat) : nat :=
  S (S (S n)).
```

```
Definition doit3times (f : nat -> nat) (n : nat) : nat :=
      f (f (f n)).

Check S : nat -> nat.
Check inc : nat -> nat.
```

```
Definition doit3times {X : Type} (f : X->X) (n : X) : X :=
  f (f (f n)).

Example doit3times inc 0 = inc3times 0.
```

ITER

ITER

```
Fixpoint iter {X : Type} (f : X->X) (n : nat) (s: X) : X :=
  match n with
  | O    => s
  | S n' => f(iter f n' s)
  end.
```

ITER

```
Fixpoint iter {X : Type} (f : X->X) (n : nat) (s: X) : X :=
  match n with
  | O    => s
  | S n' => f(iter f n' s)
  end.
```

```
Example iter_test1:
  iter S 3 O = S (S (S O)).
Example iter_test2:
  iter inc 3 O = S (S (S O)).
```

ANONYMOUS FUNCTIONS

```
(fun n => n + 1) (fun n => n * n)
```

ANONYMOUS FUNCTIONS

`(fun n => n + 1) (fun n => n * n)`

```
Example iter_anon_1:
  iter (fun n => n + 1) 3 O = S (S (S O)).
```

COLLECTION-ORIENTED PROGRAMMING

Filter, Map, ...

FILTER

```
list X

+

X -> bool

=>

list X
```

FILTER

`[1, 2, 3, 42, 1337]`

`+`

`(fun x => even x)`

`=>`

[2, 42]

FILTER

```
Fixpoint filter {X:Type} (test: X->bool) (l:list X) :list X :=
  match l with
  | [] => []
  | h :: t =>


  end.
```

FILTER

```
Fixpoint filter {X:Type} (test: X->bool) (l:list X) :list X :=
  match l with
  | [] => []
  | h :: t =>
    if test h then h :: (filter test t)
          else filter test t
  end.
```

MAP

```
[1; 2; 3; 42; 1337]

+

(fun x => x + 1)

=>

[2; 3; 4; 43; 1338]
```

MAP

```
[1; 2; 3; 42; 1337]

+

(fun x => even x)

=>

[false; true; false; true; false]
```

MAP

```
Fixpoint map {X Y : Type} (f : X->Y) (l : list X) : list Y :=
  match l with
  | [] => []
  | h :: t => (f h) :: (map f t)
  end.
```

MAP

```
Fixpoint map {X Y : Type} (f : X->Y) (l : list X) : list Y :=
  match l with
  | [] => []
  | h :: t => (f h) :: (map f t)
  end.
```

```
Example test_map1: map (fun x => plus 3 x) [2;0;2] = [5;3;5].
```

FOLD

```
[1;2;3] --- sum ---> 6
[2;4;6] --- all_even ---> true
```

FOLD

```
[1;2;3] --- sum ---> 6
[2;4;6] --- all_even ---> true


[1;2;3] --- plus_1 ---> [2,3,4]

[true, false, true] --- magic ---> [42, 42, 1337, 42, 42]
```

FOLD

```
[1;2;3] --- sum ---> 6
```

```
[2;4;6] --- all_even ---> true
```

- Input list : `list X`
- Some function : `X -> ?`
- Return value : `Y`

FOLD

```
[1;2;3] --- sum ---> 6

[2;4;6] --- all_even ---> true
```

- Input list : `list X`
- Some function : `X -> ?`
- Return value : `Y`
- Neutral element : `Y`

FOLD

```
(* [1;2;3] --- sum ---> 6 *)
((0 + 1) + 2) + 3
```

FOLD

```
(* [2;4;6] --- all_even --->  true *)
((true && even 2) && even 4) && even 6

fun head old => old && (even head)
```

FOLD

```
(* [2;4;6] --- all_even --->  true *)
((true && even 2) && even 4) && even 6

fun head old => old && (even head)
```

```
Check (fun head old => old && (even head))
  (* nat -> bool -> bool *)
```

FOLD

```
Fixpoint fold {X Y: Type} (f : X->Y->Y) (l : list X) (b : Y)
                  : Y :=
  match l with
  | nil => b
  | h :: t => f h (fold f t b)
  end.
```

FUNCTIONS THAT CONSTRUCT FUNCTIONS

```
Definition constfun {X: Type} (x: X) : nat -> X :=
  fun (k:nat) => x.

Definition ftrue := constfun true.
```

Polymorphism

Higher-Order-Functions

Fold, Map, Filter

Functions that contruct Functions

Polymorphism

```
forall X, X -> list X
```

Higher-Order-Functions

Fold, Map, Filter

Functions that contruct Functions

Polymorphism

```
forall X, X -> list X
```

Higher-Order-Functions

```
Fixpoint fold {X Y: Type} (f : X->Y->Y) (l : list X) (b : Y)
  : Y := ...
```

Fold, Map, Filter

Functions that contruct Functions

Polymorphism

```
forall X, X -> list X
```

Higher-Order-Functions

```
Fixpoint fold {X Y: Type} (f : X->Y->Y) (l : list X) (b : Y)
  : Y := ...
```

Fold, Map, Filter

```
Fixpoint filter {X:Type} (test: X->bool) (l:list X) :list X :=
```

Functions that contruct Functions

Polymorphism

```
forall X, X -> list X
```

Higher-Order-Functions

```
Fixpoint fold {X Y: Type} (f : X->Y->Y) (l : list X) (b : Y)
  : Y := ...
```

Fold, Map, Filter

```
Fixpoint filter {X:Type} (test: X->bool) (l:list X) :list X :=
```

Functions that contruct Functions

```
Definition constfun {X: Type} (x: X) : nat -> X :=
      fun (k:nat) => x.
```