CS720

Logical Foundations of Computer Science

Lecture 4: polymorphism

Tiago Cogumbreiro

We now have...

- A reasonable understanding of **proof techniques** (through tactics)
- A reasonable understanding of **functional programming** (today's class mostly concludes this part)
- A minimal understanding of **logic programming** (next class)

Why are we learning Coq?

Logical Foundations of CS

This course of CS 720 is divided into two parts:

1. **The first part:** Coq as a workbench to express the logical foundation of CS
2. **The second part:** use this workbench to formalize a programming language
   I will give you other examples of how Coq is being used to formalize CS

Today's class

1. QA about Homework 1 (`Basics.v`) **(no solutions can be discussed!)**
2. QA about `Induction.v` and `Lists.v`
3. Cover `Poly.v`

QA about Homework 1 (Basics.v)

QA about Induction.v and Lists.v

Poly.v

Due Tuesday, September 25, 11:59 EST

Today we will...

- Learn to generalize functions/data types to accept any type
- Learn that Coq is an expression language (functions as data)

Why are we learning this?

- To be able to have interesting data-structures (containers)
- To be able to have reusable/generic definitions

Recall natlist from lecture 3

```
Inductive natlist : Type :=
 | nil : natlist
 | cons : nat → natlist → natlist.
```

> How do we write a list of `bool`s?

Recall natlist from lecture 3

```
Inductive natlist : Type :=
 | nil : natlist
 | cons : nat → natlist → natlist.
```

> How do we write a list of `bool`s?

```
Inductive boollist : Type :=
 | bool_nil : boollist
 | bool_cons : nat → boollist → boollist.
```

> How to migrate the code that targeted `natlist` to `boollist`? What is missing?

Polymorphism

Inductive types can accept (type) parameters (akin to Java/C# generics, and type variables in C++ templates).

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

> What is the type of `list`? How do we print `list`?

Constructors of a polymorphic list

```
Check list.
```

yields

```
list
     : Type → Type
```

> What does `Type → Type` mean? What about the following?

```
Search list.
Check list.
Check nil nat.
Check nil 1.
```

How do we encode the list $[1;\ 2]$?

How do we encode the list $[1;\ 2]$?

```
cons nat 1 (cons nat 2 (nil nat))
```

Implement concatenation

Recall app:

```
Fixpoint app (l1 l2 : natlist) : natlist :=
 match l1 with
 | nil => l2
 | h :: t => h :: (app t l2)
 end.
```

> How do we make app polymorphic?

Implement concatenation

Recall app:

```
Fixpoint app (l1 l2 : natlist) : natlist :=
 match l1 with
 | nil ⇒ l2
 | h :: t ⇒ h :: (app t l2)
 end.
```

> How do we make app polymorphic?

```
Fixpoint app (X:Type) (l1 l2 : list X) : list X :=
 match l1 with
 | nil _ ⇒ l2
 | cons _ h  t ⇒ cons X h (app X t l2)
 end.
```

Implement concatenation

Recall app:

```
Fixpoint app (l1 l2 : natlist) : natlist :=
 match l1 with
 | nil => l2
 | h :: t => h :: (app t l2)
 end.
```

> How do we make app polymorphic?

```
Fixpoint app (X:Type) (l1 l2 : list X) : list X :=
 match l1 with
 | nil _ => l2
 | cons _ h  t => cons X h (app X t l2)
 end.
```

Type inference (1/2)

Coq infer type information:

```
Fixpoint app X l1 l2 :=
 match l1 with
 | nil _ ⇒ l2
 | cons _ h  t ⇒ cons X h (app X t l2)
 end.

Check app.
```

outputs

```
app
     : forall X : Type, list X → list X → list X
```

Type inference (2/2)

```
Fixpoint app X (l1 l2:list X) :=
 match l1 with
 | nil _ => l2
 | cons _ h  t => cons _ h (app _ t l2)
 end.

Check app.
```

```
app
     : forall X : Type, list X -> list X -> list X
```

Let us look at the output of

```
Compute cons nat 1 (cons nat 2 (nil nat)).
Compute cons _ 1 (cons _ 2 (nil _)).
```

Type information redundancy

- If Coq can infer the type, can we automate inference of type parameters?

Type information redundancy

> If Coq can infer the type, can we automate inference of type parameters?

```
Fixpoint app {X:Type} (l1 l2:list X) : list X :=
 match l1 with
 | nil ⇒ l2
 | cons h  t ⇒ cons h (app t l2)
 end.
```

Alternatively, use `Arguments` after a definition:

```
Arguments nil {X}.        (* braces should surround argument being inferred *)
Arguments cons {_} _ _.   (* you may omit the names of the arguments *)
Arguments app {X} l1 l2.  (* if the argument has a name, you *must* use the *same* name *)
```

Try the following

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

Arguments nil {_}.
Arguments cons {X} x y.

Search list.
Check list.
Check nil nat.
Compute nil nat.
```

> What went wrong?

Try the following

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

Arguments nil {_}.
Arguments cons {X} x y.

Search list.
Check list.
Check nil nat.
Compute nil nat.
```

> What went wrong? How do we supply type parameters when they are being automatically inferred?

Try the following

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

Arguments nil {_}.
Arguments cons {X} x y.

Search list.
Check list.
Check nil nat.
Compute nil nat.
```

> What went wrong? How do we supply type parameters when they are being automatically inferred?

Prefix a definition with `@`. Example: `@nil nat.`

Recall natprod and fst (lec 3)

```
Inductive natprod : Type :=
| pair : nat → nat → natprod.
Notation "( x , y )" := (pair x y).
```

> How do we make `pair` polymorphic with implicit type arguments?

Recall natprod and fst (lec 3)

```
Inductive natprod : Type :=
| pair : nat → nat → natprod.
Notation "( x , y )" := (pair x y).
```

> How do we make `pair` polymorphic with implicit type arguments?

```
 Inductive prod (X Y : Type) : Type :=
| pair : X → Y → prod X Y.
Arguments pair {_} {_}.
Notation "( x , y )" := (pair x y).

Definition fst {X Y:Type} (p : prod X Y) : X :=
  match p with
  | pair x y ⇒ x
  end.
```

> Should we make the arguments of `prod` implicit? Why?

Recall natprod

```
Theorem surjective_pairing : forall (p : natprod),
  p = (fst p, snd p).
```

> How does polymorphism affect our theorems? What about the proof?

Recall natprod

```
Theorem surjective_pairing : forall (p : natprod),
  p = (fst p, snd p).
```

> How does polymorphism affect our theorems? What about the proof?

```
Theorem surjective_pairing : forall (X Y:Type) (p : prod X Y),
  p = (fst p, snd p).
```

Low impact in proofs (usually, `intros`).

Recall indexof (lecture 3)

How do we make this function polymorphic?

```
Fixpoint indexof (n:nat) (l:natlist) : natoption :=
  match l with
  | nil ⇒ None
  | h :: t ⇒
    match beq_nat h n with
    | true ⇒ Some 0          (* element found at the head *)
    | false ⇒
      match indexof n t with  (* check for error *)
      | Some i ⇒ Some (S i)  (* increment successful result *)
      | None ⇒ None          (* propagate error *)
      end
    end
  end.
```

Higher-order functions

```
Require Import Coq.Lists.List. Import ListNotations.

  Fixpoint indexof {X:Type} (beq: X → X → bool) (v:X) (l:list X) : option nat :=
   match l with
   | nil ⇒ None
   | cons h t ⇒
    match beq h v with
     | true ⇒ Some 0          (* element found at the head *)
     | false ⇒
       match indexof beq v t with  (* check for error *)
       | Some i ⇒ Some (S i)  (* increment successful result *)
       | None ⇒ None          (* propagate error *)
       end
     end
   end.
(* A couple of unit tests to ensure indexof is behaving as expected. *)
Goal indexof beq_nat 20 [10; 20; 30] = Some 1. Proof. reflexivity. Qed.
Goal indexof beq_nat 100 [10; 20; 30] = None. Proof. reflexivity. Qed.
```

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.
```

> What is the type of this function?

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.
```

What is the type of this function?

```
forall X: Type → (X → bool) → list X → list → X
```

What does this function do?

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.
```

> What is the type of this function?

`forall X: Type → (X → bool) → list X → list → X`

> What does this function do?

Retains all elements that succeed `test`.

How do we use filter?

▌ What if we want to retain 1 and 3? How do we do this?

```
filter ??? [10; 1; 3; 4]
```

How do we use filter?

> What if we want to retain 1 and 3? How do we do this?

```
filter ??? [10; 1; 3; 4]
```

Answer 1:

```
Definition keep_1_3 (n:nat) : bool :=
match n with
| 1 => true
| 3 => true
| _ => false
end.
(* Assert that the output makes sense: *)
Goal filter keep_1_3 [10; 1; 3; 4] = [1; 3].
Proof.
  reflexivity.
Qed.
```

Revisit keep_1_3

```
Definition keep_1_3 (n:nat) : bool :=
  match n with
  | 1 => true
  | 3 => true
  | _ => false
  end.
```

> Can we rewrite `keep_1_3` by only using `beq_nat` and `orb`?

Revisit keep_1_3

```
Definition keep_1_3 (n:nat) : bool :=
  match n with
  | 1 ⇒ true
  | 3 ⇒ true
  | _ ⇒ false
  end.
```

> Can we rewrite `keep_1_3` by only using `beq_nat` and `orb`?

```
Open Scope bool. (* ensure the || operator is loaded *)

Definition keep_1_3_v2 (n:nat) : bool :=
  beq_nat 1 n || beq_nat 3 n.
```

Anonymous functions

- Are we ever going to use `keep_1_3` again?

```
Definition keep_1_3_v2 (n:nat) : bool :=
  beq_nat 1 n || beq_nat 3 n.

Compute filter keep_1_3_v2 [10; 1; 3; 4].
```

Anonymous functions

Are we ever going to use `keep_1_3` again?

```
Definition keep_1_3_v2 (n:nat) : bool :=
  beq_nat 1 n || beq_nat 3 n.

Compute filter keep_1_3_v2 [10; 1; 3; 4].
```

If you are not, consider using anonymous functions:

```
Goal filter (fun (n:nat) : nat => beq_nat 1 n || beq_nat 3 n) [10; 1; 3; 4] = [1; 3].
Proof.
  reflexivity.
Qed.
```

Anonymous functions are helpful as one-shoot usages (like anonymous classes in Java and C#).

Currying

Let us retain only 3's

With an anonymous function:

```
Goal filter (fun n ⇒ match n with | 3 ⇒ true | _ ⇒ false) [10; 1; 3; 4] = [3].
Proof.
  reflexivity.
Qed.
```

> What about `Check (beq_nat 3)`? Coq is an expression-based language, so `beq_nat 3` is an expression, as is `beq_nat` and `beq_nat 3 10`. What is the type of each expression?

Currying

Let us retain only 3's

With an anonymous function:

```
Goal filter (fun n ⇒ match n with | 3 ⇒ true | _ ⇒ false) [10; 1; 3; 4] = [3].
Proof.
  reflexivity.
Qed.
```

> What about `Check (beq_nat 3)`? Coq is an expression-based language, so `beq_nat 3` is an expression, as is `beq_nat` and `beq_nat 3 10`. What is the type of each expression?

```
Goal filter (beq_nat 3) [10; 1; 3; 4] = [1; 3]. (* filter all elements that are equal to 3 *
Proof.
  reflexivity.
Qed.
```

What we learned...

`Poly.v`

- New capability: types as (function/data) arguments
- New capability: type inference (omit types and let Coq guess the type)
- New syntax: braces `{}` and `Arguments` for type variable inference (*implicit* arguments)
- New syntax: `@` makes all type arguments *explicit*
- New syntax: `fun` declares anonymous functions
- New capability: currying (function calls with argument missing yields a *function*)

(No new tactics.)

Next class: read Tactics.v