CS 565

Programming Languages (graduate)
Spring 2025

Week 3
Higher-Order Functions, Polymorphism

Lists of Nats


```
Inductive natList : Type :=
  | nil
  | cons (n : nat) (nl : natList).
```

- Consider these functions :

```
Fixpoint mapadd2 (ln : natList) : natList :=
 match ln with
 | nil => nil
 | cons m ln' => cons (m + 2) (mapadd2 l')
 end.

Fixpoint mapadd6 (ln : natList) : natList :=
 match ln with
 | nil => nil
 | cons m ln’ => cons (m + 6) (mapadd6 l')
 end.
```

- Why not refactor the commonality here?

```
Fixpoint mapaddn (n : nat) (ln : natList) : natList :=
 match ln with
 | nil => nil
 | cons m ln’ => cons (m + n) (mapaddn n l')
 end.
```

Generalizing Functions


```
Fixpoint mapincr (ln : natList) : natList :=
  match ln with
  | nil => nil
  | cons m ln' => cons (m + 1) (mapincr l')
  end.

Fixpoint mapdecr (ln : natList) : natList :=
  match ln with
  | nil => nil
  | cons m ln’ => cons (pred m) (mapdecr l')
  end.
```

```
Fixpoint map (f : ?????) (ln : natList) : natList :=
  match ln with
  | nil => nil
  | cons n ln’ => cons (f n) (map f ln')
  end.
```

f is first-class: it can be supplied as an argument to, or returned as a result from, other functions

Function Types

Everything in Coq has a type

```
Check true.          (* : bool *)
Check (negb true).   (* : bool *)
Check mapadd2.       (* : natList -> natList *)
Check S.             (* : nat -> nat *)
```

Functions have arrow types

A -> B: A function from elements of A to elements of B

```
Check map. (* : (nat -> nat) -> natList -> natList *)
```

Higher-Order Functions

This works because **functions** are **first-class values** in Coq:

- Used as function argument
- Can be result of a function

map is a **higher-order** function:

- a function that takes another function as input

```
Check map.(* : (nat -> nat) -> natList -> natList *)
Check (map S).      (* : natList -> natList *)
```

Any function with multiple arguments returns a function:

```
Check plus.          (* : nat -> nat -> nat *)
Check (plus 5).      (* : nat -> nat *)
Check (map (plus 5)). (* : natList -> natList *)
```

Anonymous Functions

Higher-Order functions often take one-off functions:

```
Fixpoint filter (f : nat -> bool) (ln : natList) : natList :=
  match ln with
  l nil => nil
  l cons n ln' => if (f n) then (cons n (filter f ln'))
                           else (filter f ln')
  end.
Definition gtFive (n : nat) : bool := leb 5 n.
Definition filterFive : natList -> natList := filter gtFive.
```

Anonymous functions to the rescue!

Anonymous Functions

Higher-Order functions often take one-off functions:

```
Fixpoint filter (f : nat -> bool) (ln : natList) : natList :=
  match ln with
  | nil => nil
  | cons n ln' => if (f n) then (cons n (filter f ln'))
                           else (filter f ln')
  end.

Definition gtFive (n : nat) : bool := 5 <=? n.

Definition filterFive : natList -> natList := filter (fun n => leb 5 n).
```

Anonymous functions to the rescue!

Ubiquitous: any function whose signature has multiple arguments involves anonymous functions (this is called currying):

`Definition f x y := body` is equivalent to

```
Definition f := (fun x => fun y => body)
```

Maps

Dictionaries are ubiquitous in programming language implementations:

```
Inductive aexp : Type :=
  | ANum (a : nat)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult  (a1 a2 : aexp)
  | AVar (x : string).

Fixpoint aeval (a : aexp) (f : string -> nat) :=
  match a with
  | AVar x => f x
  | …
  end.
```

Total Maps

Standard operations: higher-order functions:

```
Definition map : Type := string -> nat.

Definition lookup (m : map) (x : string) : nat := m x.
Definition empty : map := fun x => 0.

Definition update (m : map) (x : string) (v : nat) : map :=
  fun y => if (eqb_string x y) then v else m y.

Definition example : map := update (update empty “x” 1 ) “y” 2.
```

What is the behavior of m?

```
Definition m : map :=
      update (update (fun y => 42) “x” 7 ) “z” 10.
```

Lists of X

Imagine we need a type for lists of booleans

```
Inductive natList : Type :=
  | nil
  | cons (n : nat) (nl : natList).

Inductive boolList : Type :=
  | bool_nil
  | bool_cons (b : bool) (bl : boolList).
```

What about lists of nats? Lists of lists of nats?
Lists of nat trees? Lists of boolean trees?
Lists of lists of lists of booleans?
Lists of lists of lists of nats? Etc….

Generic Lists

Coq supports **type abstraction** in data type declarations via **type parameters:**

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

list is a function from types to types:

```
Check list.  (* : Type -> Type *)
```

Generic Lists

The constructors of list also have type parameters:

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

In fact, any function can take type parameters making them *polymorphic*

```
Fixpoint filter (X : Type) (f : X -> bool) (ln : list X) : list X :=
  match ln with
  l nil _ => nil X
  l cons _ x l' => if (f x) then (cons X x (filter X f l'))
                           else (filter X f l')
  end.

Eval compute in
    (filter nat (Nat.ltb 10) (cons nat 1 (cons nat 11 (nil nat)))).
```

Generic Functions

Other definitions can take type parameters:

```
Fixpoint filter (X : Type) (f : X -> bool) (ln : list X) : list X :=
  match ln with
  l nil _ => nil X
  l cons _ x l' => if (f x) then (cons X x (filter X f l'))
                            else (filter X f l')
  end.

Eval compute in (filter _ (Nat.ltb 10) (cons _ 1 (cons _ 11 (nil _)))).
   (* = cons nat 11 (nil nat) *)

Fixpoint map (X Y: Type) (f : X -> Y) (ln : list X) : list Y :=
  match ln with
  l nil _ => nil Y
  l cons _ x l' => cons Y (f x) (map X Y f l')
  end.
```

Example

Consider the following definition:

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

What is the type of repeat?

nat -> nat -> list nat

forall (X Y:Type), X -> nat -> list Y

forall (X :Type), X -> nat -> list X

Ill-typed

Type Argument Inference

Coq can usually infer type arguments:

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

Could also have been written:

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

Type Argument Inference

We can tell Coq to always infer type arguments:

```
Arguments nil {X}.
Arguments cons {X} _ _.
Arguments repeat {X} x cnt.
```

We can now write:

```
Definition list123 := cons 1 (cons 2 (cons 3 nil)).
```

Can also do when defining a function or type:

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

Polymorphism

Datatypes with type variables are an example of polymorphic datatypes.

This flavor of polymorphism is called parametric polymorphism (think type *parameters*)– behavior is the *same* for all instantiations

In ad-hoc polymorphism, the implementation of a polymorphic function is selected based on the type of its arguments (type classes)

Other kinds include inclusion polymorphism (subtyping) and row polymorphism (records), in which one datatype can be treated like another

Other PP Approaches

First introduced in ML in 1975

Now found in Standard ML, OCaml, F#, Ada, Haskell, Scala, Julia, …

You've probably seen something similar before:

```
public class Box<T> {
   private T t;

   public void set(T t) { this.t = t; }
   public T get() { return t; }
```

Java Generics

```
template <class T>
class Box {
  private:
    T t;

  public:
    void set(T const&);
    T get() const; };
```

C++ Templates