Tipos Inductivos 2

- **El tipo igualdad**

```
  Inductive eq (A : Type) (x : A) : A -> Prop :=  refl_equal : x = x
For eq: Argument A is implicit
For refl_equal: Argument A is implicit
For eq: Argument scopes are [type_scope _ _]
For refl_equal: Argument scopes are [type_scope _]

eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
       P x -> forall y : A, x = y -> P y
eq_rec : forall (A : Type) (x : A) (P : A -> Set),
       P x -> forall y : A, x = y -> P y
```

El definido en el propio sistema `(eq x y)` se abrevia `x=y`.

```
eq : forall A : Type, A -> A -> Prop
```

- **El tipo existencial** En lógica constructiva una prueba de la proposición

$$\exists x : A \cdot P(x)$$

consiste en un par (a, p) donde a es de tipo A y p es una prueba de (Pa).

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> ex P
For ex: Argument A is implicit
For ex_intro: Argument A is implicit
For ex: Argument scopes are [type_scope _]
For ex_intro: Argument scopes are [type_scope _ _ _]
```

```
ex : forall A : Type, (A -> Prop) -> Prop
ex_intro : forall (A : Type) (P : A -> Prop) (x : A), P x -> ex P

Coq < Check fun (A:Set) (P:A->Prop) (x:A)
                    (p :(P x)) => (ex_intro P x p).
fun (A : Set) (P : A -> Prop) (x : A) (p : P x) => ex_intro P x p
     : forall (A : Set) (P : A -> Prop) (x : A), P x -> ex P

Coq < Check fun (A:Set) (P:A->Prop) (x:A)
                    (p :(P x)) => (ex_intro (fun a:A =>(P a)) x p).
fun (A : Set) (P : A -> Prop) (x : A) (p : P x) =>
ex_intro (fun a : A => P a) x p
     : forall (A : Set) (P : A -> Prop) (x : A), P x -> exists a : A, P a

ex_intro:  forall (A : Type) (P : A -> Prop) (x : A), P x -> ex P

ex_ind : forall (A : Type) (P : A -> Prop) (P0 : Prop),
       (forall x : A, P x -> P0) -> ex P -> P0
```

la expresión `Check forall P:nat->Prop, exists a:nat,(P a).` nos da

$$\forall P : nat \rightarrow Prop, (\exists a : nat, P\, a$$

- **La versión constructiva del tipo existencial** Dados $A : Set$ y un predicado $P : A \rightarrow Prop$, podemos construir el tipo

$$\{x \in A | (P\ x)\}$$

de tipo Set cuyos posibles elementos son los pares (x, p) donde, como en el caso anterior, $x : A$ representa el "testigo" y $p : (P\ x)$ es la justificación de que x satisface el predicado P.

```
Inductive sig (A : Set) (P : A -> Prop) : Set :=
    exist : forall x : A, P x -> sig P
For sig: Argument A is implicit
For exist: Argument A is implicit
For sig: Argument scopes are [type_scope type_scope]
For exist: Argument scopes are [type_scope _ _ _]

sig : forall A : Set, (A -> Prop) -> Set

Coq < Check forall (A:Set) (P:A->Prop), (sig P).
forall (A : Set) (P : A -> Prop), sig P
     : Type

Coq < Check forall (A:Set) (P:A->Prop), (sig (fun a:A=>(P a))).
forall (A : Set) (P : A -> Prop), {a : A | P a}
     : Type

exist : forall (A : Set) (P : A -> Prop) (x : A), P x -> sig P

sig_ind : forall (A : Set) (P : A -> Prop) (P0 : sig P -> Prop),
       (forall (x : A) (p : P x), P0 (exist P x p)) -> forall s : sig P, P0 s
sig_rec : forall (A : Set) (P : A -> Prop) (P0 : sig P -> Set),
       (forall (x : A) (p : P x), P0 (exist P x p)) -> forall s : sig P, P0 s
```

- **La definición mutuamente inductiva de naturales pares e impares**

```
 Inductive
    Even : nat -> Prop :=
      | even0 : Even 0
      | evenS : forall n : nat, Odd n -> Even (S n)
    with Odd : nat -> Prop :=
      | odd1 : Odd 1
      | oddS : forall n : nat, Even n -> Odd (S n).
Even, Odd are defined
Even_ind is defined
```

```
Odd_ind is defined
```

- **El tipo del predicado binario que expresa que un número r es la suma de m y un fijo n**

```
 Inductive
    Plus (n : nat) : nat -> nat -> Prop :=
      | Plus0 : Plus n 0 n
      | PlusS : forall m r : nat, Plus n m r -> Plus n (S m) (S r).
Plus is defined
Plus_ind is defined


Coq < Check Plus.
Plus: nat -> nat -> nat -> Prop

Coq < Check Plus0.
Plus0: forall n : nat, Plus n 0 n
Coq < Check PlusS.
PlusS:forall n m r : nat, Plus n m r -> Plus n (S m) (S r)
```

- **El tipo mutuamente inductivo de trees y forests**

```
   Inductive
    tree : Set :=
        node : forest -> tree
    with forest : Set :=
      | emptyf : forest
      | consf : tree -> forest -> forest.
tree, forest are defined
tree_rect is defined
tree_ind is defined
tree_rec is defined
forest_rect is defined
forest_ind is defined
```


```
forest_rec is defined

 Check tree.
 Set
 Check forest.
 Set
 Check node.
 forest->tree
 Check emptyf.
 forest
 Check consf.
 tree->forest->forest
```