

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

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.

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.

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 curlyes, which avoids even the need for underscores on use-site:

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

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

(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 $y : Y$.

Right.

The following two are equivalent:

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

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

Currying:

$$(A * B) \rightarrow C$$

$$A \rightarrow B \rightarrow C$$

Uncurrying:

$$A \rightarrow B \rightarrow C$$

$$(A * B) \rightarrow C$$


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

```
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
```

`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.

Values built from distinct constructors are never equal.

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.

Given a hypothesis H in context:

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

`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.

Use the `in` keyword.

For example:

```
simpl in H
```

```
apply L in H
```

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 it proved if $L1$ is already known.

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". ...
```

?

?