Introduction to the Theory of Computation
Lecture 5: Polymorphism; constructor injectivity, explosion principle
Tiago Cogumbreiro
HW1 so far...

- 18 students have **not** submitted their homework (~40%)
- >70% with at least 50 points (C)
- >40% have at least 80 points (B)
- 6 submissions are failing (<50 points)
Today we will learn about...

- Type polymorphism (types in parameters)
- Applying (using) theorems
- Rewriting rules with pre-conditions
- Applying theorems with pre-conditions
- Disjoint constructors
- Principle of explosion
Polymorphism
Recall natlist

Inductive natlist : Type :=
| nil : natlist
| cons : nat -> natlist -> natlist.

How do we write a list of bools?
Recall natlist

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

How do we write a list of bools?

```
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).

\[
\text{Inductive list } (X:\text{Type}) : \text{Type} := \\
\mid \text{nil} : \text{list } X \\
\mid \text{cons} : X \rightarrow \text{list } X \rightarrow \text{list } X.
\]

What is the type of list? How do we print list?
Constructors of a polymorphic list

Check list.
yields

list
  : Type \rightarrow Type

What does Type \rightarrow 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

```coq
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

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.

What is the type of app?
Implement concatenation

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.

What is the type of app? forall X : Type, list X -> list X -> list X
Type inference (1/2)

Coq infer type information:

```coq
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?

```coq
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:

```coq
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.
Exercise 1: transitivity over equals

Theorem eq_trans : forall (T:Type) (x y z : T),
   x = y -> y = z -> x = z.

Proof.
  intros T x y z eq1 eq2.
  rewrite eq1.

yields
  1 subgoal
  T : Type
  x, y, z : T
  eq1 : x = y
  eq2 : y = z

_______________________________________(1/1)
  y = z

How do we conclude this proof?
Exercise 1: transitivity over equals

Theorem eq_trans : \forall (T:Type) (x y z : T),
  x = y → y = z → x = z.
Proof.
  intros T x y z eq1 eq2.
  rewrite > eq1.
 yields
  1 subgoal
  T : Type
  x, y, z : T
  eq1 : x = y
  eq2 : y = z
  ______________________________(1/1)
  y = z

How do we conclude this proof? Yes, rewrite → eq2. reflexivity. works.
Exercise 1: introducing `apply`

Apply takes an hypothesis/lemma to conclude the goal.

```
  apply eq2.
Qed.
```

apply takes $?X$ to conclude a goal $?X$ (resolves foralls in the hypothesis).

1 subgoal
T : Type
x, y, z : T
eq1 : x = y
eq2 : y = z

---------(1/1)
y = z
Applying conditional hypothesis

apply uses an hypothesis/theorem of format $H_1 \rightarrow \ldots \rightarrow H_n \rightarrow G$, then solves goal $G$, and produces new goals $H_1, \ldots, H_n$.

**Theorem** eq_trans_2 : forall (T:Type) (x y z: T),

- $(x = y \rightarrow y = z \rightarrow x = z) \rightarrow (*)$  (*eq1*)
- $x = y \rightarrow$  (*eq2*)
- $y = z \rightarrow$  (*eq3*)
- $x = z$.

**Proof.**

- intros T x y z eq1 eq2 eq3.
- apply eq1.  (* $x = y \rightarrow y = z \rightarrow x = z$ *)

*(Done in class.)*
Rewriting conditional hypothesis

apply uses an hypothesis/theorem of format $H_1 \rightarrow \ldots \rightarrow H_n \rightarrow G$, then solves goal $G$, and produces new goals $H_1, \ldots, H_n$.

**Theorem** eq_trans_3 : forall (T:Type) (x y z: T),

$(x = y \rightarrow y = z \rightarrow x = z) \rightarrow (* \text{eq1} *)$

$x = y \rightarrow (* \text{eq2} *)$

$y = z \rightarrow (* \text{eq3} *)$

$x = z$

**Proof.**

`intros T x y z eq1 eq2 eq3.`

`rewrite \rightarrow eq1. (* x = y \rightarrow y = z \rightarrow x = z *)`

*(Done in class.)*

- Notice that there are 2 conditions in eq1, so we get 3 goals to solve.
Recap

What's the difference between reflexivity, rewrite, and apply?

1. reflexivity solves goals that can be simplified as an equality like \( ?X = ?X \)

2. rewrite \( \rightarrow \) \( H \) takes an hypothesis \( H \) of type \( H_1 \rightarrow \ldots \rightarrow H_n \rightarrow ?X = ?Y \), finds any sub-term of the goal that matches \( ?X \) and replaces it by \( ?Y \); it also produces goals \( H_1, \ldots, H_n \).

   rewrite does not care about what your goal is, just that the goal must contain a pattern \( ?X \).

3. apply \( \rightarrow \) \( H \) takes an hypothesis \( H \) of type \( H_1 \rightarrow \ldots \rightarrow H_n \rightarrow G \) and solves goal \( G \); it creates goals \( H_1, \ldots, H_n \).
Apply with/Rewrite with

Theorem eq_trans_nat : forall (x y z: nat),
  x = 1 \rightarrow
  x = y \rightarrow
  y = z \rightarrow
  z = 1.
Proof.
  intros x y z eq1 eq2 eq3.
  assert (eq4: x = z). {
  apply eq_trans.
outputs
Possible error message: Unable to find an instance for the variable y.
We can supply the missing arguments using the keyword with: apply eq_trans with (y:=y).

Can we solve the same theorem but use rewrite instead?
Symmetry

What about this exercise?

**Theorem** eq_trans_nat : \(\forall (x \ y \ z: \text{nat}),\)
\[x = 1 \rightarrow\]
\[x = y \rightarrow\]
\[y = z \rightarrow\]
\[1 = z.\]

**Proof.**
\begin{verbatim}
intros x y z eq1 eq2 eq3.
assert (eq4: x = z).
\end{verbatim}
Symmetry

What about this exercise?

**Theorem** eq_trans_nat : \( \forall (x \ y \ z : \text{nat}), \)
\[
x = 1 \rightarrow \\
x = y \rightarrow \\
y = z \rightarrow \\
1 = z.
\]

**Proof.**
\[
\text{intros } x \ y \ z \ \text{eq1 eq2 eq3}. \\
\text{assert } (\text{eq4: } x = z). \}
\]

We can rewrite a goal \(?X = ?Y\) into \(?Y = ?X\) with symmetry.
Apply in example

Theorem silly3' : forall (n : nat),
  (beq_nat n 5 = true -> beq_nat (S (S n)) 7 = true) ->
  true = beq_nat n 5 ->
  true = beq_nat (S (S n)) 7.

Proof.
intros n eq H.
symmetry in H.
apply eq in H.

(Done in class.)
Targetting hypothesis

- rewrite $\rightarrow$ H1 in H2
- symmetry in H
- apply H1 in H2
Forward vs backward reasoning

If we have a theorem $L : C_1 \rightarrow C_2 \rightarrow G$:

- **Goal takes last**: apply to goal of type $G$ and replaces $G$ by $C_1$ and $C_2$
- **Assumption takes first**: apply to hypothesis $L$ to an hypothesis $H : C_1$ and rewrites $H : C_2 \rightarrow G$

Proof styles:

- **Forward reasoning**: (apply in hypothesis) manipulate the hypothesis until we reach a goal. **Standard in math textbooks.**
- **Backward reasoning**: (apply to goal) manipulate the goal until you reach a state where you can apply the hypothesis. **Idiomatic in Coq.**

CS420 ☽ Polymorphism; constructor injectivity, explosion principle ☽ Lecture 5 ☽ Tiago Cogumbreiro
Recall our encoding of natural numbers

\[
\text{Inductive } \text{nat} : \text{Type} := \\
| \text{O} : \text{nat} \\
| \text{S} : \text{nat} \rightarrow \text{nat}.
\]

1. Does the equation \( S \ n = 0 \) hold? Why?
Recall our encoding of natural numbers

```latex
Inductive nat : Type :=
  | 0 : nat
  | S : nat -> nat.
```

1. Does the equation $S \ n = 0$ hold? Why?
   No the constructors are implicitly disjoint.

2. If $S \ n = S \ m$, can we conclude something about the relation between $n$ and $m$?
Recall our encoding of natural numbers

```
Inductive nat : Type :=
  | 0 : nat
  | S : nat → nat.
```

1. Does the equation \( S \ n = 0 \) hold? Why?
   **No the constructors are implicitly disjoint.**

2. If \( S \ n = S \ m \), can we conclude something about the relation between \( n \) and \( m \)?
   **Yes, constructor \( S \) is injective. That is, if \( S \ n = S \ m \), then \( n = m \) holds.**

These two principles are available to all inductive definitions! How do we use these two properties in a proof?
Proving that $S$ is injective (1/2)

**Theorem** $S$-injective : \( \forall (n \ m : \text{nat}), \n S \ n = S \ m \rightarrow \n n = m. \)

**Proof.**

```
intros n m eq1.
inversion eq1.
```

If we run `inversion`, we get:

1 subgoal
n, m : nat
eq1 : S n = S m
H0 : n = m

`-------------------------------(1/1)
m = m`
Injectivity in constructors

**Theorem** $S$-injective : \forall (n m : \text{nat}),
\[ S\ n = S\ m \rightarrow n = m. \]

**Proof.**

\begin{verbatim}
intros n m eq1.
inversion eq1 as [eq2].
\end{verbatim}

If you want to name the generated hypothesis you must figure out the destruction pattern and use as [...]. For instance, if we run `inversion eq1 as [eq2]`, we get:

1 subgoal
n, m : \text{nat}
eq1 : S\ n = S\ m
eq2 : n = m

\[ \text{-------------------------------}(1/1) \]

m = m
Disjoint constructors

Theorem beq_nat_0_l : forall n,
   beq_nat 0 n = true → n = 0.
Proof.
   intros n eq1.
   destruct n.
(To do in class.)
Principle of explosion

Ex falso (sequitur) quodlibet

inversion concludes absurd hypothesis, where there is an equality between different constructors. Use inversion eq1 to conclude the proof below.

1 subgoal
n : nat
eq1 : false = true
____________________________________(__)
S n = 0