Polymorphic inductive definitions can be thought of as what?

Functions from `Type`s to `Inductive` definitions. Or, since we always use `Inductive` definitions to create types, then a function from `Types` to `Types`.

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
```

The `forall` portion should be read as an additional argument to the constructor.

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

What won't Coq automatically infer?

Coq will automatically attempt to infer the types of arguments, even when those types are polymorphic.

However, Coq won't automatically attempt to infer *arguments*, for example the `Type` argument to a polymorphic constructor.

How can we avoid writing type arguments in an invocation?

Throughout a module?

At declaration site?

At constructor/function invocation:

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

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

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

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

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

How can you use explicit type arguments after having requested inference?Use `@`.

```
(* This fails if inference has been requested.*)
(* Definition mynil := nil nat *)
Definition mynil := @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`.

Which way does the type arrow associate?

Right.

The following two are equivalent:

```
nat -> nat ->
nat -> (nat -> nat)
```

What are the type signatures of currying and uncurrying a function?

Currying:

```
(A * B) -> C
A -> B -> C
```

Uncurrying:

```
A -> B -> C
(A * B) -> C
```

Give the syntax of anonymous functions.

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

Define a function that overrides other functions whose domain is `nat`s.

Give its type.

```
Definition override {X:Type} (f: nat =>X) (k:nat) (x:X) : nat->X :=
  fun(k':nat) => if beq_nat k k' then x else f k'.

===> override : forall X : Type, (nat -> X) -> nat -> X -> nat -> X
```

Describe the `unfold` and `fold` tactics.

`unfold` replaces a term with its definitions, "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:

```
c a1 ... an = d b1 ... bn
```

`inversion H` will conclude either:

- If `c` and `d` are the same constructor then by the inversion property `ai = bi`. 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.

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

Use the `in` keyword.

For example:

```
simpl in H
apply L in H
```

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

Forward reasoning is just modus ponens:
Given `L1` and `L1->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->L2` and the *goal* `L2`, consider it proved if `L1` is already known.

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

The cases will be equations with the expression and its possible values.

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

Explain the `remember` tactic.

?

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

?