

Polymorphic `list` can be thought of as what?

- A function from `Types` to `Inductive definitions`.
- A function from `Types` to `Types`.

How are polymorphic types written?

With the type parameters following the bare name of the datatype.

E.g., `list nat` or `list X`.

What are the types of the *polymorphic constructors* `nil` and `cons`?

Check nil.

==> nil : forall X : Type, list X

Check cons.

==> cons : forall X : Type, X -> list X -> list X

How should `forall x` be read in the type of a polymorphic constructor?

As an additional argument to the constructor that determines the expected types of the arguments that follow.



With respect to polymorphic inductive types,  
Coq will automatically infer what?

What won't Coq automatically infer?

Coq will automatically attempt to infer the type of an arguments, even if that type is `Type`.

However, Coq won't automatically attempt to infer *arguments*, for example to the type parameters of a polymorphic constructor.

When no inference has been requested,  
where do you write type argument in a  
constructor invocation?

How about in a constructor that is a case of a  
match on a polymorphic datatype?

You provide the type argument when invoking a polymorphic constructor.

You do not provide the type argument to a polymorphic constructors in a match on a polymorphic datatype. This is because the polymorphic parameter belongs to the datatype as a whole, not to its constructors.

Underscores can replace what kinds of type information?

- They can replace type annotations. Of course, you could also simply omit the annotation altogether.
- They can replace type arguments demanded by polymorphic constructors.

How does Coq determine what to fill in for the implicit argument?

Coq attempts to *unify* all locally available information -- the type of the function being applied, the types of the other arguments, and the type expected by the context in which the application appears -- to determine what concrete type should replace the underscore.



How can type argument inference be requested by a user?

At constructor/function invocation provide the *implicit argument*:

```
length bool someBoolList  
length _ someBoolList
```

For the remainder of the module, works on functions or inductive constructors:

```
Implicit Arguments length [[X]].
```

How can type argument inference be requested at definition site?

This does not work on ...

On function declaration site use curlies, which avoids even the need for underscores on use-site:

```
Fixpoint length {X:Type} (l:list X) : nat :=  
  ...
```

... inductive definitions.

What is the limit of implicit arguments?

Even after an `Implicit Argument` statement there will be cases Coq cannot figure out the right argument. In those cases you can use `@` and provide the type argument explicitly or intentionally omit it without Coq attempting to fill it in.

```
(* This fails if inference has been requested. *)  
(* Check nil. *)  
Check @nil.  
Check @nil nat.
```

What's the difference between  $(x, y)$  and  $(X * Y)$  ?

$(x, y)$  is a 2-tuple *value*, using special notation.

$(X * Y)$  is its *type*, also using special notation.

It just so happens that  $x : X$  and  $y : Y$ .



What is `type_scope`?

The scope used when Coq is parsing symbols. Notations for types, like  $X * Y$  will be declared in `type_scope`, in this case to avoid clash with multiplication.

Matching on multiple arguments is actually  
what?

Matching on a single product argument, with the parens omitted.

Which way does the type arrow associate?

Right.

The following two are equivalent:

$\text{nat} \rightarrow \text{nat} \rightarrow$

$\text{nat} \rightarrow (\text{nat} \rightarrow \text{nat})$

What is the partial application syntax?

Simply omit arguments in an invocation.



What are the types of the currying and uncurrying a function?

## Currying:

`forall X Y Z : Type, (X * Y -> Z) -> X -> Y -> Z`

## Uncurrying:

`forall X Y Z : Type, (X -> Y -> Z) -> X * Y -> Z`

Give the syntax of anonymous functions.

```
fun arg1 ... argN => expr
```

What is the type of the `override` function?

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

Describe the `unfold` and `fold` tactics.

`unfold` replaces a term with its definition, "expanding" functions.

`fold` takes the expanded term and packs it back up into a definition. It is much less used.



What does it mean to say that constructors in Coq are *disjoint*?

Values built from distinct constructors are never equal.

What does it mean to say that constructors in Coq are *injective*?

The only way for two values produced by a constructor to be equal is for them to have been constructed with the same values as functions.

Describe the `inversion` tactic.

Given a hypothesis  $H$  in context or previously proven:

$c\ a_1\ \dots\ a_n = d\ b_1\ \dots\ b_n$

`inversion H` will conclude either:

- If  $c$  and  $d$  are the same constructor then by the inversion property  $a_i = b_i$ . Those facts will be added to the context and will be used to rewrite the goal.

- If  $c$  and  $d$  are different then by disjointness property we have reached contradiction so the current goal is proved, since *any* goal is now provable.

`inversion` can act on what?

Equations involving simple constructor values or even complex terms like lists.



How can tactics be applied to hypothesis instead of the goal?

Use the `in` keyword.

For example:

```
simpl in H
```

```
apply L in H
```

Describe the difference between `apply` and `apply...in`.

apply L in H uses forward reasoning, matching H against the premise of L, replacing H with the conclusion of L.

apply L however matches the current goal against the *conclusion* of L, ending the current goal and creating new goals for each premise of L.

Explain the difference between *forward* and *backward* reasoning.

Forward reasoning is just modus ponens:

Given  $L1$  and  $L1 \rightarrow L2$  conclude  $L2$ .

Backward reasoning starts with the goal and reasons about what would imply the goal until a premise or previous theorem is reached.

Given  $L1 \rightarrow L2$  and the *goal*  $L2$ , consider if we can establish  $L1$ .

What happens if you `destruct` on an expression instead of a value?

The cases will be equations with the expression and its possible values, one per constructor.

```
destruct (beq_nat n 3) .  
  Case "beq_nat n 3 = true". ...  
  Case "beq_nat n 3 = false". ...
```



Explain the `remember` tactic.

`remember (expr) as name.`

`name` will be added to context along with a new assumption `name=expr`. Further, instances of `expr` will be replaced with `name`.

What is a common use-case for `remember`?

The substitution done by `destruct` (i.e., substituting instances of the destructed expression with the current possible value) eradicates information needed for the proof.

Explain the `apply ... with ... tactic`.

Sometimes when you do `apply H` Coq is not able to bind every variable in `H` to the goal, perhaps because some variables don't apply in the conclusion of `H`, only in the premises.

This tactic provides a way to explicitly give Coq bindings between `H` and the goal.

```
apply trans_eq with (m:=[c,d]).
```

That binds the variable `m` to the list `[c,d]`.