

# Demostración asistida por ordenador con Coq

José A. Alonso Jiménez

---

Grupo de Lógica Computacional

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Sevilla, 31 de julio de 2018 (versión del 20 de agosto de 2018)

Esta obra está bajo una licencia Reconocimiento-NoComercial-CompartirIgual 2.5 Spain de Creative Commons.

**Se permite:**

- copiar, distribuir y comunicar públicamente la obra
- hacer obras derivadas

**Bajo las condiciones siguientes:**

**Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor.



**No comercial.** No puede utilizar esta obra para fines comerciales.



**Compartir bajo la misma licencia.** Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.

Esto es un resumen del texto legal (la licencia completa). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-sa/2.5/es/> o envíe una carta a Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

# Índice general

1 Programación funcional y métodos elementales de demostración en Coq	7
2 Demostraciones por inducción sobre los números naturales en Coq	35
3 Datos estructurados en Coq	57
4 Polimorfismo y funciones de orden superior en Coq	97
5 Tácticas básicas de Coq	133
6 Lógica en Coq	195



# Introducción

En este libro se incluye unos apuntes de demostración asistida por ordenador con [Coq](#) para los cursos de

- [Razonamiento automático](#) del [Máster Universitario en Lógica, computación e inteligencia artificial](#) de la [Universidad de Sevilla](#).
- [Lógica matemática y fundamentos](#) del [Grado en Matemáticas](#) de la [Universidad de Sevilla](#).

Esencialmente los apuntes son una adaptación del libro [Software foundations \(Vol. 1: Logical foundations\)](#) de Benjamin Peirce y otros.

Una primera versión de estos apuntes se han usado este año en el [Seminario de Lógica Computacional](#).

## Cuaderno de bitácora

En esta sección se registran los cambios realizados en las sucesivas versiones del libro.

### Versión del 12 de agosto de 2018

Se ha añadido el capítulo 5 (Tácticas básicas de Coq).

### Versión del 20 de agosto de 2018

Se ha añadido el capítulo 6 (Lógica en Coq).



# Tema 1

## Programación funcional y métodos elementales de demostración en Coq

*(\* T1: Programación funcional y métodos elementales de demostración en Coq \*)*

*(\* El contenido de la teoría es*

*1. Datos y funciones*

*1. Tipos enumerados*

*2. Booleanos*

*3. Tipos de las funciones*

*4. Tipos compuestos*

*5. Módulos*

*6. Números naturales*

*2. Métodos elementales de demostración*

*1. Demostraciones por simplificación*

*2. Demostraciones por reescritura*

*3. Demostraciones por análisis de casos \*)*

*(\* =====  
§ 1. Datos y funciones  
===== \*)*

*(\* =====  
§§ 1.1. Tipos enumerados  
===== \*)*

*(\* -----*

*Ejemplo 1.1.1. Definir el tipo dia cuyos constructores sean los días de la semana.*

----- \*)

**Inductive** dia: **Type** :=

```
| lunes      : dia
| martes     : dia
| miercoles  : dia
| jueves     : dia
| viernes    : dia
| sabado     : dia
| domingo    : dia.
```

(\* -----  
*Ejemplo 1.1.2. Definir la función  
 siguiente\_laborable : dia -> dia  
 tal que (siguiente\_laborable d) es el día laboral siguiente a d.*  
 ----- \*)

**Definition** siguiente\_laborable (d:dia) : dia:=

**match** d **with**

```
| lunes      => martes
| martes     => miercoles
| miercoles  => jueves
| jueves     => viernes
| viernes    => lunes
| sabado     => lunes
| domingo    => lunes
```

**end.**

(\* -----  
*Ejemplo 1.1.3. Calcular el valor de las siguientes expresiones  
 + siguiente\_laborable jueves  
 + siguiente\_laborable viernes  
 + siguiente\_laborable (siguiente\_laborable sabado)*  
 ----- \*)

Compute (siguiente\_laborable jueves).

(\* ==> viernes : dia \*)



```

Compute (siguiente_laborable viernes).
(* ==> lunes : dia *)

```

```

Compute (siguiente_laborable (siguiente_laborable sabado)).
(* ==> martes : dia *)

```

```

(* -----
   Ejemplo 1.1.4. Demostrar que
       siguiente_laborable (siguiente_laborable sabado) = martes
   ----- *)

```

```

Example siguiente_laborable1:
  siguiente_laborable (siguiente_laborable sabado) = martes.

```

```

Proof.
  simpl.      (* ⊢ martes = martes *)
  reflexivity. (* ⊢ *)

```

```

Qed.

```

```

(* =====
   §§ 1.2. Booleanos
   ===== *)

```

```

(* -----
   Ejemplo 1.2.1. Definir el tipo bool (□) cuyos constructores son true
   y false.
   ----- *)

```

```

Inductive bool : Type :=
| true  : bool
| false : bool.

```

```

(* -----
   Ejemplo 1.2.2. Definir la función
       negacion : bool -> bool
   tal que (negacion b) es la negacion de b.
   ----- *)

```

```

Definition negacion (b:bool) : bool :=
  match b with
  | true  => false

```

```
| false => true
end.
```

```
(* -----
   Ejemplo 1.2.3. Definir la función
       conjuncion : bool -> bool -> bool
   tal que (conjuncion b1 b2) es la conjunción de b1 y b2.
   ----- *)
```

```
Definition conjuncion (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true  => b2
  | false => false
  end.
```

```
(* -----
   Ejemplo 1.2.4. Definir la función
       disyuncion : bool -> bool -> bool
   tal que (disyuncion b1 b2) es la disyunción de b1 y b2.
   ----- *)
```

```
Definition disyuncion (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true  => true
  | false => b2
  end.
```

```
(* -----
   Ejemplo 1.2.5. Demostrar las siguientes propiedades
       disyuncion true  false = true.
       disyuncion false false = false.
       disyuncion false true  = true.
       disyuncion true  true  = true.
   ----- *)
```

```
Example disyuncion1: disyuncion true false = true.
Proof. simpl. reflexivity. Qed.
```

```
Example disyuncion2: disyuncion false false = false.
Proof. simpl. reflexivity. Qed.
```

**Example** disyuncion3: disyuncion false true = true.

**Proof.** simpl. reflexivity. Qed.

**Example** disyuncion4: disyuncion true true = true.

**Proof.** simpl. reflexivity. Qed.

```
(* -----
  Ejemplo 1.2.6. Definir los operadores (&&) y (||) como abreviaturas
  de las funciones conjuncion y disyuncion.
  ----- *)
```

**Notation** "x && y" := (conjuncion x y).

**Notation** "x || y" := (disyuncion x y).

```
(* -----
  Ejemplo 1.2.7. Demostrar que
    false || false || true = true.
  ----- *)
```

**Example** disyuncion5: false || false || true = true.

**Proof.** simpl. reflexivity. Qed.

```
(* -----
  Ejercicio 1.2.1. Definir la función
    nand : bool -> bool -> bool
  tal que (nand x y) se verifica si x e y no son verdaderos.

  Demostrar las siguientes propiedades de nand
    nand true false = true.
    nand false false = true.
    nand false true = true.
    nand true true = false.
  ----- *)
```

**Definition** nand (b1:bool) (b2:bool) : bool :=  
negacion (b1 && b2).

**Example** nand1: nand true false = true.

**Proof.** simpl. reflexivity. Qed.

**Example** nand2: nand false false = true.

**Proof.** simpl. reflexivity. Qed.

**Example** nand3: nand false true = true.

**Proof.** simpl. reflexivity. Qed.

**Example** nand4: nand true true = false.

**Proof.** simpl. reflexivity. Qed.

```
(* -----
  Ejercicio 1.2.2. Definir la función
    conjuncion3 : bool -> bool -> bool -> bool
  tal que (conjuncion3 x y z) se verifica si x, y y z son verdaderos.

  Demostrar las siguientes propiedades de conjuncion3
    conjuncion3 true  true  true  = true.
    conjuncion3 false true  true  = false.
    conjuncion3 true  false true  = false.
    conjuncion3 true  true  false = false.
  ----- *)
```

**Definition** conjuncion3 (b1:bool) (b2:bool) (b3:bool) : bool :=  
b1 && b2 && b3.

**Example** conjuncion3a: conjuncion3 true true true = true.

**Proof.** simpl. reflexivity. Qed.

**Example** conjuncion3b: conjuncion3 false true true = false.

**Proof.** simpl. reflexivity. Qed.

**Example** conjuncion3c: conjuncion3 true false true = false.

**Proof.** simpl. reflexivity. Qed.

**Example** conjuncion3d: conjuncion3 true true false = false.

**Proof.** simpl. reflexivity. Qed.

```
(* =====
  §§ 1.3. Tipos de las funciones
  ===== *)
```

```
(* -----
  Ejemplo 1.3.1. Calcular el tipo de las siguientes expresiones
    + true
    + (negacion true)
    + negacion
  ----- *)
```

**Check true.**

```
(* ==> true : bool *)
```

**Check (negacion true).**

```
(* ==> negacion true : bool *)
```

**Check negacion.**

```
(* ==> negacion : bool -> bool *)
```

```
(* =====
  §§ 1.4. Tipos compuestos
  ===== *)
```

```
(* -----
  Ejemplo 1.4.1. Definir el tipo rva cuyos constructores son rojo, verde
  y azul.
  ----- *)
```

**Inductive rva : Type :=**

```
| rojo   : rva
| verde  : rva
| azul   : rva.
```

```
(* -----
  Ejemplo 1.4.2. Definir el tipo color cuyos constructores son negro,
  blanco y primario, donde primario es una función de rva en color.
  ----- *)
```

**Inductive color : Type :=**

```
| negro   : color
| blanco  : color
| primario : rva -> color.
```

```
(* -----
Ejemplo 1.4.3. Definir la función
    monocromático : color -> bool
tal que (monocromático c) se verifica si c es monocromático.
----- *)
```

```
Definition monocromático (c : color) : bool :=
match c with
| negro      => true
| blanco     => true
| primario p => false
end.
```

```
(* -----
Ejemplo 1.4.4. Definir la función
    esRojo : color -> bool
tal que (esRojo c) se verifica si c es rojo.
----- *)
```

```
Definition esRojo (c : color) : bool :=
match c with
| negro      => false
| blanco     => false
| primario rojo => true
| primario _  => false
end.
```

```
(* =====
§§ 1.5. Módulos
===== *)
```

```
(* -----
Ejemplo 1.5.1. Iniciar el módulo Naturales.
----- *)
```

```
Module Naturales.
```

```
(* =====
§§ 1.6. Números naturales
===== *)
```

```

===== *)

(* -----
   Ejemplo 1.6.1. Definir el tipo nat de los números naturales con los
   constructores 0 (para el 0) y S (para el siguiente).
   ----- *)

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

(* -----
   Ejemplo 1.6.2. Definir la función
   pred : nat -> nat
   tal que (pred n) es el predecesor de n.
   ----- *)

Definition pred (n : nat) : nat :=
  match n with
  | 0    => 0
  | S n' => n'
  end.

(* -----
   Ejemplo 1.6.3. Finalizar el módulo Naturales.
   ----- *)

End Naturales.

(* -----
   Ejemplo 1.6.4. Calcular el tipo y valor de la expresión
   (S (S (S (S 0)))).
   ----- *)

Check (S (S (S (S 0)))).
(* ==> 4 : nat *)

(* -----
   Ejemplo 1.6.5. Definir la función
   menosDos : nat -> nat

```

*tal que (menosDos n) es n-2.*

----- \*)

**Definition** menosDos (n : nat) : nat :=

**match** n **with**

| 0 => 0

| S 0 => 0

| S (S n') => n'

**end.**

(\* -----  
*Ejemplo 1.6.6. Evaluar la expresión (menosDos 4).*  
 ----- \*)

Compute (menosDos 4).

(\* ==> 2 : nat \*)

(\* -----  
*Ejemplo 1.6.7. Calcular et tipo de las funcionse S, pred y menosDos.*  
 ----- \*)

**Check** S.

(\* ==> S : nat -> nat \*)

**Check** pred.

(\* ==> pred : nat -> nat \*)

**Check** menosDos.

(\* ==> menosDos : nat -> nat \*)

(\* -----  
*Ejemplo 1.6.8. Definir la función*  
*esPar : nat -> bool*  
*tal que (esPar n) se verifica si n es par.*  
 ----- \*)

**Fixpoint** esPar (n:nat) : bool :=

**match** n **with**

| 0 => true

| S 0 => false



```
| S (S n') => esPar n'
end.
```

```
(* -----
  Ejemplo 1.6.9. Definir la función
    esImpar : nat -> bool
  tal que (esImpar n) se verifica si n es impar.
  ----- *)
```

**Definition** esImpar (n:nat) : bool :=  
negacion (esPar n).

```
(* -----
  Ejemplo 1.6.10. Demostrar que
    + esImpar 1 = true.
    + esImpar 4 = false.
  ----- *)
```

**Example** esImpar1: esImpar 1 = true.

**Proof.** simpl. reflexivity. Qed.

**Example** esImpar2: esImpar 4 = false.

**Proof.** simpl. reflexivity. Qed.

```
(* -----
  Ejemplo 1.6.12. Iniciar el módulo Naturales2.
  ----- *)
```

```
(* Module Naturales2. *)
```

```
(* -----
  Ejemplo 1.6.13. Definir la función
    suma : nat -> nat -> nat
  tal que (suma n m) es la suma de n y m. Por ejemplo,
    suma 3 2 = 5

  Nota: Es equivalente a la predefinida plus
  ----- *)
```

**Fixpoint** suma (n : nat) (m : nat) : nat :=

```

match n with
  | 0    => m
  | S n' => S (suma n' m)
end.

```

Compute (suma 3 2).

(\* ==> 5: nat \*)

```

(* -----
   Ejemplo 1.6.14. Definir la función
       producto : nat -> nat -> nat
   tal que (producto n m) es el producto de n y m. Por ejemplo,
       producto 3 2 = 6

   Nota: Es equivalente a la predefinida mult.
   ----- *)

```

```

Fixpoint producto (n m : nat) : nat :=
  match n with
  | 0    => 0
  | S n' => suma m (producto n' m)
end.

```

**Example** productol: (producto 2 3) = 6.

**Proof.** **simpl.** **reflexivity.** **Qed.**

```

(* -----
   Ejemplo 1.6.15. Definir la función
       resta : nat -> nat -> nat
   tal que (resta n m) es la diferencia de n y m. Por ejemplo,
       resta 3 2 = 1

   Nota: Es equivalente a la predefinida minus.
   ----- *)

```

```

Fixpoint resta (n m: nat) : nat :=
  match (n, m) with
  | (0 , _)    => 0
  | (S _ , 0)   => n
  | (S n' , S m') => resta n' m'

```

**end.**

```
(* -----
   Ejemplo 1.6.16. Cerrar el módulo Naturales2.
   ----- *)
```

```
(* End Naturales2. *)
```

```
(* -----
   Ejemplo 1.6.17. Definir la función
       potencia : nat -> nat -> nat
   tal que (potencia x n) es la potencia n-ésima de x. Por ejemplo,
       potencia 2 3 = 8

   Nota: En lugar de producto, usar la predefinida mult.
   ----- *)
```

```
Fixpoint potencia (x n : nat) : nat :=
  match n with
  | 0    => S 0
  | S m => mult x (potencia x m)
end.
```

Compute (potencia 2 3).

```
(* ==> 8 : nat *)
```

```
(* -----
   Ejercicio 1.6.1. Definir la función
       factorial : nat -> nat
   tal que (factorial n) es el factorial de n.
       factorial 3 = 6.
       factorial 5 = mult 10 12
   ----- *)
```

```
Fixpoint factorial (n:nat) : nat :=
  match n with
  | 0    => 1
  | S n' => S n' * factorial n'
end.
```

**Example** prop factorial1: factorial 3 = 6.

Proof. simpl. reflexivity. Qed.

**Example** prop factorial2: factorial 5 = mult 10 12.

**Proof.** `simpl. reflexivity. Qed.`

```
(* -----
Ejemplo 1.6.18. Definir los operadores +, - y * como abreviaturas de
las funciones plus, rminus y mult.
----- *)
```

```

Notation "x + y" := (plus x y)
                        (at level 50, left associativity)
                        : nat_scope.

```

```

Notation "x - y" := (minus x y)
                        (at level 50, left associativity)
                        : nat scope.

```

```

Notation "x * y" := (mult x y)
                        (at level 40, left associativity)
                        : nat_scope.

```

```
(* -----
Ejemplo 1.6.19. Definir la función
    iguales_nat : nat -> nat -> bool
tal que (iguales_nat n m) se verifica si n y m son iguales.
----- *)
```

```
Fixpoint iguales_nat (n m : nat) : bool :=
  match n with
  | 0 => match m with
        | 0      => true
        | S m'   => false
      end
  | S n' => match m with
            | 0      => false
            | S m'   => iguales_nat n' m'
          end
  end.
```

( \* -----

*Ejemplo 1.6.20. Definir la función*

*menor\_o\_igual : nat -> nat -> bool*  
*tal que (menor\_o\_igual n m) se verifica si n es menor o igual que m.*  
 ----- \*)

```
Fixpoint menor_o_igual (n m : nat) : bool :=
match n with
| 0    => true
| S n' => match m with
      | 0    => false
      | S m' => menor_o_igual n' m'
end
end.
```

(\* -----  
*Ejemplo 1.6.21. Demostrar las siguientes propiedades*  
 + menor\_o\_igual 2 2 = true.  
 + menor\_o\_igual 2 4 = true.  
 + menor\_o\_igual 4 2 = false.  
 ----- \*)

**Example** menor\_o\_igual1: menor\_o\_igual 2 2 = true.

**Proof.** simpl. reflexivity. Qed.

**Example** menor\_o\_igual2: menor\_o\_igual 2 4 = true.

**Proof.** simpl. reflexivity. Qed.

**Example** menor\_o\_igual3: menor\_o\_igual 4 2 = false.

**Proof.** simpl. reflexivity. Qed.

(\* -----  
*Ejercicio 1.6.2. Definir la función*  
*menor\_nat : nat -> nat -> bool*  
*tal que (menor\_nat n m) se verifica si n es menor que m.*  
  
*Demostrar las siguientes propiedades*  
 menor\_nat 2 2 = false.  
 menor\_nat 2 4 = true.  
 menor\_nat 4 2 = false.  
 ----- \*)

**Definition** menor\_nat (n m : nat) : bool :=  
negacion (iguales\_nat (m-n) 0).

**Example** menor\_nat1: (menor\_nat 2 2) = false.

**Proof.** simpl. reflexivity. Qed.

**Example** menor\_nat2: (menor\_nat 2 4) = true.

**Proof.** simpl. reflexivity. Qed.

**Example** menor\_nat3: (menor\_nat 4 2) = false.

**Proof.** simpl. reflexivity. Qed.

```
(* =====
  § 2. Métodos elementales de demostración
  ===== *)
```

```
(* =====
  § 2.1. Demostraciones por simplificación
  ===== *)
```

```
(* -----
  Ejemplo 2.1.1. Demostrar que el 0 es el elemento neutro por la
  izquierda de la suma de los números naturales.
  ----- *)
```

(\* 1ª demostración \*)

**Theorem** suma\_0\_n : forall n : nat, 0 + n = n.

**Proof.**

```
  intros n.      (* n : nat
                  =====
                  0 + n = n *)
  simpl.         (* n = n *)
  reflexivity.
```

**Qed.**

(\* 2ª demostración \*)

**Theorem** suma\_0\_n' : forall n : nat, 0 + n = n.

**Proof.**

```
  intros n.      (* n : nat
```

```

=====
0 + n = n *)
reflexivity.
Qed.

(* -----
   Ejemplo 2.1.2. Demostrar que la suma de 1 y n es el siguiente de n.
   ----- *)

Theorem suma_1_l : forall n:nat, 1 + n = S n.
Proof.
  intros n.      (* n : nat
                  =====
                  1 + n = S n *)
  simpl.          (* S n = S n *)
  reflexivity.
Qed.

Theorem suma_1_l' : forall n:nat, 1 + n = S n.
Proof.
  intros n.
  reflexivity.
Qed.

(* -----
   Ejemplo 2.1.3. Demostrar que el producto de 0 por n es 0.
   ----- *)

Theorem producto_0_l : forall n:nat, 0 * n = 0.
Proof.
  intros n.      (* n : nat
                  =====
                  0 * n = 0 *)
  simpl.          (* 0 = 0 *)
  reflexivity.
Qed.

(* =====
   § 2.2. Demostraciones por reescritura
   ===== *)

```

```
(* -----
  Ejemplo 2.2.1. Demostrar que si  $n = m$ , entonces  $n + n = m + m$ .
  ----- *)
```

**Theorem** suma\_iguales : forall n m: nat,  
 n = m ->  
 n + n = m + m.

**Proof.**

```
intros n m. (* n : nat
              m : nat
              =====
              n = m -> n + n = m + m *)

intros H. (* n : nat
            m : nat
            H : n = m
            =====
            n + n = m + m *)

rewrite H. (* m + m = m + m *)
reflexivity.
```

**Qed.**

```
(* -----
  Ejercicio 2.2.1. Demostrar que si  $n = m$  y  $m = o$ , entonces
   $n + m = m + o$ .
  ----- *)
```

**Theorem** suma\_iguales\_ejercicio : forall n m o : nat,  
 n = m -> m = o -> n + m = m + o.

**Proof.**

```
intros n m o H1 H2. (* n : nat
                       m : nat
                       o : nat
                       H1 : n = m
                       H2 : m = o
                       =====
                       n + m = m + o *)

rewrite H1. (* m + m = m + o *)
rewrite H2. (* o + o = o + o *)
```



**reflexivity.**  
**Qed.**

```
(* -----
   Ejemplo 2.2.2. Demostrar que  $(0 + n) * m = n * m$ .
   ----- *)
```

**Theorem** producto\_0\_mas : **forall** n m : **nat**,  
 $(0 + n) * m = n * m$ .

**Proof.**

```
intros n m.          (* n : nat
                      m : nat
                      =====
                       $(0 + n) * m = n * m$  *)
rewrite suma_0_n.    (*  $n * m = n * m$  *)
reflexivity.
```

**Qed.**

```
(* -----
   Ejercicio 2.2.2. Demostrar que si  $m = S n$ , entonces  $m * (1 + n) = m * m$ .
   ----- *)
```

**Theorem** producto\_S\_1 : **forall** n m : **nat**,  
 $m = S n \rightarrow m * (1 + n) = m * m$ .

**Proof.**

```
intros n m H. (* n : nat
                 m : nat
                 H :  $m = S n$ 
                 =====
                  $m * (1 + n) = m * m$  *)
simpl.      (*  $m * S n = m * m$  *)
rewrite H.    (*  $S n * S n = S n * S n$  *)
reflexivity.
```

**Qed.**

```
(* =====
   § 2.3. Demostraciones por análisis de casos
   ===== *)
```

```
(* -----
```

*Ejemplo 2.3.1. Demostrar que  $n + 1$  es distinto de  $0$ .*

```

----- *)

(* 1º intento *)
Theorem siguiente_distinto_cero_primer_intento : forall n : nat,
  iguales_nat (n + 1) 0 = false.
Proof.
  intros n. (* n : nat
    =====
    iguales_nat (n + 1) 0 = false *)
  simpl. (* n : nat
    =====
    iguales_nat (n + 1) 0 = false *)

Abort.

(* 2º intento *)
Theorem siguiente_distinto_cero : forall n : nat,
  iguales_nat (n + 1) 0 = false.
Proof.
  intros n. (* n : nat
    =====
    iguales_nat (n + 1) 0 = false *)
  destruct n as [| n'].
  - (*
    =====
    iguales_nat (0 + 1) 0 = false *)
    reflexivity.
  - (* n' : nat
    =====
    iguales_nat (S n' + 1) 0 = false *)
    reflexivity.
Qed.

(* -----
  Ejemplo 2.3.2. Demostrar que la negacion es involutiva; es decir, la
  negacion de la negacion de b es b.
  ----- *)

Theorem negacion_involutiva : forall b : bool,
  negacion (negacion b) = b.
```

**Proof.**

```

intros b.      (*
                    =====
                    negacion (negacion b) = b *)

destruct b.
-              (*
                    =====
                    negacion (negacion true) = true *)

    reflexivity.
-              (*
                    =====
                    negacion (negacion false) = false *)

    reflexivity.

```

**Qed.**

```

(* -----
   Ejemplo 2.3.3. Demostrar que la conjuncion es conmutativa.
   ----- *)

```

(\* 1ª demostración \*)

**Theorem** conjuncion\_commutativa : **forall** b c,  
 conjuncion b c = conjuncion c b.

**Proof.**

```

intros b c.    (* b : bool
                  c : bool
                  =====
                  b && c = c && b *)

destruct b.
-              (* c : bool
                  =====
                  true && c = c && true *)

    destruct c.
    +          (* =====
                  true && true = true && true *)

        reflexivity.
    +          (*
                  =====
                  true && false = false && true *)

        reflexivity.
-              (* c : bool

```

```

=====
false && c = c && false *)
destruct c.
+
  (*
=====
false && true = true && false *)
  reflexivity.
+
  (*
=====
false && false = false && false *)
  reflexivity.
Qed.

```

(\* 2ª demostración \*)

**Theorem** conjuncion\_commutativa2 : forall b c,  
conjuncion b c = conjuncion c b.

**Proof.**

```

intros b c.
destruct b.
{ destruct c.
  { reflexivity. }
  { reflexivity. } }
{ destruct c.
  { reflexivity. }
  { reflexivity. } }

```

**Qed.**

```

(* -----
Ejemplo 2.3.4. Demostrar que
conjuncion (conjuncion b c) d = conjuncion (conjuncion b d) c.
----- *)

```

**Theorem** conjuncion\_intercambio : forall b c d,  
conjuncion (conjuncion b c) d = conjuncion (conjuncion b d) c.

**Proof.**

```

intros b c d.
destruct b.
- destruct c.
  { destruct d.
    - reflexivity. (* (true && true) && true = (true && true) && true *)

```

```

- reflexivity. } (* (true && true) && false = (true && false) && true *)
{ destruct d.
- reflexivity. (* (true && false) && true = (true && true) && false *)
- reflexivity. } (* (true && false) && false = (true && false) && false *)
- destruct c.
{ destruct d.
- reflexivity. (* (false && true) && true = (false && true) && true *)
- reflexivity. } (* (false && true) && false = (false && false) && true *)
{ destruct d.
- reflexivity. (* (false && false) && true = (false && true) && false *)
- reflexivity. } (* (false && false) && false = (false && false) && false *)
Qed.

```

```

(* -----
   Ejemplo 2.3.5. Demostrar que  $n + 1$  es distinto de 0.
   ----- *)

```

**Theorem** siguiente\_distinto\_cero' : forall n : nat,  
iguales\_nat (n + 1) 0 = false.

**Proof.**

```

intros [|n].
- reflexivity. (* iguales_nat (0 + 1) 0 = false *)
- reflexivity. (* iguales_nat (S n + 1) 0 = false *)
Qed.

```

```

(* -----
   Ejemplo 2.3.6. Demostrar que la conjuncion es conmutativa.
   ----- *)

```

**Theorem** conjuncion\_commutativa'' : forall b c,  
conjuncion b c = conjuncion c b.

**Proof.**

```

intros [] [].
- reflexivity. (* true && true = true && true *)
- reflexivity. (* true && false = false && true *)
- reflexivity. (* false && true = true && false *)
- reflexivity. (* false && false = false && false *)
Qed.

```

```

(* -----

```

*Ejercicio 2.2.3. Demostrar que si  
conjuncion b c = true, entonces c = true.*

----- \*)

**Theorem** conjuncion\_true\_elim : **forall** b c : **bool**,  
conjuncion b c = **true** -> c = **true**.

**Proof.**

```

intros b c.      (* b : bool
                  c : bool
                  =====
                  b && c = true -> c = true *)

destruct c.
-
  (* b : bool
  =====
  b && true = true -> true = true *)

  reflexivity.
-
  (* b : bool
  =====
  b && false = true -> false = true *)

  destruct b.
+
  (*
  =====
  true && false = true -> false = true *)

  simpl.
  (*
  =====
  false = true -> false = true *)

  intros H.
  (* H : false = true
  =====
  false = true *)

  rewrite H.
  (* H : false = true
  =====
  true = true *)

  reflexivity.
+
  (*
  =====
  false && false = true -> false = true *)

  simpl.
  (*
  =====
  false = true -> false = true *)

  intros H.
  (* H : false = true

```

```

=====
      false = true *)
rewrite H.   (* H : false = true
=====
      true = true *)

reflexivity.

Qed.

(* -----
   Ejercicio 2.2.4. Demostrar que 0 es distinto de n + 1.
   ----- *)

Theorem cero_distinto_mas_uno: forall n : nat,
  iguales_nat 0 (n + 1) = false.
Proof.
  intros [| n'].
  - reflexivity. (* iguales_nat 0 (0 + 1) = false *)
  - reflexivity. (* iguales_nat 0 (S n' + 1) = false *)
Qed.

(* =====
   § 3. Ejercicios complementarios
   ===== *)

(* -----
   Ejercicio 3.1. Demostrar que
   forall (f : bool -> bool),
     (forall (x : bool), f x = x) -> forall (b : bool), f (f b) = b.
   ----- *)

Theorem aplica_dos_veces_la_identidad : forall (f : bool -> bool),
  (forall (x : bool), f x = x) -> forall (b : bool), f (f b) = b.
Proof.
  intros f H b. (* f : bool -> bool
                  H : forall x : bool, f x = x
                  b : bool
                  =====
                  f (f b) = b *)
  rewrite H.   (* f b = b *)
  rewrite H.   (* b = b *)

```

**reflexivity.**  
**Qed.**

```
(* -----
Ejercicio 3.2. Demostrar que
  forall (b c : bool),
    (conjuncion b c = disyuncion b c) -> b = c.
----- *)
```

**Theorem** conjuncion\_igual\_disyuncion: **forall** (b c : **bool**),  
 (conjuncion b c = disyuncion b c) -> b = c.

**Proof.**

```
intros [] c.
-
  (* c : bool
  =====
    true && c = true || c -> true = c *)
simpl.
  (* c : bool
  =====
    c = true -> true = c *)
intros H.
  (* c : bool
    H : c = true
    =====
    true = c *)
rewrite H.
  (* c : bool
    H : c = true
    =====
    true = true *)
reflexivity.
-
  (* c : bool
  =====
    false && c = false || c -> false = c *)
simpl.
  (* c : bool
  =====
    false = c -> false = c *)
intros H.
  (* c : bool
    H : false = c
    =====
    false = c *)
rewrite H.
  (* c : bool
    H : false = c
```



```

=====
c = c *)

reflexivity.
Qed.

(* -----
Ejercicio 3.3. En este ejercicio se considera la siguiente
representación de los números naturales
  Inductive nat2 : Type :=
    | C : nat2
    | D : nat2 -> nat2
    | SD : nat2 -> nat2.
donde C representa el cero, D el doble y SD el siguiente del doble.

  Definir la función
    nat2Anat : nat2 -> nat
  tal que (nat2Anat x) es el número natural representado por x.

  Demostrar que
    nat2Anat (SD (SD C))      = 3
    nat2Anat (D (SD (SD C))) = 6.
----- *)

Inductive nat2 : Type :=
  | C : nat2
  | D : nat2 -> nat2
  | SD : nat2 -> nat2.

Fixpoint nat2Anat (x:nat2) : nat :=
  match x with
  | C    => 0
  | D n  => 2 * nat2Anat n
  | SD n => (2 * nat2Anat n) + 1
  end.

Example prop_nat2Anat1: (nat2Anat (SD (SD C))) = 3.
Proof. reflexivity. Qed.

Example prop_nat2Anat2: (nat2Anat (D (SD (SD C)))) = 6.
Proof. reflexivity. Qed.

```

```
(* =====  
  § Bibliografía  
  ===== *)  
  
(*  
  + "Functional programming in Coq" de Peirce et als. http://bit.ly/2zRCL6t  
  *)
```

## Tema 2

# Demostraciones por inducción sobre los números naturales en Coq

```
(* T2: Demostraciones por inducción sobre los números naturales en Coq *)
```

```
Require Export T1_PF_en_Coq.
```

```
(* El contenido de la teoría es  
  1. Demostraciones por inducción.  
  2. Demostraciones anidadas.  
  3. Demostraciones formales vs demostraciones informales.  
  4. Ejercicios complementarios *)
```

```
(* =====  
  § 1. Demostraciones por inducción  
  ===== *)
```

```
(* -----  
  Ejemplo 1.1. Demostrar que  
    forall n:nat, n = n + 0.  
  ----- *)
```

```
(* 1º intento: con métodos elementales *)
```

```
Theorem suma_n_0_a: forall n:nat, n = n + 0.
```

```
Proof.
```

```
  intros n. (* n : nat
```

```
            =====
```

```

      n = n + 0 *)
simpl.    (* n : nat
            =====
            n = n + 0 *)

```

Abort.

(\* 2º intento: con casos \*)

**Theorem** suma\_n\_0\_b : **forall** n:**nat**,  
 n = n + 0.

**Proof.**

```

intros n.          (* n : nat
                    =====
                    n = n + 0 *)

destruct n as [| n'].
-
  (*
  =====
  0 = 0 + 0 *)

  reflexivity.
-
  (* n' : nat
  =====
  S n' = S n' + 0 *)

  simpl.          (* n' : nat
                    =====
                    S n' = S (n' + 0) *)

```

Abort.

(\* 3ª intento: con inducción \*)

**Theorem** suma\_n\_0 : **forall** n:**nat**,  
 n = n + 0.

**Proof.**

```

intros n.          (* n : nat
                    =====
                    n = n + 0 *)

induction n as [| n' IHn'].
+
  (*
  =====
  0 = 0 + 0 *)

  reflexivity.
+
  (* n' : nat
     IHn' : n' = n' + 0

```

```

                                =====
                                S n' = S n' + 0 *)
simpl.                        (* S n' = S (n' + 0) *)
rewrite <- IHn'.              (* S n' = S n' *)
reflexivity.
Qed.

(* -----
   Ejemplo 1.2. Demostrar que
   forall n, n - n = 0.
   ----- *)

Theorem resta_n_n: forall n, n - n = 0.
Proof.
  intros n.                    (* n : nat
                                =====
                                n - n = 0 *)

  induction n as [| n' IHn'].
  +
    (*
      =====
      0 - 0 = 0 *)

    reflexivity.
  +
    (* n' : nat
       IHn' : n' - n' = 0
       =====
       S n' - S n' = 0 *)
    simpl.                    (* n' - n' = 0 *)
    rewrite -> IHn'.           (* 0 = 0 *)
    reflexivity.
Qed.

(* -----
   Ejercicio 1.1. Demostrar que
   forall n:nat, n * 0 = 0.
   ----- *)

```

**Theorem** multiplica\_n\_0: **forall** n:nat, n \* 0 = 0.

**Proof.**

```

intros n.                    (* n : nat
                                =====

```

```

                                n * 0 = 0 *)
induction n as [| n' IHn'].
+
                                (*
                                =====
                                0 * 0 = 0 *)

                                reflexivity.

+
                                (* n' : nat
                                IHn' : n' * 0 = 0
                                =====
                                S n' * 0 = 0 *)
                                simpl.
                                rewrite IHn'.
                                reflexivity.
Qed.

(* -----
   Ejercicio 1.2. Demostrar que,
   forall n m : nat, S (n + m) = n + (S m).
   ----- *)

Theorem suma_n_Sm: forall n m : nat, S (n + m) = n + (S m).
Proof.
  intros n m.
                                (* n, m : nat
                                =====
                                S (n + m) = n + S m *)

  induction n as [| n' IHn'].
+
                                (* m : nat
                                =====
                                S (0 + m) = 0 + S m *)

                                simpl.
                                (* m : nat
                                =====
                                S m = S m *)

                                reflexivity.

+
                                (* S (S n' + m) = S n' + S m *)
                                simpl.
                                (* S (S (n' + m)) = S (n' + S m) *)
                                rewrite IHn'.
                                (* S (n' + S m) = S (n' + S m) *)
                                reflexivity.
Qed.

(* -----
```

*Ejercicio 1.3. Demostrar que*  
*forall n m : nat, n + m = m + n.*

----- \*)

**Theorem** suma\_conmutativa: forall n m : nat,  
 n + m = m + n.

**Proof.**

```

intros n m.                                (* n, m : nat
                                             =====
                                             n + m = m + n *)

induction n as [|n' IHn'].
+
  (* m : nat
   =====
   0 + m = m + 0 *)

  simpl.
  rewrite <- suma_n_0.
  reflexivity.
+
  (* n', m : nat
   IHn' : n' + m = m + n'
   =====
   S n' + m = m + S n' *)

  simpl.
  rewrite IHn'.
  rewrite <- suma_n_Sm.
  reflexivity.

```

**Qed.**

(\* -----  
*Ejercicio 1.4. Demostrar que*  
*forall n m p : nat, n + (m + p) = (n + m) + p.*  
 ----- \*)

**Theorem** suma\_asociativa: forall n m p : nat, n + (m + p) = (n + m) + p.

**Proof.**

```

intros n m p.                                (* n, m, p : nat
                                             =====
                                             n + (m + p) = (n + m) + p *)

induction n as [|n' IHn'].
+
  (* m, p : nat
   =====

```

```

                                 $\theta + (m + p) = (\theta + m) + p$  *)
reflexivity.
+
                                (* n', m, p : nat
                                IHn' : n' + (m + p) = n' + m + p
                                =====
                                S n' + (m + p) = (S n' + m) + p *)
simpl.
rewrite IHn'.
reflexivity.
Qed.

(* -----
   Ejercicio 1.5. Se considera la siguiente función que dobla su argumento.
   Fixpoint doble (n:nat) :=
     match n with
     | 0      => 0
     | S n'  => S (S (doble n'))
     end.

   Demostrar que
     forall n, doble n = n + n.
   ----- *)

Fixpoint doble (n:nat) :=
match n with
| 0      => 0
| S n'  => S (S (doble n'))
end.

Lemma doble_suma : forall n, doble n = n + n .
Proof.
intros n.
                                (* n : nat
                                =====
                                doble n = n + n *)

induction n as [|n' IHn'].
+
                                (*
                                =====
                                doble 0 = 0 + 0 *)

reflexivity.
+
                                (* n' : nat

```



```

                                IHn' : doble n' = n' + n'
                                =====
                                doble (S n') = S n' + S n' *)
simpl.                        (* S (S (doble n')) = S (n' + S n') *)
rewrite IHn'.                  (* S (S (n' + n')) = S (n' + S n') *)
rewrite suma_n_Sm.            (* S (n' + S n') = S (n' + S n') *)
reflexivity.
Qed.

(* -----
   Ejercicio 1.6. Demostrar que
   forall n : nat, esPar (S n) = negacion (esPar n).
   ----- *)

Theorem esPar_S : forall n : nat,
  esPar (S n) = negacion (esPar n).
Proof.
  intros n.                      (* n : nat
                                  =====
                                  esPar (S n) = negacion (esPar n) *)

  induction n as [|n' IHn'].
  +
    (*
      =====
      esPar 1 = negacion (esPar 0) *)
    simpl.                      (*
      =====
      false = false *)

    reflexivity.
  +
    (* n' : nat
       IHn' : esPar (S n') = negacion (esPar n')
       =====
       esPar (S (S n')) =
         negacion (esPar (S n')) *)
    rewrite IHn'.                (* esPar (S (S n')) =
                                   negacion (negacion (esPar n')) *)
    rewrite negacion_involutiva. (* esPar (S (S n')) = esPar n' *)
    simpl.                      (* esPar n' = esPar n' *)
    reflexivity.
Qed.

```

```
(* =====
§ 2. Demostraciones anidadas
===== *)
```

```
(* -----
Ejemplo 2.1. Demostrar que
  forall n m : nat, (0 + n) * m = n * m.
----- *)
```

**Theorem** producto\_0\_suma': forall n m : nat, (0 + n) \* m = n \* m.

**Proof.**

```
  intros n m.          (* n, m : nat
                        =====
                        (0 + n) * m = n * m *)

  assert (H: 0 + n = n).
-
    (* n, m : nat
    =====
    0 + n = n *)

    reflexivity.

-
    (* n, m : nat
    H : 0 + n = n
    =====
    (0 + n) * m = n * m *)

    rewrite -> H.      (* n * m = n * m *)
    reflexivity.
```

**Qed.**

```
(* -----
Ejemplo 2.2. Demostrar que
  forall n m p q : nat, (n + m) + (p + q) = (m + n) + (p + q)
----- *)
```

(\* 1º intento sin assert\*)

**Theorem** suma\_reordenada\_1: forall n m p q : nat,  
(n + m) + (p + q) = (m + n) + (p + q).

**Proof.**

```
  intros n m p q.      (* n, m, p, q : nat
                        =====
                        (n + m) + (p + q) = (m + n) + (p + q) *)

  rewrite -> suma_conmutativa. (* n, m, p, q : nat
```

$$p + q + (n + m) = m + n + (p + q) \quad *)$$

Abort.

(\* 2º intento con assert \*)

**Theorem** suma\_reordenada: **forall** n m p q : **nat**,  
 $(n + m) + (p + q) = (m + n) + (p + q)$ .

**Proof.**

```

intros n m p q.
    (* n, m, p, q : nat
    =====
    (n + m) + (p + q) = (m + n) + (p + q) *)

assert (H: n + m = m + n).
-
    (* n, m, p, q : nat
    =====
    n + m = m + n *)
    rewrite -> suma_conmutativa. (* m + n = m + n *)
    reflexivity.
-
    (* n, m, p, q : nat
    H : n + m = m + n
    =====
    (n + m) + (p + q) = (m + n) + (p + q) *)
    (* m + n + (p + q) = m + n + (p + q) *)

    rewrite -> H.
    reflexivity.

```

**Qed.**

```

(* =====
   § 3. Demostraciones formales vs demostraciones informales
   ===== *)

(* -----
   Ejercicio 3.1. Escribir la demostración informal (en lenguaje natural)
   correspondiente a la demostración formal de la asociatividad de la
   suma del ejercicio 1.4.
   ----- *)

(* Demostración por inducción en n.

```

- Caso base: Se supone que  $n$  es  $0$  y hay que demostrar que  
 $0 + (m + p) = (0 + m) + p$ .  
 Esto es consecuencia inmediata de la definición de suma.

- Paso de inducción: Suponemos la hipótesis de inducción

$$n' + (m + p) = (n' + m) + p.$$

Hay que demostrar que

$$(S\ n') + (m + p) = ((S\ n') + m) + p.$$

que, por la definición de suma, se reduce a

$$S\ (n' + (m + p)) = S\ ((n' + m) + p)$$

que por la hipótesis de inducción se reduce a

$$S\ ((n' + m) + p) = S\ ((n' + m) + p)$$

que es una identidad. \*)

(\* -----  
Ejercicio 3.2. Escribir la demostración informal (en lenguaje natural)  
correspondiente a la demostración formal de la asociatividad de la  
suma del ejercicio 1.3.  
----- \*)

(\* Demostración por inducción en  $n$ .

- Caso base: Se supone que  $n$  es  $0$  y hay que demostrar que

$$0 + m = m + 0$$

que, por la definición de la suma, se reduce a

$$m = m + 0$$

que se verifica por el lema `suma_n_0`.

- Paso de inducción: Suponemos la hipótesis de inducción

$$n' + m = m + n'$$

Hay que demostrar que

$$S\ n' + m = m + S\ n'$$

que, por la definición de suma, se reduce a

$$S\ (n' + m) = m + S\ n'$$

que, por la hipótesis de inducción, se reduce a

$$S\ (m + n') = m + S\ n'$$

que, por el lema `suma_n_Sm`, se reduce a

$$S\ (m + n') = S\ (m + n')$$

que es una identidad. \*)

(\* -----  
Ejercicio 3.3. Demostrar que  
forall  $n:\text{nat}$ , `iguales_nat n n = true`.

```

----- *)

Theorem iguales_nat_refl: forall n : nat,
  iguales_nat n n = true.
Proof.
  intros n.                                (* n : nat
                                           =====
                                           iguales_nat n n = true *)

  induction n as [|n' IHn'].
  -                                         (*
                                           =====
                                           iguales_nat 0 0 = true *)

    reflexivity.
  -                                         (* n' : nat
                                           IHn' : iguales_nat n' n' = true
                                           =====
                                           iguales_nat (S n') (S n') = true *)

    simpl.                                (* iguales_nat n' n' = true *)
    rewrite <- IHn'.                        (* true = true *)
    reflexivity.

Qed.

(* -----
   Ejercicio 3.4. Escribir la demostración informal (en lenguaje natural)
   correspondiente la demostración del ejercicio anterior.
   ----- *)

(* Demostración por inducción en n.

- Caso base: Se supone que n es 0 y hay que demostrar que
  true = iguales_nat 0 0
  que se verifica por la definición de iguales_nat.

- Paso de inducción: Suponemos la hipótesis de inducción
  true = iguales_nat n' n'
  Hay que demostrar que
  true = iguales_nat (S n') (S n')
  que, por la definición de iguales_nat, se reduce a
  true = iguales_nat n' n
  que, por la hipótesis de inducción, se reduce a

```

```

    true = true
    que es una identidad. *)

(* =====
   § 4. Ejercicios complementarios
   ===== *)

(* -----
   Ejercicio 4.1. Demostrar, usando assert pero no induct,
   forall n m p : nat, n + (m + p) = m + (n + p).
   ----- *)

Theorem suma_permutada: forall n m p : nat,
  n + (m + p) = m + (n + p).
Proof.
  intros n m p. (* n, m, p : nat
                  =====
                  n + (m + p) = m + (n + p) *)
  rewrite suma_asociativa. (* n, m, p : nat
                             =====
                             (n + m) + p = m + (n + p) *)
  rewrite suma_asociativa. (* n, m, p : nat
                             =====
                             n + m + p = m + n + p *)
  assert (H : n + m = m + n).
  - (* n, m, p : nat
      =====
      n + m = m + n *)
    rewrite suma_conmutativa. (* m + n = m + n *)
    reflexivity.
  - (* n, m, p : nat
      H : n + m = m + n
      =====
      (n + m) + p = (m + n) + p *)
    rewrite H. (* (m + n) + p = (m + n) + p *)
    reflexivity.
Qed.

(* -----
   Ejercicio 4.2. Demostrar que la multiplicación es conmutativa.
   ----- *)

```

```

----- *)

Lemma producto_n_1 : forall n: nat,
  n * 1 = n.
Proof.
  intro n.
  (* n : nat
     =====
     n * 1 = n *)

  induction n as [|n' IHn'].
  -
  (*
     =====
     0 * 1 = 0 *)

    reflexivity.
  -
  (* n' : nat
     IHn' : n' * 1 = n'
     =====
     S n' * 1 = S n' *)
    simpl.
    rewrite IHn'.
    reflexivity.
Qed.

Theorem suma_n_1 : forall n : nat,
  n + 1 = S n.
Proof.
  intro n.
  (* n : nat
     =====
     n + 1 = S n *)

  induction n as [|n' HIn'].
  -
  (*
     =====
     0 + 1 = 1 *)

    reflexivity.
  -
  (* n' : nat
     HIn' : n' + 1 = S n'
     =====
     S n' + 1 = S (S n') *)
    simpl.
    rewrite HIn'.
    reflexivity.

```

**Qed.**

**Theorem** producto\_n\_Sm: **forall** n m : **nat**,

$$n * (m + 1) = n * m + n.$$

**Proof.**

```

intros n m.                                (* n, m : nat
                                             =====
                                             n * (m + 1) = n * m + n *)

induction n as [|n' IHn'].
-
  reflexivity.
-
  (* n', m : nat
     IHn' : n' * (m + 1) = n' * m + n'
     =====
     S n' * (m + 1) = S n' * m + S n' *)
  simpl.
  (* (m + 1) + n' * (m + 1) =
     (m + n' * m) + S n' *)
  rewrite IHn'.
  (* (m + 1) + (n' * m + n') =
     (m + n' * m) + S n' *)
  rewrite suma_permutada.
  (* n' * m + ((m + 1) + n') =
     (m + n' * m) + S n' *)
  rewrite <- suma_asociativa.
  (* n' * m + (m + (1 + n')) =
     (m + n' * m) + S n' *)
  rewrite <- suma_n_1.
  (* n' * m + (m + (n' + 1)) =
     (m + n' * m) + S n' *)
  rewrite suma_n_1.
  (* n' * m + (m + S n') = (m + n' * m) + S n' *)
  rewrite suma_permutada.
  (* m + (n' * m + S n') = (m + n' * m) + S n' *)
  rewrite suma_asociativa.
  (* m + (n' * m + S n') = (m + n' * m) + S n' *)
  reflexivity.

```

**Qed.**

**Theorem** producto\_conmutativa: **forall** m n : **nat**,

$$m * n = n * m.$$

**Proof.**

```

intros n m.                                (* n, m : nat
                                             =====
                                             n * m = m * n *)

induction n as [|n' HIn'].

```



```

-
      (* m : nat
      =====
      0 * m = m * 0 *)
rewrite multiplica_n_0. (* 0 * m = 0 *)
reflexivity.
-
      (* n', m : nat
      HIn' : n' * m = m * n'
      =====
      S n' * m = m * S n' *)
simpl. (* m + n' * m = m * S n' *)
rewrite HIn'. (* m + m * n' = m * S n' *)
rewrite <- suma_n_1. (* m + m * n' = m * (n' + 1) *)
rewrite producto_n_Sm. (* m + m * n' = m * n' + m *)
rewrite suma_conmutativa. (* m * n' + m = m * n' + m *)
reflexivity.
Qed.

(* -----
   Ejercicio 4.3. Demostrar que
   forall n : nat, true = menor_o_igual n n.
   ----- *)

Theorem menor_o_igual_refl: forall n : nat,
  true = menor_o_igual n n.
Proof.
  intro n.
      (* n : nat
      =====
      true = menor_o_igual n n *)
  induction n as [| n' HIn'].
  -
      (*
      =====
      true = menor_o_igual 0 0 *)
      reflexivity.
  -
      (* n' : nat
      HIn' : true = menor_o_igual n' n'
      =====
      true = menor_o_igual (S n') (S n') *)
      simpl. (* true = menor_o_igual n' n' *)
      rewrite HIn'. (* menor_o_igual n' n' = menor_o_igual n' n' *)

```

```

    reflexivity.
Qed.

```

```

(* -----
   Ejercicio 4.4. Demostrar que
       forall n : nat, iguales_nat 0 (S n) = false.
   ----- *)

```

```

Theorem cero_distinto_S: forall n : nat,
    iguales_nat 0 (S n) = false.

```

```

Proof.

```

```

    intros n.      (* n : nat
                     =====
                     iguales_nat 0 (S n) = false *)
    simpl.         (* false = false *)
    reflexivity.
Qed.

```

```

(* -----
   Ejercicio 4.5. Demostrar que
       forall b : bool, conjuncion b false = false.
   ----- *)

```

```

Theorem conjuncion_false_r : forall b : bool,
    conjuncion b false = false.

```

```

Proof.

```

```

    intros b.      (* b : bool
                     =====
                     b && false = false *)
    destruct b.
    -              (*
                     =====
                     true && false = false *)
      simpl.      (* false = false *)
      reflexivity.
    -              (*
                     =====
                     false && false = false *)
      simpl.      (* false = false *)
      reflexivity.

```

**Qed.**

```
(* -----
  Ejercicio 4.6. Demostrar que
    forall n m p : nat, menor_o_igual n m = true ->
      menor_o_igual (p + n) (p + m) = true.
  ----- *)
```

**Theorem** menor\_o\_igual\_suma: **forall** n m p : **nat**,  
 menor\_o\_igual n m = **true** -> menor\_o\_igual (p + n) (p + m) = **true**.

**Proof.**

```
  intros n m p H. (* n, m, p : nat
                   H : menor_o_igual n m = true
                   =====
                   menor_o_igual (p + n) (p + m) = true *)

  induction p as [|p' HIp'].
  - (* n, m : nat
      H : menor_o_igual n m = true
      =====
      menor_o_igual (0 + n) (0 + m) = true *)

    simpl.
    rewrite H.
    reflexivity.

  - (* n, m, p' : nat
      H : menor_o_igual n m = true
      HIp' : menor_o_igual (p' + n) (p' + m) = true
      =====
      menor_o_igual (S p' + n) (S p' + m) = true *)

    simpl.
    rewrite HIp'.
    reflexivity.
```

**Qed.**

```
(* -----
  Ejercicio 4.7. Demostrar que
    forall n : nat, iguales_nat (S n) 0 = false.
  ----- *)
```

**Theorem** S\_distinto\_0 : **forall** n:**nat**,  
 iguales\_nat (S n) 0 = **false**.

**Proof.**

```

intro n.      (* n : nat
                =====
                iguales_nat (S n) 0 = false *)
simpl.      (* false = false *)
reflexivity.
Qed.

```

```

(* -----
   Ejercicio 4.8. Demostrar que
   forall n:nat, 1 * n = n.
   ----- *)

```

**Theorem** producto\_1\_n: **forall** n:**nat**, 1 \* n = n.

**Proof.**

```

intro n.      (* n : nat
                =====
                1 * n = n *)
simpl.      (* n + 0 = n *)
rewrite suma_n_0. (* n + 0 = n + 0 *)
reflexivity.
Qed.

```

```

(* -----
   Ejercicio 4.9. Demostrar que
   forall b c : bool, disyuncion (conjuncion b c)
                       (disyuncion (negacion b)
                                   (negacion c))
                       = true.
   ----- *)

```

**Theorem** alternativas: **forall** b c : **bool**,

```

disyuncion
  (conjuncion b c)
  (disyuncion (negacion b)
              (negacion c))
= true.

```

**Proof.**

```

intros [] [].
- reflexivity. (* (true && true) || (negacion true || negacion true) = true *)

```

```

- reflexivity. (* (true && false) || (negacion true || negacion false) = true *)
- reflexivity. (* (false && true) || (negacion false || negacion true) = true *)
- reflexivity. (* (false && false) || (negacion false || negacion false)=true *)

```

**Qed.**

```

(* -----
   Ejercicio 4.10. Demostrar que
     forall n m p : nat, (n + m) * p = (n * p) + (m * p).
   ----- *)

```

**Theorem** producto\_suma\_distributiva\_d: **forall** n m p : **nat**,  
 (n + m) \* p = (n \* p) + (m \* p).

**Proof.**

```

intros n m p. (* n, m, p : nat
                =====
                (n + m) * p = n * p + m * p *)

induction n as [|n' HIn'].
- (* m, p : nat
  =====
  (0 + m) * p = 0 * p + m * p *)

  reflexivity.
- (* n', m, p : nat
   HIn' : (n' + m) * p = n' * p + m * p
   =====
   (S n' + m) * p = S n' * p + m * p *)

  simpl. (* p + (n' + m) * p = (p + n' * p) + m * p *)
  rewrite HIn'. (* p + (n' * p + m * p) = (p + n') * p + m * p *)
  rewrite suma_asociativa. (* (p + n' * p) + m * p = (p + n' * p) + m * p *)
  reflexivity.

```

**Qed.**

```

(* -----
   Ejercicio 4.11. Demostrar que
     forall n m p : nat, n * (m * p) = (n * m) * p.
   ----- *)

```

**Theorem** producto\_asociativa: **forall** n m p : **nat**,  
 n \* (m \* p) = (n \* m) \* p.

**Proof.**

```

intros n m p. (* n, m, p : nat

```

```

=====
n * (m * p) = (n * m) * p *)
induction n as [|n' HIn'].
-
  (* m, p : nat
  =====
  0 * (m * p) = (0 * m) * p *)
  simpl.
  reflexivity.
-
  (* n', m, p : nat
  HIn' : n' * (m * p) = (n' * m) * p
  =====
  S n' * (m * p) = (S n' * m) * p *)
  simpl.
  (* m * p + n' * (m * p) = (m + n' * m) * p *)
  rewrite HIn'.
  (* m * p + (n' * m) * p = (m + n' * m) * p *)
  rewrite producto_suma_distributiva_d.
  (* m * p + (n' * m) * p = m * p + (n' * m) * p *)
  reflexivity.
Qed.

```

```

(* -----
Ejercicio 11. La táctica replace permite especificar el subtérmino
que se desea reescribir y su sustituto:
  replace t with u
sustituye todas las copias de la expresión t en el objetivo por la
expresión u y añade la ecuación (t = u) como un nuevo subobjetivo.

El uso de la táctica replace es especialmente útil cuando la táctica
rewrite actúa sobre una parte del objetivo que no es la que se desea.

Demostrar, usando la táctica replace y sin usar
[assert (n + m = m + n)], que
  forall n m p : nat, n + (m + p) = m + (n + p).
----- *)

```

**Theorem** suma\_permutada' : **forall** n m p : **nat**,  
 n + (m + p) = m + (n + p).

**Proof.**

```

intros n m p.
(* n, m, p : nat
=====
n + (m + p) = m + (n + p) *)

```

```

rewrite suma_asociativa.      (* (n + m) + p = m + (n + p) *)
rewrite suma_asociativa.      (* (n + m) + p = (m + n) + p *)
replace (n + m) with (m + n).
-                               (* n, m, p : nat
                               =====
                               (m + n) + p = (m + n) + p *)

  reflexivity.
-                               (* n, m, p : nat
                               =====
                               m + n = n + m *)

  rewrite suma_conmutativa.    (* n + m = n + m *)
  reflexivity.
Qed.

(* =====
   § Bibliografía
   ===== *)

(*
+ "Demostraciones por inducción" de Peirce et als. http://bit.ly/2NRSWTF
*)

```





# Tema 3

## Datos estructurados en Coq

*(\* T3: Datos estructurados en Coq \*)*

**Require Export** T2\_Induccion.

*(\* El contenido de la teoría es*

- 1. Pares de números*
- 2. Listas de números*
  - 1. El tipo de la lista de números.*
  - 2. La función repite (repeat)*
  - 3. La función longitud (length)*
  - 4. La función conc (app)*
  - 5. Las funciones primero (hd) y resto (tl)*
  - 6. Ejercicios sobre listas de números*
  - 7. Multiconjuntos como listas*
- 3. Razonamiento sobre listas*
  - 1. Demostraciones por simplificación*
  - 2. Demostraciones por casos*
  - 3. Demostraciones por inducción*
  - 4. Ejercicios*
- 4. Opcionales*
- 5. Diccionarios (o funciones parciales)*
- 6. Bibliografía*

*\*)*

*(\* =====*  
*§ 1. Pares de números*  
*===== \*)*

```
(* -----
   Nota. Se inicia el módulo ListaNat.
   ----- *)
```

**Module** ListaNat.

```
(* -----
   Ejemplo 1.1. Definir el tipo ProdNat para los pares de números
   naturales con el constructor
       par : nat -> nat -> ProdNat.
   ----- *)
```

**Inductive** ProdNat : **Type** :=  
 par : **nat** -> **nat** -> ProdNat.

```
(* -----
   Ejemplo 1.2. Calcular el tipo de la expresión (par 3 5)
   ----- *)
```

**Check** (par 3 5).

```
(* ==> par 3 5 : ProdNat *)
```

```
(* -----
   Ejemplo 1.3. Definir la función
       fst : ProdNat -> nat
   tal que (fst p) es la primera componente de p.
   ----- *)
```

**Definition** fst (p : ProdNat) : **nat** :=  
**match** p **with**  
 | par x y => x  
**end**.

```
(* -----
   Ejemplo 1.4. Evaluar la expresión
       fst (par 3 5)
   ----- *)
```

**Compute** (fst (par 3 5)).

```
(* ==> 3 : nat *)
```

```
(* -----
  Ejemplo 1.5. Definir la función
    snd : ProdNat -> nat
  tal que (snd p) es la segunda componente de p.
  ----- *)
```

```
Definition snd (p : ProdNat) : nat :=
  match p with
  | par x y => y
  end.
```

```
(* -----
  Ejemplo 1.6. Definir la notación (x,y) como una abreviatura de
  (par x y).
  ----- *)
```

```
Notation "( x , y )" := (par x y).
```

```
(* -----
  Ejemplo 1.7. Evaluar la expresión
    fst (3,5)
  ----- *)
```

```
Compute (fst (3,5)).
(* ==> 3 : nat *)
```

```
(* -----
  Ejemplo 1.8. Redefinir la función fst usando la abreviatura de pares.
  ----- *)
```

```
Definition fst' (p : ProdNat) : nat :=
  match p with
  | (x,y) => x
  end.
```

```
(* -----
  Ejemplo 1.9. Redefinir la función snd usando la abreviatura de pares.
  ----- *)
```

**Definition** snd' (p : ProdNat) : nat :=  
 match p with  
 | (x,y) => y  
 end.

(\* -----  
 Ejemplo 1.10. Definir la función  
 intercambia : ProdNat -> ProdNat  
 tal que (intercambia p) es el par obtenido intercambiando las  
 componentes de p.  
 ----- \*)

**Definition** intercambia (p : ProdNat) : ProdNat :=  
 match p with  
 | (x,y) => (y,x)  
 end.

(\* -----  
 Ejemplo 1.11. Demostrar que para todos los naturales  
 (n,m) = (fst (n,m), snd (n,m)).  
 ----- \*)

**Theorem** par\_componentes1 : forall (n m : nat),  
 (n,m) = (fst (n,m), snd (n,m)).

**Proof.**

reflexivity.

**Qed.**

(\* -----  
 Ejemplo 1.12. Demostrar que para todo par de naturales  
 p = (fst p, snd p).  
 ----- \*)

(\* 1º intento \*)

**Theorem** par\_componentes2 : forall (p : ProdNat),  
 p = (fst p, snd p).

**Proof.**

simpl. (\*

=====

forall p : ProdNat, p = (fst p, snd p) \*)

Abort.

*(\* 2º intento \*)*

**Theorem** par\_componentes : **forall** (p : ProdNat),  
p = (fst p, snd p).

**Proof.**

```

intros p.          (* p : ProdNat
                     =====
                     p = (fst p, snd p) *)

destruct p as [n m]. (* n, m : nat
                     =====
                     (n, m) = (fst (n, m), snd (n, m)) *)

simpl.            (* (n, m) = (n, m) *)
reflexivity.

```

**Qed.**

*(\* -----  
Ejercicio 1.1. Demostrar que para todo par de naturales p,  
(snd p, fst p) = intercambia p.  
----- \*)*

**Theorem** ejercicio\_1\_1: **forall** p : ProdNat,  
(snd p, fst p) = intercambia p.

**Proof.**

```

intro p.          (* p : ProdNat
                     =====
                     (snd p, fst p) = intercambia p *)

destruct p as [n m]. (* n, m : nat
                     =====
                     (snd (n, m), fst (n, m)) = intercambia (n, m) *)

simpl.            (* (m, n) = (m, n) *)
reflexivity.

```

**Qed.**

*(\* -----  
Ejercicio 1.2. Demostrar que para todo par de naturales p,  
fst (intercambia p) = snd p.  
----- \*)*

**Theorem** ejercicio\_1\_2: **forall** p : ProdNat,

`fst (intercambia p) = snd p.`

**Proof.**

```

intro p.                (* p : ProdNat
                           =====
                           fst (intercambia p) = snd p *)
destruct p as [n m]. (* n, m : nat
                           =====
                           fst (intercambia (n, m)) = snd (n, m) *)
simpl.                 (* m = m *)
reflexivity.

```

**Qed.**

```

(* =====
   § 2. Listas de números
   ===== *)

```

```

(* =====
   §§ 2.1. El tipo de la lista de números.
   ===== *)

```

```

(* -----
   Ejemplo 2.1.1. Definir el tipo ListaNat de la lista de los números
   naturales y cuyo constructores son
   + nil (la lista vacía) y
   + cons (tal que (cons x ys) es la lista obtenida añadiéndole x a ys.
   ----- *)

```

```

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

```

```

(* -----
   Ejemplo 2.1.2. Definir la constante
   ejLista : ListaNat
   que es la lista cuyos elementos son 1, 2 y 3.
   ----- *)

```

**Definition** ejLista := cons 1 (cons 2 (cons 3 nil)).

```

(* -----

```

*Ejemplo 2.1.3. Definir la notación  $(x :: ys)$  como una abreviatura de  $(cons\ x\ ys)$ .*

----- \*)

**Notation** " $x :: l$ " := (cons x l)  
 (at level 60, **right** associativity).

(\* -----  
*Ejemplo 2.1.4. Definir la notación de las listas finitas escribiendo sus elementos entre corchetes y separados por puntos y comas.*  
 ----- \*)

**Notation** "[ ]" := nil.

**Notation** "[  $x ; .. ; y$  ]" := (cons x .. (cons y nil) ..).

(\* -----  
*Ejemplo 2.1.5. Definir la lista cuyos elementos son 1, 2 y 3 mediante distintas representaciones.*  
 ----- \*)

**Definition** ejLista1 := 1 :: (2 :: (3 :: nil)).

**Definition** ejLista2 := 1 :: 2 :: 3 :: nil.

**Definition** ejLista3 := [1;2;3].

(\* =====  
 §§ 2.2. La función repite (repeat)  
 ===== \*)

(\* -----  
*Ejemplo 2.2.1. Definir la función*  
*repite : nat -> nat -> ListaNat*  
*tal que (repite n k) es la lista formada por k veces el número n. Por*  
*ejemplo,*  
*repite 5 3 = [5; 5; 5]*

*Nota: La función repite es equivalente a la predefinida repeat.*

----- \*)

**Fixpoint** repite (n k : **nat**) : ListaNat :=  
 match k with

```

| 0      => nil
| S k' => n :: (repite n k')
end.

```

Compute (repite 5 3).

```
(* ==> [5; 5; 5] : ListaNat*)
```

```
(* =====
  §§ 2.3. La función longitud (length)
  ===== *)
```

```
(* -----
  Ejemplo 2.3.1. Definir la función
    longitud : ListaNat -> nat
  tal que (longitud xs) es el número de elementos de xs. Por ejemplo,
    longitud [4;2;6] = 3

  Nota: La función longitud es equivalente a la predefinida length
  ----- *)
```

```

Fixpoint longitud (l:ListaNat) : nat :=
match l with
| nil      => 0
| h :: t => S (longitud t)
end.

```

Compute (longitud [4;2;6]).

```
(* ==> 3 : nat *)
```

```
(* =====
  §§ 2.4. La función conc (app)
  ===== *)
```

```
(* -----
  Ejemplo 2.4.1. Definir la función
    conc : ListaNat -> ListaNat -> ListaNat
  tal que (conc xs ys) es la concatenación de xs e ys. Por ejemplo,
    conc [1;3] [4;2;3;5] = [1; 3; 4; 2; 3; 5]

```

Nota: La función conc es equivalente a la predefinida app.



```

----- *)

Fixpoint conc (xs ys : ListaNat) : ListaNat :=
  match xs with
  | nil    => ys
  | x :: zs => x :: (conc zs ys)
  end.

Compute (conc [1;3] [4;2;3;5]).
(* ==> [1; 3; 4; 2; 3; 5] : ListaNat *)

(* -----
   Ejemplo 2.4.2. Definir la notación (xs ++ ys) como una abreviatura de
   (conc xs ys).
   ----- *)

Notation "x ++ y" := (conc x y)
                  (right associativity, at level 60).

(* -----
   Ejemplo 2.4.3. Demostrar que
   [1;2;3] ++ [4;5] = [1;2;3;4;5].
   nil ++ [4;5] = [4;5].
   [1;2;3] ++ nil = [1;2;3].
   ----- *)

Example test_conc1: [1;2;3] ++ [4;5] = [1;2;3;4;5].
Proof. reflexivity. Qed.

Example test_conc2: nil ++ [4;5] = [4;5].
Proof. reflexivity. Qed.

Example test_conc3: [1;2;3] ++ nil = [1;2;3].
Proof. reflexivity. Qed.

(* =====
   §§ 2.5. Las funciones primero (hd) y resto (tl)
   ===== *)

(* -----
```

*Ejemplo 2.5.1. Definir la función*

```

    primero : nat -> ListaNat -> ListaNat
    tal que (primero d xs) es el primer elemento de xs o d, si xs es la lista
    vacía. Por ejemplo,
        primero 7 [3;2;5] = 3
        primero 7 []      = 7

```

*Nota. La función primero es equivalente a la predefinida hd*

----- \*)

**Definition** primero (d : nat) (xs : ListaNat) : nat :=  
**match** xs **with**  
 | nil => d  
 | y :: ys => y  
**end.**

Compute (primero 7 [3;2;5]).

(\* ==> 3 : nat \*)

Compute (primero 7 []).

(\* ==> 7 : nat \*)

(\* -----  
*Ejemplo 2.5.2. Demostrar que*  
 primero 0 [1;2;3] = 1.  
 resto [1;2;3] = [2;3].  
 ----- \*)

**Example** prop\_primero1: primero 0 [1;2;3] = 1.

**Proof.** reflexivity. **Qed.**

**Example** prop\_primero2: primero 0 [] = 0.

**Proof.** reflexivity. **Qed.**

(\* -----  
*Ejemplo 2.5.3. Definir la función*  
 resto : ListaNat -> ListaNat  
 tal que (resto xs) es el resto de xs. Por ejemplo.  
 resto [3;2;5] = [2; 5]  
 resto [] = [ ]  
 ----- \*)

*Nota. La función resto es equivalente la predefinida tl.*

----- \*)

**Definition** resto (xs:ListaNat) : ListaNat :=  
**match** xs **with**  
 | nil => nil  
 | y :: ys => ys  
**end.**

Compute (resto [3;2;5]).  
 (\* ==> [2; 5] : ListaNat \*)  
 Compute (resto []).  
 (\* ==> [ ] : ListaNat \*)

(\* -----  
 Ejemplo 2.5.4. Demostrar que  
 resto [1;2;3] = [2;3].  
 ----- \*)

**Example** prop\_resto: resto [1;2;3] = [2;3].  
**Proof.** reflexivity. **Qed.**

(\* =====  
 §§ 2.6. Ejercicios sobre listas de números  
 ===== \*)

(\* -----  
 Ejercicio 2.6.1. Definir la función  
 noCeros : ListaNat -> ListaNat  
 tal que (noCeros xs) es la lista de los elementos de xs distintos de  
 cero. Por ejemplo,  
 noCeros [0;1;0;2;3;0;0] = [1;2;3].  
 ----- \*)

**Fixpoint** noCeros (xs:ListaNat) : ListaNat :=  
**match** xs **with**  
 | nil => nil  
 | a::bs => **match** a **with**  
 | 0 => noCeros bs  
 | \_ => a :: noCeros bs

```

    end
end.

```

```

Compute (noCeros [0;1;0;2;3;0;0]).
(* ==> [1; 2; 3] : ListaNat *)

```

```

(* -----
   Ejercicio 2.6.2. Definir la función
   impares : ListaNat -> ListaNat
   tal que (impares xs) es la lista de los elementos impares de
   xs. Por ejemplo,
   impares [0;1;0;2;3;0;0] = [1;3].
   ----- *)

```

```

Fixpoint impares (xs:ListaNat) : ListaNat :=
  match xs with
  | nil    => nil
  | y::ys => if esImpar y
             then y :: impares ys
             else impares ys
  end.

```

```

Compute (impares [0;1;0;2;3;0;0]).
(* ==> [1; 3] : ListaNat *)

```

```

(* -----
   Ejercicio 2.6.3. Definir la función
   nImpares : ListaNat -> nat
   tal que (nImpares xs) es el número de elementos impares de xs. Por
   ejemplo,
   nImpares [1;0;3;1;4;5] = 4.
   nImpares [0;2;4]      = 0.
   nImpares nil          = 0.
   ----- *)

```

```

Definition nImpares (xs:ListaNat) : nat :=
  longitud (impares xs).

```

```

Example prop_nImpares1: nImpares [1;0;3;1;4;5] = 4.

```

```

Proof. reflexivity. Qed.

```

**Example** prop\_nImpares2: nImpares [0;2;4] = 0.

**Proof.** reflexivity. Qed.

**Example** prop\_nImpares3: nImpares nil = 0.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 2.6.4. Definir la función
  intercaladas : ListaNat -> ListaNat -> ListaNat
tal que (intercaladas xs ys) es la lista obtenida intercalando los
elementos de xs e ys. Por ejemplo,
  intercaladas [1;2;3] [4;5;6] = [1;4;2;5;3;6].
  intercaladas [1] [4;5;6]     = [1;4;5;6].
  intercaladas [1;2;3] [4]     = [1;4;2;3].
  intercaladas [] [20;30]      = [20;30].
----- *)
```

**Fixpoint** intercaladas (xs ys : ListaNat) : ListaNat :=

**match** xs **with**

  | nil     => ys

  | x::xs' => **match** ys **with**

    | nil     => xs

    | y::ys' => x::y::intercaladas xs' ys'

**end**

**end.**

**Example** prop\_intercaladas1: intercaladas [1;2;3] [4;5;6] = [1;4;2;5;3;6].

**Proof.** reflexivity. Qed.

**Example** prop\_intercaladas2: intercaladas [1] [4;5;6] = [1;4;5;6].

**Proof.** reflexivity. Qed.

**Example** prop\_intercaladas3: intercaladas [1;2;3] [4] = [1;4;2;3].

**Proof.** reflexivity. Qed.

**Example** prop\_intercaladas4: intercaladas [] [20;30] = [20;30].

**Proof.** reflexivity. Qed.

```
(* =====
```

## §§ 2.7. Multiconjuntos como listas

===== \*)

(\* -----  
*Ejemplo 2.7.1. Un multiconjunto es una colección de elementos donde no importa el orden de los elementos, pero sí el número de ocurrencias de cada elemento.*

*Definir el tipo multiconjunto de los multiconjuntos de números naturales.*

----- \*)

**Definition** multiconjunto := ListaNat.

(\* -----  
*Ejercicio 2.7.2. Definir la función*  
*n0currencias : nat -> multiconjunto -> nat*  
*tal que (n0currencias x ys) es el número de veces que aparece el elemento x en el multiconjunto ys. Por ejemplo,*  
*n0currencias 1 [1;2;3;1;4;1] = 3.*  
*n0currencias 6 [1;2;3;1;4;1] = 0.*  
 ----- \*)

**Fixpoint** n0currencias (x:nat) (ys:multiconjunto) : nat :=  
 match ys with  
 | nil => 0  
 | y::ys' => if iguales\_nat y x  
           then 1 + n0currencias x ys'  
           else n0currencias x ys'  
 end.

**Example** prop\_n0currencias1: n0currencias 1 [1;2;3;1;4;1] = 3.

**Proof.** reflexivity. **Qed.**

**Example** prop\_n0currencias2: n0currencias 6 [1;2;3;1;4;1] = 0.

**Proof.** reflexivity. **Qed.**

(\* -----  
*Ejercicio 2.7.3. Definir la función*  
*suma : multiconjunto -> multiconjunto -> multiconjunto*

*tal que (suma xs ys) es la suma de los multiconjuntos xs e ys. Por ejemplo,*

*suma [1;2;3] [1;4;1] = [1; 2; 3; 1; 4; 1]  
n0currencias 1 (suma [1;2;3] [1;4;1]) = 3.*

----- \*)

**Definition** suma : multiconjunto -> multiconjunto -> multiconjunto :=  
conc.

**Example** prop\_sum: n0currencias 1 (suma [1;2;3] [1;4;1]) = 3.

**Proof. reflexivity. Qed.**

(\* -----  
*Ejercicio 2.7.4. Definir la función  
agrega : nat -> multiconjunto -> multiconjunto  
tal que (agrega x ys) es el multiconjunto obtenido añadiendo el  
elemento x al multiconjunto ys. Por ejemplo,  
n0currencias 1 (agrega 1 [1;4;1]) = 3.  
n0currencias 5 (agrega 1 [1;4;1]) = 0.*  
----- \*)

**Definition** agrega (x:nat) (ys:multiconjunto) : multiconjunto :=  
x :: ys.

**Example** prop\_agrega1: n0currencias 1 (agrega 1 [1;4;1]) = 3.

**Proof. reflexivity. Qed.**

**Example** prop\_agrega2: n0currencias 5 (agrega 1 [1;4;1]) = 0.

**Proof. reflexivity. Qed.**

(\* -----  
*Ejercicio 2.7.5. Definir la función  
pertenece : nat -> multiconjunto -> bool  
tal que (pertenece x ys) se verifica si x pertenece al multiconjunto  
ys. Por ejemplo,  
pertenece 1 [1;4;1] = true.  
pertenece 2 [1;4;1] = false.*  
----- \*)

**Definition** pertenece (x:nat) (ys:multiconjunto) : bool :=

```
negacion (iguales_nat 0 (n0currencias x ys)).
```

**Example** prop\_pertenece1: pertenece 1 [1;4;1] = true.

**Proof.** reflexivity. Qed.

**Example** prop\_pertenece2: pertenece 2 [1;4;1] = false.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 2.7.6. Definir la función
  eliminaUna : nat -> multiconjunto -> multiconjunto
tal que (eliminaUna x ys) es el multiconjunto obtenido eliminando una
ocurrencia de x en el multiconjunto ys. Por ejemplo,
  n0currencias 5 (eliminaUna 5 [2;1;5;4;1]) = 0.
  n0currencias 4 (eliminaUna 5 [2;1;4;5;1;4]) = 2.
  n0currencias 5 (eliminaUna 5 [2;1;5;4;5;1;4]) = 1.
----- *)
```

```
Fixpoint eliminaUna (x:nat) (ys:multiconjunto) : multiconjunto :=
match ys with
| nil      => nil
| y :: ys' => if iguales_nat y x
               then ys'
               else y :: eliminaUna x ys'
end.
```

**Example** prop\_eliminaUna1: n0currencias 5 (eliminaUna 5 [2;1;5;4;1]) = 0.

**Proof.** reflexivity. Qed.

**Example** prop\_eliminaUna2: n0currencias 5 (eliminaUna 5 [2;1;4;1]) = 0.

**Proof.** reflexivity. Qed.

**Example** prop\_eliminaUna3: n0currencias 4 (eliminaUna 5 [2;1;4;5;1;4]) = 2.

**Proof.** reflexivity. Qed.

**Example** prop\_eliminaUna4: n0currencias 5 (eliminaUna 5 [2;1;5;4;5;1;4]) = 1.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 2.7.7. Definir la función
```



```

    eliminaTodas : nat -> multiconjunto -> multiconjunto
    tal que (eliminaTodas x ys) es el multiconjunto obtenido eliminando
    todas las ocurrencias de x en el multiconjunto ys. Por ejemplo,
    nOcurrencias 5 (eliminaTodas 5 [2;1;5;4;1])      = 0.
    nOcurrencias 5 (eliminaTodas 5 [2;1;4;1])        = 0.
    nOcurrencias 4 (eliminaTodas 5 [2;1;4;5;1;4])    = 2.
    nOcurrencias 5 (eliminaTodas 5 [2;1;5;4;5;1;4;5;1;4]) = 0.
    ----- *)

```

```

Fixpoint eliminaTodas (x:nat) (ys:multiconjunto) : multiconjunto :=
  match ys with
  | nil      => nil
  | y :: ys' => if iguales_nat y x
                then eliminaTodas x ys'
                else y :: eliminaTodas x ys'
  end.

```

**Example** prop\_eliminaTodas1: nOcurrencias 5 (eliminaTodas 5 [2;1;5;4;1]) = 0.

**Proof.** **reflexivity.** **Qed.**

**Example** prop\_eliminaTodas2: nOcurrencias 5 (eliminaTodas 5 [2;1;4;1]) = 0.

**Proof.** **reflexivity.** **Qed.**

**Example** prop\_eliminaTodas3: nOcurrencias 4 (eliminaTodas 5 [2;1;4;5;1;4]) = 2.

**Proof.** **reflexivity.** **Qed.**

**Example** prop\_eliminaTodas4: nOcurrencias 5 (eliminaTodas 5 [1;5;4;5;4;5;1]) = 0.

**Proof.** **reflexivity.** **Qed.**

```

(* -----
   Ejercicio 2.7.8. Definir la función
   submulticonjunto : multiconjunto -> multiconjunto -> bool
   tal que (submulticonjunto xs ys) se verifica si xs es un
   submulticonjunto de ys. Por ejemplo,
   submulticonjunto [1;2] [2;1;4;1] = true.
   submulticonjunto [1;2;2] [2;1;4;1] = false.
   ----- *)

```

```

Fixpoint submulticonjunto (xs:multiconjunto) (ys:multiconjunto) : bool :=
  match xs with

```

```

| nil      => true
| x::xs'   => pertenece x ys && submulticonjunto xs' (eliminaUna x ys)
end.

```

**Example** prop\_submulticonjunto1: submulticonjunto [1;2] [2;1;4;1] = true.

**Proof.** reflexivity. Qed.

**Example** prop\_submulticonjunto2: submulticonjunto [1;2;2] [2;1;4;1] = false.

**Proof.** reflexivity. Qed.

```

(* -----
   Ejercicio 2.7.9. Escribir una propiedad sobre multiconjuntos con las
   funciones n0currencias y demostrarla.
   ----- *)

```

**Theorem** n0currencias\_conc: forall xs ys : multiconjunto, forall n:nat,  
n0currencias n (conc xs ys) = n0currencias n xs + n0currencias n ys.

**Proof.**

```

intros xs ys n.                                     (* xs, ys : multiconjunto
                                                       n : nat
                                                       =====
                                                       n0currencias n (xs ++ ys) =
                                                       n0currencias n xs + n0currencias n ys *)

induction xs as [|x xs' HI].
-
  (* ys : multiconjunto
     n : nat
     =====
     n0currencias n ([ ] ++ ys) =
     n0currencias n [ ] + n0currencias n ys *)
  (* n0currencias n ys = n0currencias n ys *)

  simpl.
  reflexivity.
-
  (* x : nat
     xs' : ListaNat
     ys : multiconjunto
     n : nat
     HI : n0currencias n (xs' ++ ys) =
          n0currencias n xs' + n0currencias n ys
     =====
     n0currencias n ((x :: xs') ++ ys) =
     n0currencias n (x :: xs') +

```

```

                                n0currencias n ys *)
simpl.                        (* (if iguales_nat x n
                                then S (n0currencias n (xs' ++ ys))
                                else n0currencias n (xs' ++ ys)) =
                                (if iguales_nat x n
                                then S (n0currencias n xs')
                                else n0currencias n xs') +
                                n0currencias n ys *)

destruct (iguales_nat x n).
+                               (* S (n0currencias n (xs' ++ ys)) =
                                S (n0currencias n xs') +
                                n0currencias n ys *)

simpl.                       (* S (n0currencias n (xs' ++ ys)) =
                                S (n0currencias n xs' +
                                n0currencias n ys) *)

rewrite HI.                   (* S (n0currencias n xs' + n0currencias n ys) =
                                S (n0currencias n xs' + n0currencias n ys) *)

reflexivity.
+                               (* n0currencias n (xs' ++ ys) =
                                n0currencias n xs' + n0currencias n ys *)

rewrite HI.                   (* n0currencias n xs' + n0currencias n ys =
                                n0currencias n xs' + n0currencias n ys *)

reflexivity.

Qed.

(* =====
  § 3. Razonamiento sobre listas
  ===== *)

(* =====
  §§ 3.1. Demostraciones por simplificación
  ===== *)

(* -----
  Ejemplo 3.1.1. Demostrar que, para toda lista de naturales xs,
    [] ++ xs = xs
  ----- *)

Theorem nil_conc : forall xs:ListNat,
  [] ++ xs = xs.

```

**Proof.**

**reflexivity.**

**Qed.**

```
(* =====
  §§ 3.2. Demostraciones por casos
  ===== *)
```

```
(* -----
  Ejemplo 3.2.1. Demostrar que, para toda lista de naturales xs,
    pred (longitud xs) = longitud (resto xs)
  ----- *)
```

**Theorem** resto\_longitud\_pred : **forall** xs:ListaNat,  
pred (longitud xs) = longitud (resto xs).

**Proof.**

```
intros xs.                (* xs : ListaNat
                           =====
                           Nat.pred (longitud xs) = longitud (resto xs) *)

destruct xs as [|x xs'].
-
  (*
    =====
    Nat.pred (longitud []) = longitud (resto []) *)

  reflexivity.
-
  (* x : nat
     xs' : ListaNat
     =====
     Nat.pred (longitud (x :: xs')) =
       longitud (resto (x :: xs')) *)

  reflexivity.
```

**Qed.**

```
(* =====
  §§ 3.3. Demostraciones por inducción
  ===== *)
```

```
(* -----
  Ejemplo 3.3.1. Demostrar que la concatenación de listas de naturales
  es asociativa; es decir,
    (xs ++ ys) ++ zs = xs ++ (ys ++ zs).
```

```

----- *)

Theorem conc_asociativa: forall xs ys zs : ListaNat,
  (xs ++ ys) ++ zs = xs ++ (ys ++ zs).
Proof.
  intros xs ys zs.
  (* xs, ys, zs : ListaNat
     =====
     (xs ++ ys) ++ zs = xs ++ (ys ++ zs) *)

  induction xs as [|x xs' HI].
  -
    (* ys, zs : ListaNat
       =====
       ([ ] ++ ys) ++ zs = [ ] ++ (ys ++ zs) *)

    reflexivity.
  -
    (* x : nat
       xs', ys, zs : ListaNat
       HI : (xs' ++ ys) ++ zs = xs' ++ (ys ++ zs)
       =====
       ((x :: xs') ++ ys) ++ zs =
        (x :: xs') ++ (ys ++ zs) *)

    simpl.
    (* (x :: (xs' ++ ys)) ++ zs =
       x :: (xs' ++ (ys ++ zs)) *)

    rewrite -> HI.
    (* x :: (xs' ++ (ys ++ zs)) =
       x :: (xs' ++ (ys ++ zs)) *)

    reflexivity.
Qed.

(* -----
   Ejemplo 3.3.2. Definir la función
   inversa : ListaNat -> ListaNat
   tal que (inversa xs) es la inversa de xs. Por ejemplo,
   inversa [1;2;3] = [3;2;1].
   inversa nil = nil.

   Nota. La función inversa es equivalente a la predefinida rev.
   ----- *)

Fixpoint inversa (xs:ListaNat) : ListaNat :=
  match xs with
  | nil => nil
  | x::xs' => inversa xs' ++ [x]

```

**end.**

**Example** prop\_inversa1: inversa [1;2;3] = [3;2;1].

**Proof.** **reflexivity.** **Qed.**

**Example** prop\_inversa2: inversa nil = nil.

**Proof.** **reflexivity.** **Qed.**

```
(* -----
   Ejemplo 3.3.3. Demostrar que
       longitud (inversa xs) = longitud xs
   ----- *)

(* 1º intento *)
Theorem longitud_inversa1: forall xs:ListaNat,
  longitud (inversa xs) = longitud xs.
Proof.
  intros xs.
  induction xs as [|x xs' HI].
  -
    (* =====
       longitud (inversa [ ]) = longitud [ ] *)

    reflexivity.

  -
    (* x : nat
       xs' : ListaNat
       HI : longitud (inversa xs') = longitud xs'
       =====
       longitud (inversa (x :: xs')) =
         longitud (x :: xs') *)

    simpl.
    (* longitud (inversa xs' ++ [x]) =
       S (longitud xs') *)

    rewrite <- HI.
    (* longitud (inversa xs' ++ [x]) =
       S (longitud (inversa xs')) *)

Abort.

(* Nota: Para simplificar la última expresión se necesita el siguiente lema. *)

Lemma longitud_conc : forall xs ys : ListaNat,
  longitud (xs ++ ys) = longitud xs + longitud ys.
Proof.
```

```

intros xs ys.
(* xs, ys : ListaNat
=====
longitud (xs ++ ys) =
longitud xs + longitud ys *)

induction xs as [| x xs' HI].
-
(* ys : ListaNat
=====
longitud ([ ] ++ ys) =
longitud [ ] + longitud ys *)

reflexivity.
-
(* x : nat
xs', ys : ListaNat
HI : longitud (xs' ++ ys) =
longitud xs' + longitud ys
=====
longitud ((x :: xs') ++ ys) =
longitud (x :: xs') + longitud ys *)

simpl.
(* S (longitud (xs' ++ ys)) =
S (longitud xs' + longitud ys) *)

rewrite -> HI.
(* S (longitud xs' + longitud ys) =
S (longitud xs' + longitud ys) *)

reflexivity.
Qed.

(* 2º intento *)
Theorem longitud_inversa : forall xs:ListaNat,
longitud (inversa xs) = longitud xs.
Proof.
intros xs.
(* xs : ListaNat
=====
longitud (inversa xs) = longitud xs *)

induction xs as [| x xs' HI].
-
(*
=====
longitud (inversa [ ]) = longitud [ ] *)

reflexivity.
-
(* x : nat
xs' : ListaNat
HI : longitud (inversa xs') = longitud xs'
=====

```

```

                                longitud (inversa (x :: xs')) =
                                longitud (x :: xs') *)
simpl.                        (* longitud (inversa xs' ++ [x]) =
                                S (longitud xs') *)
rewrite longitud_conc.        (* longitud (inversa xs') + longitud [x] =
                                S (longitud xs') *)
rewrite HI.                  (* longitud xs' + longitud [x] =
                                S (longitud xs') *)
simpl.                      (* longitud xs' + 1 = S (longitud xs') *)
rewrite suma_conmutativa.    (* 1 + longitud xs' = S (longitud xs') *)
reflexivity.
Qed.

(* =====
   §§ 3.4. Ejercicios
   ===== *)

(* -----
   Ejercicio 3.4.1. Demostrar que la lista vacía es el elemento neutro
   por la derecha de la concatenación de listas.
   ----- *)

Theorem conc_nil: forall xs:ListaNat,
  xs ++ [] = xs.
Proof.
  intros xs.                      (* xs : ListaNat
                                   =====
                                   xs ++ [ ] = xs *)

  induction xs as [| x xs' HI].
  -
    (* =====
       [ ] ++ [ ] = [ ] *)

    reflexivity.
  -
    (* x : nat
       xs' : ListaNat
       HI : xs' ++ [ ] = xs'
       =====
       (x :: xs') ++ [ ] = x :: xs' *)

    simpl.                      (* x :: (xs' ++ [ ]) = x :: xs' *)
    rewrite HI.                  (* x :: xs' = x :: xs' *)

```



**reflexivity.**  
**Qed.**

```
(* -----
   Ejercicio 3.4.2. Demostrar que inversa es un endomorfismo en
   (ListaNat,++); es decir,
       inversa (xs ++ ys) = inversa ys ++ inversa xs.
   ----- *)
```

**Theorem** inversa\_conc: **forall** xs ys : ListaNat,  
 inversa (xs ++ ys) = inversa ys ++ inversa xs.

**Proof.**

```
intros xs ys. (* xs, ys : ListaNat
               =====
               inversa (xs ++ ys) =
               inversa ys ++ inversa xs *)

induction xs as [|x xs' HI].
- (* ys : ListaNat
   =====
   inversa ([ ] ++ ys) =
   inversa ys ++ inversa [ ] *)
simpl. (* inversa ys = inversa ys ++ [ ] *)
rewrite conc_nil. (* inversa ys = inversa ys *)
reflexivity.

- (* x : nat
   xs', ys : ListaNat
   HI : inversa (xs' ++ ys) =
       inversa ys ++ inversa xs'
   =====
   inversa ((x :: xs') ++ ys) =
   inversa ys ++ inversa (x :: xs') *)
simpl. (* inversa (xs' ++ ys) ++ [x] =
   inversa ys ++ (inversa xs' ++ [x]) *)
rewrite HI. (* (inversa ys ++ inversa xs') ++ [x] =
   inversa ys ++ (inversa xs' ++ [x]) *)
rewrite conc_asociativa. (* inversa ys ++ (inversa xs' ++ [x]) =
   inversa ys ++ (inversa xs' ++ [x]) *)
reflexivity.
Qed.
```

```
(* -----
  Ejercicio 3.4.3. Demostrar que inversa es involutiva; es decir,
    inversa (inversa xs) = xs.
  ----- *)
```

**Theorem** inversa\_involutiva: **forall** xs:ListaNat,  
inversa (inversa xs) = xs.

**Proof.**

**induction** xs **as** [|x xs' HI].

```
- (*
    =====
    inversa (inversa [ ]) = [ ] *)

  reflexivity.

- (* x : nat
    xs' : ListaNat
    HI : inversa (inversa xs') = xs'
    =====
    inversa (inversa (x :: xs')) = x :: xs' *)
  simpl. (* inversa (inversa xs' ++ [x]) = x :: xs' *)
  rewrite inversa_conc. (* inversa [x] ++ inversa (inversa xs') =
    x :: xs' *)
  simpl. (* x :: inversa (inversa xs') = x :: xs' *)
  rewrite HI. (* x :: xs' = x :: xs' *)
  reflexivity.
```

**Qed.**

```
(* -----
  Ejercicio 3.4.4. Demostrar que
    xs ++ (ys ++ (zs ++ vs)) = ((xs ++ ys) ++ zs) ++ vs.
  ----- *)
```

**Theorem** conc\_asociativa4 : **forall** xs ys zs vs : ListaNat,  
xs ++ (ys ++ (zs ++ vs)) = ((xs ++ ys) ++ zs) ++ vs.

**Proof.**

```
intros xs ys zs vs. (* xs, ys, zs, vs : ListaNat
    =====
    xs ++ (ys ++ (zs ++ vs)) =
    ((xs ++ ys) ++ zs) ++ vs *)
rewrite conc_asociativa. (* xs ++ (ys ++ (zs ++ vs)) =
    (xs ++ ys) ++ (zs ++ vs) *)
```

```

rewrite conc_asociativa. (* xs ++ (ys ++ (zs ++ vs)) =
                           xs ++ (ys ++ (zs ++ vs)) *)

reflexivity.
Qed.

(* -----
   Ejercicio 3.4.5. Demostrar que al concatenar dos listas no aparecen ni
   desaparecen ceros.
   ----- *)

Lemma noCeros_conc : forall xs ys : ListaNat,
  noCeros (xs ++ ys) = (noCeros xs) ++ (noCeros ys).
Proof.
  intros xs ys. (* xs, ys : ListaNat
                  =====
                  noCeros (xs ++ ys) =
                  noCeros xs ++ noCeros ys *)

  induction xs as [|x xs' HI].
  - (* ys : ListaNat
      =====
      noCeros ([]) ++ ys =
      noCeros [] ++ noCeros ys *)

    reflexivity.
  - (* x : nat
      xs', ys : ListaNat
      HI : noCeros (xs' ++ ys) =
          noCeros xs' ++ noCeros ys
      =====
      noCeros ((x :: xs') ++ ys) =
      noCeros (x :: xs') ++ noCeros ys *)

    destruct x.
    + (* noCeros ((0 :: xs') ++ ys) =
        noCeros (0 :: xs') ++ noCeros ys *)

      simpl.
      (* noCeros (xs' ++ ys) =
         noCeros xs' ++ noCeros ys *)

      rewrite HI.
      (* noCeros xs' ++ noCeros ys =
         noCeros xs' ++ noCeros ys *)

      reflexivity.
    + (* noCeros ((S x :: xs') ++ ys) =
        noCeros (S x :: xs') ++ noCeros ys *)

```

```

simpl.                                (* S x :: noCeros (xs' ++ ys) =
                                         (S x :: noCeros xs') ++ noCeros ys *)
rewrite HI.                            (* S x :: (noCeros xs' ++ noCeros ys) =
                                         (S x :: noCeros xs') ++ noCeros ys *)
reflexivity.
Qed.

(* -----
Ejercicio 3.4.6. Definir la función
    iguales_lista : ListaNat -> ListaNat -> bool
tal que (iguales_lista xs ys) se verifica si las listas xs e ys son
iguales. Por ejemplo,
    iguales_lista nil nil           = true.
    iguales_lista [1;2;3] [1;2;3] = true.
    iguales_lista [1;2;3] [1;2;4] = false.
----- *)

Fixpoint iguales_lista (xs ys : ListaNat) : bool :=
  match xs, ys with
  | nil,    nil      => true
  | x::xs', y::ys'  => iguales_nat x y && iguales_lista xs' ys'
  | _, _          => false
end.

Example prop_iguales_lista1: (iguales_lista nil nil = true).
Proof. reflexivity. Qed.

Example prop_iguales_lista2: iguales_lista [1;2;3] [1;2;3] = true.
Proof. reflexivity. Qed.

Example prop_iguales_lista3: iguales_lista [1;2;3] [1;2;4] = false.
Proof. reflexivity. Qed.

(* -----
Ejercicio 3.4.7. Demostrar que la igualdad de listas cumple la
propiedad reflexiva.
----- *)

Theorem iguales_lista_refl : forall xs:ListaNat,
  iguales_lista xs xs = true.

```

**Proof.**

```

induction xs as [|x xs' HI].
-
    (*
    =====
    iguales_lista [ ] [ ] = true *)

    reflexivity.
-
    (* x : nat
       xs' : ListaNat
       HI : iguales_lista xs' xs' = true
       =====
       iguales_lista (x :: xs') (x :: xs') = true *)

    simpl.
    (* iguales_nat x x &&
       iguales_lista xs' xs' = true *)

    rewrite HI.
    rewrite iguales_nat_refl.
    reflexivity.

```

**Qed.**

```

(* -----
   Ejercicio 3.4.8. Demostrar que al incluir un elemento en un
   multiconjunto, ese elemento aparece al menos una vez en el
   resultado.
   ----- *)

```

**Theorem** n0currencias\_agrega: **forall** (x:nat) (xs:multiconjunto),  
 menor\_o\_igual 1 (n0currencias x (agrega x xs)) = **true**.

**Proof.**

```

intros x xs.
    (* x : nat
       xs : multiconjunto
       =====
       menor_o_igual 1 (n0currencias x (agrega x xs)) =
       true *)

    simpl.
    (* match
       (if iguales_nat x x then S (n0currencias x xs)
        else n0currencias x xs)

       with
       | 0 => false
       | S _ => true
       end =
       true *)

```

```

rewrite iguales_nat_refl. (* true = true *)
reflexivity.
Qed.

```

```

(* -----
   Ejercicio 3.4.9. Demostrar que cada número natural es menor o igual
   que su siguiente.
   ----- *)

```

**Theorem** menor\_o\_igual\_n\_Sn: **forall** n:**nat**,  
 menor\_o\_igual n (S n) = **true**.

**Proof.**

```

intros n. (* n : nat
           =====
           menor_o_igual n (S n) = true *)

induction n as [|n' HI].
- (*
   =====
   menor_o_igual 0 1 = true *)

  reflexivity.
- (* n' : nat
   HI : menor_o_igual n' (S n') = true
   =====
   menor_o_igual (S n') (S (S n')) = true *)

  simpl. (* menor_o_igual n' (S n') = true *)
  rewrite HI. (* true = true *)
  reflexivity.
Qed.

```

```

(* -----
   Ejercicio 3.4.10. Demostrar que al borrar una ocurrencia de 0 de un
   multiconjunto el número de ocurrencias de 0 en el resultado es menor
   o igual que en el original.
   ----- *)

```

**Theorem** remove\_decreases\_n0currencias: **forall** (xs : multiconjunto),  
 menor\_o\_igual (n0currencias 0 (eliminaUna 0 xs)) (n0currencias 0 xs) = **true**.

**Proof.**

```

induction xs as [|x xs' HI].
- (*

```

```

=====
menor_o_igual (n0currencias 0 (eliminaUna 0
                               (n0currencias 0 []))
              = true *)

reflexivity.
-
(* x : nat
   xs' : ListaNat
   HI: menor_o_igual (n0currencias 0 (eliminaUna 0
                                           (n0currencias 0 xs'))
                     = true
   =====
   menor_o_igual (n0currencias 0 (eliminaUna 0
                                           (n0currencias 0 (x :: xs'))
                     = true *)

destruct x.
+
(* menor_o_igual (n0currencias 0 (eliminaUna 0
                                     (n0currencias 0 (0 :: xs'))
                       = true *)

simpl.
(* menor_o_igual (n0currencias 0 xs')
   (S (n0currencias 0 xs'))
   = true *)

rewrite menor_o_igual_n_Sn. (* true = true *)
reflexivity.
+
(* menor_o_igual (n0currencias 0
                  (eliminaUna 0 (S x :: xs')))
   (n0currencias 0 (S x :: xs'))
   = true *)

simpl.
(* menor_o_igual (n0currencias 0 (eliminaUna 0
                                     (n0currencias 0 xs'))
                       = true *)

rewrite HI.
reflexivity.

Qed.

(* -----
   Ejercicio 3.4.11. Escribir un teorema con las funciones n0currencias
   y suma de los multiconjuntos.
   ----- *)

```

**Theorem** n0currencias\_suma:

```

forall x : nat, forall xs ys : multiconjunto,
  n0currencias x (suma xs ys) = n0currencias x xs + n0currencias x ys.
Proof.
intros x xs ys.
induction xs as [|x' xs' HI].
-
  reflexivity.
-
  simpl.
destruct (iguales_nat x' x).
+
  rewrite HI.
  reflexivity.
+
  rewrite HI.

```

```

(* x : nat
   xs, ys : multiconjunto
   =====
   n0currencias x (suma xs ys) =
   n0currencias x xs + n0currencias x ys *)

(* x : nat
   ys : multiconjunto
   =====
   n0currencias x (suma [ ] ys) =
   n0currencias x [ ] + n0currencias x ys *)

(* x, x' : nat
   xs' : ListaNat
   ys : multiconjunto
   HI : n0currencias x (suma xs' ys) =
        n0currencias x xs' + n0currencias x ys
   =====
   n0currencias x (suma (x' :: xs') ys) =
   n0currencias x (x' :: xs') + n0currencias x ys *)

(* (if iguales_nat x' x
    then S (n0currencias x (suma xs' ys))
    else n0currencias x (suma xs' ys))
   =
   (if iguales_nat x' x
    then S (n0currencias x xs')
    else n0currencias x xs') + n0currencias x ys *)

(* S (n0currencias x (suma xs' ys)) =
   S (n0currencias x xs') + n0currencias x ys *)

(* S (n0currencias x xs' + n0currencias x ys) =
   S (n0currencias x xs') + n0currencias x ys *)

(* n0currencias x (suma xs' ys) =
   n0currencias x xs' + n0currencias x ys *)

(* n0currencias x xs' + n0currencias x ys =
   n0currencias x xs' + n0currencias x ys *)

```



**reflexivity.**

**Qed.**

```
(* -----
Ejercicio 3.4.12. Demostrar que la función inversa es inyectiva; es
decir,
  forall (xs ys : ListaNat), inversa xs = inversa ys -> xs = ys.
----- *)
```

**Theorem** inversa\_inyectiva: **forall** (xs ys : ListaNat),  
inversa xs = inversa ys -> xs = ys.

**Proof.**

```
intros xs ys H.                                (* xs, ys : ListaNat
                                                H : inversa xs = inversa ys
                                                =====
                                                xs = ys *)
rewrite <- inversa_involutiva. (* xs = inversa (inversa ys) *)
rewrite <- H.                  (* xs = inversa (inversa xs) *)
rewrite inversa_involutiva.    (* xs = xs *)
reflexivity.
```

**Qed.**

```
(* =====
§ 4. Opcionales
===== *)
```

```
(* -----
Ejemplo 4.1. Definir el tipo OpcionalNat con los constructores
  Some : nat -> OpcionalNat
  None : OpcionalNat.
----- *)
```

**Inductive** OpcionalNat : **Type** :=

```
| Some : nat -> OpcionalNat
| None : OpcionalNat.
```

```
(* -----
Ejemplo 4.2. Definir la función
  nthOpcional : ListaNat -> nat -> OpcionalNat
tal que (nthOpcional xs n) es el n-ésimo elemento de la lista xs o None
```

*si la lista tiene menos de  $n$  elementos. Por ejemplo,*  
 *$\text{nthOpcional } [4;5;6;7] \ 0 = \text{Some } 4.$*   
 *$\text{nthOpcional } [4;5;6;7] \ 3 = \text{Some } 7.$*   
 *$\text{nthOpcional } [4;5;6;7] \ 9 = \text{None}.$*

----- \*)

```
Fixpoint nthOpcional (xs:ListaNat) (n:nat) : OpcionalNat :=
match xs with
| nil      => None
| x :: xs' => match iguales_nat n 0 with
| true  => Some x
| false => nthOpcional xs' (pred n)
end
end.
```

**Example** prop\_nthOpcional1 : nthOpcional [4;5;6;7] 0 = Some 4.

**Proof.** reflexivity. **Qed**.

**Example** prop\_nthOpcional2 : nthOpcional [4;5;6;7] 3 = Some 7.

**Proof.** reflexivity. **Qed**.

**Example** prop\_nthOpcional3 : nthOpcional [4;5;6;7] 9 = None.

**Proof.** reflexivity. **Qed**.

(\* Introduciendo condicionales nos queda: \*)

```
Fixpoint nthOpcional' (xs:ListaNat) (n:nat) : OpcionalNat :=
match xs with
| nil      => None
| x :: xs' => if iguales_nat x 0
|         => then Some x
|         => else nthOpcional' xs' (pred n)
end.
```

(\* -----

*Ejemplo 4.3. Definir la función*

*eliminaOpcionalNat -> OpcionalNat -> nat*

*tal que (option\_elim d o) es el valor de o, si o tiene valor o es d en caso contrario. Por ejemplo,*

*eliminaOpcionalNat 3 (Some 7) = 7*

*eliminaOpcionalNat 3 None = 3*

```

----- *)

Definition eliminaOpcionalNat (d : nat) (o : OpcionalNat) : nat :=
  match o with
  | Some n' => n'
  | None    => d
  end.

Compute (eliminaOpcionalNat 3 (Some 7)).
(* ==> 7 : nat *)
Compute (eliminaOpcionalNat 3 None).
(* ==> 3 : nat *)

(* -----
   Ejercicio 4.1. Definir la función
   primeroOpcional : ListaNat -> OpcionalNat
   tal que (primeroOpcional xs) es el primer elemento de xs, si xs es no
   vacía; o es None, en caso contrario. Por ejemplo,
   primeroOpcional []      = None.
   primeroOpcional [1]     = Some 1.
   primeroOpcional [5;6] = Some 5.
   ----- *)

Definition primeroOpcional (xs : ListaNat) : OpcionalNat :=
  match xs with
  | nil      => None
  | x::xs'   => Some x
  end.

Example prop_primeroOpcional1 : primeroOpcional [] = None.
Proof. reflexivity. Qed.

Example prop_primeroOpcional2 : primeroOpcional [1] = Some 1.
Proof. reflexivity. Qed.

Example prop_primeroOpcional3 : primeroOpcional [5;6] = Some 5.
Proof. reflexivity. Qed.

(* -----
   Ejercicio 4.2. Demostrar que

```

```

    primero d xs = eliminaOpcionalNat d (primeroOpcional xs).
----- *)

Theorem primero_primeroOpcional: forall (xs:ListaNat) (d:nat),
  primero d xs = eliminaOpcionalNat d (primeroOpcional xs).
Proof.
  intros xs d.          (* xs : ListaNat
                        d : nat
                        =====
                        primero d xs = eliminaOpcionalNat d (primeroOpcional xs) *)

  destruct xs as [|x xs'].
-
  (* d : nat
  =====
  primero d [] = eliminaOpcionalNat d (primeroOpcional []) *)

  reflexivity.
-
  (* x : nat
  xs' : ListaNat
  d : nat
  =====
  primero d (x :: xs') =
  eliminaOpcionalNat d (primeroOpcional (x :: xs')) *)

  simpl.
  reflexivity.
Qed.

(* -----
  Nota. Finalizar el módulo ListaNat.
  ----- *)

End ListaNat.

(* =====
  § 5. Diccionarios (o funciones parciales)
  ===== *)

(* -----
  Ejemplo 5.1. Definir el tipo id (por identificador) con el
  constructor
  Id : nat -> id.
  ----- *)

```

```
Inductive id : Type :=
| Id : nat -> id.
```

```
(* -----
Ejemplo 5.2. Definir la función
    iguales_id : id -> id -> bool
tal que (iguales_id x1 x2) se verifica si tienen la misma clave. Por
ejemplo,
    iguales_id (Id 3) (Id 3) = true : bool
    iguales_id (Id 3) (Id 4) = false : bool
----- *)
```

```
Definition iguales_id (x1 x2 : id) :=
match x1, x2 with
| Id n1, Id n2 => iguales_nat n1 n2
end.
```

```
Compute (iguales_id (Id 3) (Id 3)).
```

```
(* ==> true : bool *)
```

```
Compute (iguales_id (Id 3) (Id 4)).
```

```
(* ==> false : bool *)
```

```
(* -----
Ejercicio 5.1. Demostrar que iguales_id es reflexiva.
----- *)
```

```
Theorem iguales_id_refl : forall x:id, iguales_id x x = true.
```

```
Proof.
```

```
  intro x.                                (* x : id
                                           =====
                                           iguales_id x x = true *)
  destruct x.                             (* iguales_id (Id n) (Id n) = true *)
  simpl.                                  (* iguales_nat n n = true *)
  rewrite iguales_nat_refl. (* true = true *)
  reflexivity.
```

```
Qed.
```

```
(* -----
Ejemplo 5.3. Iniciar el módulo Diccionario que importa a ListaNat.
```

```

----- *)

Module Diccionario.
Export ListaNat.

(* -----
   Ejemplo 5.4. Definir el tipo diccionario con los constructores
       vacio      : diccionario
       registro : id -> nat -> diccionario -> diccionario.
   ----- *)

Inductive diccionario : Type :=
| vacio      : diccionario
| registro : id -> nat -> diccionario -> diccionario.

(* -----
   Ejemplo 5.5. Definir los diccionarios cuyos elementos son
       + []
       + [(3,6)]
       + [(2,4), (3,6)]
   ----- *)

Definition diccionario1 := vacio.
Definition diccionario2 := registro (Id 3) 6 diccionario1.
Definition diccionario3 := registro (Id 2) 4 diccionario2.

(* -----
   Ejemplo 5.6. Definir la función
       valor : id -> diccionario -> OpcionalNat
   tal que (valor i d) es el valor de la entrada de d con clave i, o
   None si d no tiene ninguna entrada con clave i. Por ejemplo,
       valor (Id 2) diccionario3 = Some 4
       valor (Id 2) diccionario2 = None
   ----- *)

Fixpoint valor (x : id) (d : diccionario) : OpcionalNat :=
match d with
| vacio      => None
| registro y v d' => if iguales_id x y
                     then Some v

```

```

        else valor x d'
    end.

```

```

Compute (valor (Id 2) diccionario3).
(* = Some 4 : OpcionalNat *)
Compute (valor (Id 2) diccionario2).
(* = None : OpcionalNat*)

```

```

(* -----
   Ejemplo 5.7. Definir la función
       actualiza : diccionario -> id -> nat -> diccionario
   tal que (actualiza d x v) es el diccionario obtenido a partir del d
   + si d tiene un elemento con clave x, le cambia su valor a v
   + en caso contrario, le añade el elemento v con clave x
   ----- *)

```

```

Definition actualiza (d : diccionario)
              (x : id) (v : nat)
              : diccionario :=
    registro x v d.

```

```

(* -----
   Ejercicio 5.2. Demostrar que
       forall (d : diccionario) (x : id) (v: nat),
       valor x (actualiza d x v) = Some v.
   ----- *)

```

```

Theorem valor_actualiza: forall (d : diccionario) (x : id) (v: nat),
    valor x (actualiza d x v) = Some v.

```

**Proof.**

```

    intros d x v.
    (* d : diccionario
       x : id
       v : nat
       =====
       valor x (actualiza d x v) = Some v *)
    destruct x.
    simpl.
    (* valor (Id n) (actualiza d (Id n) v) = Some v *)
    (* (if iguales_nat n n then Some v
        else valor (Id n) d)
        = Some v *)
    rewrite iguales_nat_refl. (* Some v = Some v *)

```

**reflexivity.**  
**Qed.**

```
(* -----
Ejercicio 5.3. Demostrar que
  forall (d : diccionario) (x y : id) (o: nat),
    iguales_id x y = false -> valor x (actualiza d y o) = valor x d.
----- *)
```

**Theorem** actualiza\_neq :  
**forall** (d : diccionario) (x y : id) (o: **nat**),  
 iguales\_id x y = **false** -> valor x (actualiza d y o) = valor x d.

**Proof.**

```
intros d x y o p. (* d : diccionario
                     x, y : id
                     o : nat
                     p : iguales_id x y = false
                     =====
                     valor x (actualiza d y o) = valor x d *)
simpl. (* (if iguales_id x y then Some o
            else valor x d)
            = valor x d *)
rewrite p. (* valor x d = valor x d *)
reflexivity.
```

**Qed.**

```
(* -----
Ejemplo 5.8. Finalizar el módulo Diccionario
----- *)
```

**End** Diccionario.

```
(* =====
§ Bibliografía
===== *)
```

```
(*
+ "Working with structured data" de Peirce et als.
  http://bit.ly/2LQABsv
*)
```



# Tema 4

## Polimorfismo y funciones de orden superior en Coq

*(\* T4: Polimorfismo y funciones de orden superior en Coq \*)*

**Require Export** T3\_Listas.

*(\* El contenido de la teoría es*

- 1. Polimorfismo*
  - 1. Listas polimórficas*
    - 1. Inferencia de tipos*
    - 2. Síntesis de los tipos de los argumentos*
    - 3. Argumentos implícitos*
    - 4. Explicitación de argumentos*
    - 5. Ejercicios*
  - 2. Polimorfismo de pares*
  - 3. Resultados opcionales polimórficos*
- 2. Funciones como datos*
  - 1. Funciones de orden superior*
  - 2. Filtrado*
  - 3. Funciones anónimas*
  - 4. Aplicación a todos los elementos (map)*
  - 5. Plegados (fold)*
  - 6. Funciones que construyen funciones*
- 3. Ejercicios*
- 4. Bibliografía*

*\*)*

*(\* =====*

### § 1. Polimorfismo

===== \*)

(\* =====  
 §§ 1.1. Listas polimórficas  
 ===== \*)

(\* -----  
 Nota. Se suprimen algunos avisos.  
 ----- \*)

**Set** Warnings "-notation-overridden,-parsing".

(\* -----  
 Ejemplo 1.1.1. Definir el tipo (list X) para representar las listas  
 de elementos de tipo X con los constructores nil y cons tales que  
 + nil es la lista vacía y  
 + (cons x ys) es la lista obtenida añadiendo el elemento x a la  
 lista ys.  
 ----- \*)

**Inductive** list (X:Type) : Type :=

| nil : list X  
 | cons : X -> list X -> list X.

(\* -----  
 Ejemplo 1.1.2. Calcular el tipo de list.  
 ----- \*)

**Check** list.

(\* ==> list : Type -> Type \*)

(\* -----  
 Ejemplo 1.1.3. Calcular el tipo de (nil nat).  
 ----- \*)

**Check** (nil nat).

(\* ==> nil nat : list nat \*)

(\* -----

*Ejemplo 1.1.4. Calcular el tipo de (cons nat 3 (nil nat)).*

----- \*)

**Check** (cons **nat** 3 (nil **nat**)).

(\* ==> cons nat 3 (nil nat) : list nat \*)

(\* -----

*Ejemplo 1.1.5. Calcular el tipo de nil.*

----- \*)

**Check** nil.

(\* ==> nil : forall X : Type, list X \*)

(\* -----

*Ejemplo 1.1.6. Calcular el tipo de cons.*

----- \*)

**Check** cons.

(\* ==> cons : forall X : Type, X -> list X -> list X \*)

(\* -----

*Ejemplo 1.1.7. Calcular el tipo de  
(cons nat 2 (cons nat 1 (nil nat))).*

----- \*)

**Check** (cons **nat** 2 (cons **nat** 1 (nil **nat**))).

(\* ==> cons nat 2 (cons nat 1 (nil nat)) : list nat \*)

(\* -----

*Ejemplo 1.1.8. Definir la función*

*repite (X : Type) (x : X) (n : nat) : list X*

*tal que (repite X x n) es la lista, de elementos de tipo X, obtenida repitiendo n veces el elemento x. Por ejemplo,*

*repite nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).*

*repite bool false 1 = cons bool false (nil bool).*

----- \*)

**Fixpoint** repite (X : **Type**) (x : X) (n : **nat**) : **list** X :=

**match** n **with**

| 0 => nil X

```
| S n' => cons X x (repite X x n')
end.
```

**Example** prop\_repitel :

```
repite nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
```

**Proof.** reflexivity. Qed.

**Example** prop\_repite2 :

```
repite bool false 1 = cons bool false (nil bool).
```

**Proof.** reflexivity. Qed.

```
(* =====
   §§§ 1.1.1. Inferencia de tipos
   ===== *)
```

```
(* -----
   Ejemplo 1.1.9. Definir la función
       repite' X x n : list X
   tal que (repite' X x n) es la lista obtenida repitiendo n veces el
   elemento x. Por ejemplo,
       repite' nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
       repite' bool false 1 = cons bool false (nil bool).
   ----- *)
```

**Fixpoint** repite' X x n : list X :=

```
match n with
```

```
| 0 => nil X
```

```
| S n' => cons X x (repite' X x n')
```

```
end.
```

```
(* -----
   Ejemplo 1.1.10. Calcular los tipos de repite' y repite.
   ----- *)
```

**Check** repite'.

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

**Check** repite.

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

```
(* =====
```

### §§§ 1.1.2. Síntesis de los tipos de los argumentos

===== \*)

```
(* -----
Ejemplo 1.1.11. Definir la función
  repite'' X x n : list X
tal que (repite'' X x n) es la lista obtenida repitiendo n veces el
elemento x, usando argumentos implícitos. Por ejemplo,
  repite'' nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
  repite'' bool false 1 = cons bool false (nil bool).
----- *)
```

```
Fixpoint repite'' X x n : list X :=
  match n with
  | 0    => nil _
  | S n' => cons _ x (repite'' _ x n')
end.
```

```
(* -----
Ejemplo 1.1.12. Definir la lista formada por los números naturales 1,
2 y 3.
----- *)
```

```
Definition list123 :=
  cons nat 1 (cons nat 2 (cons nat 3 (nil nat))).
```

```
Definition list123' :=
  cons _ 1 (cons _ 2 (cons _ 3 (nil _))).
```

```
(* =====
§§§ 1.1.3. Argumentos implícitos
===== *)
```

```
(* -----
Ejemplo 1.1.13. Especificar las siguientes funciones y sus argumentos
explícitos e implícitos:
+ nil
+ constructor
+ repite
----- *)
```

**Arguments** nil {X}.

**Arguments** cons {X} \_ \_.

**Arguments** repite {X} x n.

```
(* -----
  Ejemplo 1.1.14. Definir la lista formada por los números naturales 1,
  2 y 3.
  ----- *)
```

**Definition** list123'' := cons 1 (cons 2 (cons 3 nil)).

```
(* -----
  Ejemplo 1.1.15. Definir la función
    repite''' {X : Type} (x : X) (n : nat) : list X
  tal que (repite'' X x n) es la lista obtenida repitiendo n veces el
  elemento x, usando argumentos implícitos. Por ejemplo,
    repite'' nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
    repite'' bool false 1 = cons bool false (nil bool).
  ----- *)
```

**Fixpoint** repite''' {X : **Type**} (x : X) (n : **nat**) : **list** X :=  
**match** n **with**  
 | 0 => nil  
 | S n' => cons x (repite''' x n')  
**end**.

**Example** prop\_repite'''1 :  
 repite''' 4 2 = cons 4 (cons 4 nil).

**Proof.** reflexivity. **Qed.**

**Example** prop\_repite'''2 :  
 repite false 1 = cons false nil.

**Proof.** reflexivity. **Qed.**

```
(* -----
  Ejemplo 1.1.16. Definir el tipo (list' {X}) para representar las
  listas de elementos de tipo X con los constructores nil' y cons'
  tales que
  + nil' es la lista vacía y
```

+ (cons' x ys) es la lista obtenida añadiendo el elemento x a la lista ys.

----- \*)

```
Inductive list' {X:Type} : Type :=
| nil'   : list'
| cons'  : X -> list' -> list'.
```

```
(* -----
  Ejemplo 1.1.17. Definir la función
    conc {X : Type} (xs ys : list X) : (list X)
  tal que (conc xs ys) es la concatenación de xs e ys.
  ----- *)
```

```
Fixpoint conc {X : Type} (xs ys : list X) : (list X) :=
match xs with
| nil      => ys
| cons x xs' => cons x (conc xs' ys)
end.
```

```
(* -----
  Ejemplo 1.1.18. Definir la función
    inversa {X:Type} (l:list X) : list X
  tal que (inversa xs) es la inversa de xs. Por ejemplo,
    inversa (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
    inversa (cons true nil)      = cons true nil.
  ----- *)
```

```
Fixpoint inversa {X:Type} (xs:list X) : list X :=
match xs with
| nil      => nil
| cons x xs' => conc (inversa xs') (cons x nil)
end.
```

```
Example prop_inversa1 :
  inversa (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
```

```
Proof. reflexivity. Qed.
```

```
Example prop_inversa2:
  inversa (cons true nil) = cons true nil.
```

**Proof. reflexivity. Qed.**

```
(* -----
  Ejemplo 1.1.19. Definir la función
    longitud {X : Type} (xs : list X) : nat
  tal que (longitud xs) es el número de elementos de xs. Por ejemplo,
    longitud (cons 1 (cons 2 (cons 3 nil))) = 3.
  ----- *)
```

```
Fixpoint longitud {X : Type} (xs : list X) : nat :=
match xs with
| nil      => 0
| cons _ xs' => S (longitud xs')
end.
```

**Example** prop\_longitud1:

longitud (cons 1 (cons 2 (cons 3 nil))) = 3.

**Proof. reflexivity. Qed.**

```
(* =====
  §§§ 1.1.4. Explicitación de argumentos
  ===== *)
```

```
(* -----
  Ejemplo 1.1.20. Evaluar la siguiente expresión
    Fail Definition n_nil := nil.
  ----- *)
```

Fail **Definition** n\_nil := nil.

```
(* ==> Error: Cannot infer the implicit parameter X of nil. *)
```

```
(* -----
  Ejemplo 1.1.21. Completar la definición anterior para obtener la
  lista vacía de números naturales.
  ----- *)
```

```
(* 1ª solución *)
```

**Definition** n\_nil : **list** **nat** := nil.

```
(* 2ª solución *)
```



**Definition** `n_nil'` := @nil **nat**.

```
(* -----
  Ejemplo 1.1.22. Definir las siguientes abreviaturas
  + "x :: y"           para (cons x y)
  + "[ ]"              para nil
  + "[ x ; .. ; y ]"   para (cons x .. (cons y []) ..).
  + "x ++ y"           para (conc x y)
  ----- *)
```

**Notation** `"x :: y"` := (cons x y)  
(at level 60, **right** associativity).

**Notation** `"[ ]"` := nil.

**Notation** `"[ x ; .. ; y ]"` := (cons x .. (cons y []) ..).

**Notation** `"x ++ y"` := (conc x y)  
(at level 60, **right** associativity).

```
(* -----
  Ejemplo 1.1.23. Definir la lista cuyos elementos son 1, 2 y 3.
  ----- *)
```

**Definition** `list123'''` := [1; 2; 3].

```
(* =====
  §§§ 1.1.5. Ejercicios
  ===== *)
```

```
(* -----
  Ejercicio 1.1.1. Demostrar que la lista vacía es el elemento neutro
  por la derecha de la concatenación.
  ----- *)
```

**Theorem** `conc_nil`: **forall** (X:**Type**), **forall** xs:**list** X,  
xs ++ [] = xs.

**Proof.**

**induction** xs **as** [|x xs' HI].

-

(\* X : Type

=====

[ ] ++ [ ] = [ ] \*)

**reflexivity.**

```

-
(* X : Type
   x : X
   xs' : list X
   HI : xs' ++ [ ] = xs'
   =====
   (x :: xs') ++ [ ] = x :: xs' *)
simpl.
rewrite HI.
reflexivity.
Qed.

(* -----
   Ejercicio 1.1.2. Demostrar que la concatenación es asociativa.
   ----- *)

Theorem conc_asociativa : forall A (xs ys zs: list A),
  xs ++ (ys ++ zs) = (xs ++ ys) ++ zs.
Proof.
  intros A xs ys zs.
  (* A : Type
     xs, ys, zs : list A
     =====
     xs ++ (ys ++ zs) = (xs ++ ys) ++ zs *)

  induction xs as [|x xs' HI].
  +
  (* A : Type
     ys, zs : list A
     =====
     [ ] ++ (ys ++ zs) = ([ ] ++ ys) ++ zs *)

  reflexivity.
  +
  (* A : Type
     x : A
     xs', ys, zs : list A
     HI : xs' ++ (ys ++ zs) = (xs' ++ ys) ++ zs
     =====
     (x :: xs') ++ (ys ++ zs) =
     ((x :: xs') ++ ys) ++ zs *)

  simpl.
  (* x :: (xs' ++ (ys ++ zs)) =
     x :: ((xs' ++ ys) ++ zs) *)

  rewrite HI.
  (* x :: ((xs' ++ ys) ++ zs) =
     x :: ((xs' ++ ys) ++ zs) *)

  reflexivity.

```

**Qed.**

```
(* -----
   Ejercicio 1.1.3. Demostrar que la longitud de una concatenación es la
   suma de las longitudes de las listas (es decir, es un homomorfismo).
   ----- *)
```

**Lemma** conc\_longitud: **forall** (X:**Type**) (xs ys : **list** X),  
 longitud (xs ++ ys) = longitud xs + longitud ys.

**Proof.**

```
intros X xs ys. (* X : Type
                  xs, ys : list X
                  =====
                  longitud (xs ++ ys) =
                  longitud xs + longitud ys *)

induction xs as [|x xs' HI].
+ (* X : Type
   ys : list X
   =====
   longitud ([ ] ++ ys) =
   longitud [ ] + longitud ys *)

  reflexivity.

+ (* X : Type
   x : X
   xs', ys : list X
   HI : longitud (xs' ++ ys) =
       longitud xs' + longitud ys
   =====
   longitud ((x :: xs') ++ ys) =
   longitud (x :: xs') + longitud ys *)

  simpl. (* S (longitud (xs' ++ ys)) =
           S (longitud xs' + longitud ys) *)

  rewrite HI. (* S (longitud xs' + longitud ys) =
               S (longitud xs' + longitud ys) *)

  reflexivity.
```

**Qed.**

```
(* -----
   Ejercicio 1.1.4. Demostrar que
   inversa (xs ++ ys) = inversa ys ++ inversa xs.
```

```

----- *)

Theorem inversa_conc: forall X (xs ys : list X),
  inversa (xs ++ ys) = inversa ys ++ inversa xs.
Proof.
  intros X xs ys.
  (* X : Type
     xs, ys : list X
     =====
     inversa (xs ++ ys) =
     inversa ys ++ inversa xs *)

  induction xs as [|x xs' HI].
  -
    (* X : Type
       ys : list X
       =====
       inversa ([ ] ++ ys) =
       inversa ys ++ inversa [ ] *)

    simpl.
    rewrite conc_nil.
    reflexivity.
  -
    (* X : Type
       x : X
       xs', ys : list X
       HI : inversa (xs' ++ ys) =
            inversa ys ++ inversa xs'
       =====
       inversa ((x :: xs') ++ ys) =
       inversa ys ++ inversa (x :: xs') *)

    simpl.
    (* inversa (xs' ++ ys) ++ [x] =
       inversa ys ++ (inversa xs' ++ [x]) *)

    rewrite HI.
    (* (inversa ys ++ inversa xs') ++ [x] =
       inversa ys ++ (inversa xs' ++ [x]) *)

    rewrite conc_asociativa.
    (* (inversa ys ++ inversa xs') ++ [x] =
       (inversa ys ++ inversa xs') ++ [x] *)

    reflexivity.
Qed.

(* -----
   Ejercicio 1.1.5. Demostrar que la inversa es involutiva; es decir,
   inversa (inversa xs) = xs.
   ----- *)

```

**Theorem** inversa\_involutiva : forall X : Type, forall xs : list X,  
inversa (inversa xs) = xs.

**Proof.**

```

intros X xs.
(* X : Type
   xs : list X
   =====
   inversa (inversa xs) = xs *)

induction xs as [|x xs' HI].
+
(* X : Type
   =====
   inversa (inversa [ ]) = [ ] *)

  reflexivity.
+
(* X : Type
   x : X
   xs' : list X
   HI : inversa (inversa xs') = xs'
   =====
   inversa (inversa (x :: xs')) = x :: xs' *)

  simpl.
  rewrite inversa_conc.
  (* inversa (inversa xs' ++ [x]) = x :: xs' *)
  (* inversa [x] ++ inversa (inversa xs') =
     x :: xs' *)
  rewrite HI.
  (* inversa [x] ++ xs' = x :: xs' *)
  reflexivity.

```

**Qed.**

```

(* =====
   §§ 1.2. Polimorfismo de pares
   ===== *)

(* -----
   Ejemplo 1.2.1. Definir el tipo prod (X Y) con el constructor par tal
   que (par x y) es el par cuyas componentes son x e y.
   ----- *)

```

**Inductive** prod (X Y : Type) : Type :=  
| par : X -> Y -> prod X Y.

**Arguments** par {X} {Y} \_ \_.

```
(* -----
  Ejemplo 1.2.2. Definir la abreviaturas
    "( x , y )" para (par x y).
  ----- *)
```

**Notation** "( x , y )" := (par x y).

```
(* -----
  Ejemplo 1.2.3. Definir la abreviatura
    "X * Y" para (prod X Y)
  ----- *)
```

**Notation** "X \* Y" := (prod X Y) : type\_scope.

```
(* -----
  Ejemplo 1.2.4. Definir la función
    fst {X Y : Type} (p : X * Y) : X
  tal que (fst p) es la primera componente del par p. Por ejemplo,
    fst (par 3 5) = 3
  ----- *)
```

**Definition** fst {X Y : **Type**} (p : X \* Y) : X :=  
**match** p **with**  
 | (x, y) => x  
**end.**

Compute (fst (par 3 5)).  
 (\* = 3 : nat\*)

**Example** prop\_fst: fst (par 3 5) = 3.

**Proof.** reflexivity. Qed.

```
(* -----
  Ejemplo 2.2.5. Definir la función
    snd {X Y : Type} (p : X * Y)
  tal que (snd p) es la segunda componente del par p. Por ejemplo,
    snd (par 3 5) = 5
  ----- *)
```

**Definition** snd {X Y : **Type**} (p : X \* Y) : Y :=

```

match p with
| (x, y) => y
end.

```

```

Compute (snd (par 3 5)).
(* = 5 : nat*)

```

**Example** prop\_snd: snd (par 3 5) = 5.

**Proof.** reflexivity. **Qed.**

```

(* -----
  Ejercicio 2.2.1. Definir la función
    empareja {X Y : Type} (xs : list X) (ys : list Y) : list (X*Y)
    tal que (empareja xs ys) es la lista obtenida emparejando los
    elementos de xs y ys. Por ejemplo,
    empareja [2;6] [3;5;7]      = [(2, 3); (6, 5)].
    empareja [2;6;4;8] [3;5;7] = [(2, 3); (6, 5); (4, 7)].
  ----- *)

```

```

Fixpoint empareja {X Y : Type} (xs : list X) (ys : list Y) : list (X*Y) :=
match xs, ys with
| []      , _      => []
| _      , []      => []
| x :: tx, y :: ty => (x, y) :: (empareja tx ty)
end.

```

```

Compute (empareja [2;6] [3;5;7]).
(* = [(2, 3); (6, 5)] : list (nat * nat)*)
Compute (empareja [2;6;4;8] [3;5;7]).
(* = [(2, 3); (6, 5); (4, 7)] : list (nat * nat)*)

```

**Example** prop\_combina1: empareja [2;6] [3;5;7] = [(2, 3); (6, 5)].

**Proof.** reflexivity. **Qed.**

**Example** prop\_combina2: empareja [2;6;4;8] [3;5;7] = [(2,3);(6, 5);(4,7)].

**Proof.** reflexivity. **Qed.**

```

(* -----
  Ejercicio 2.2.2. Evaluar la expresión
    Check @empareja

```

```
----- *)
```

**Check** @empareja.

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

```
(* -----
Ejercicio 2.2.3. Definir la función
  desempareja {X Y : Type} (ps : list (X*Y)) : (list X) * (list Y)
tal que (desempareja ps) es el par de listas (xs,ys) cuyo
emparejamiento es l. Por ejemplo,
  desempareja [(1,false);(2,false)] = ([1;2],[false;false]).
----- *)
```

```
Fixpoint desempareja {X Y : Type} (ps : list (X*Y)) : (list X) * (list Y) :=
match ps with
| [] => ([], [])
| (x, y) :: ps' =>
  match desempareja ps' with
  | (xs, ys) => (x :: xs, y :: ys)
  end
end.
```

```
Compute (desempareja [(2, 3); (6, 5)]).
```

```
(* = ([2; 6], [3; 5]) : list nat * list nat*)
```

**Example** prop\_desempareja:

```
desempareja [(2, 3); (6, 5)] = ([2; 6], [3; 5]).
```

**Proof.** reflexivity. **Qed.**

```
(* =====
§§ 1.3. Resultados opcionales polimórficos
===== *)
```

```
(* -----
Ejemplo 1.3.1. Definir el tipo (Opcional X) con los constructores Some
y None tales que
+ (Some x) es un valor de tipo X.
+ None es el valor nulo.
----- *)
```



**Inductive** Opcional (X:Type) : Type :=

| Some : X -> Opcional X  
| None : Opcional X.

**Arguments** Some {X} \_.

**Arguments** None {X}.

```
(* -----
Ejercicio 1.3.1. Definir la función
  nthOpcional {X : Type} (xs : list X) (n : nat) : Opcional X :=
tal que (nthOpcional xs n) es el n-ésimo elemento de xs o None
si la lista tiene menos de n elementos. Por ejemplo,
  nthOpcional [4;5;6;7] 0 = Some 4.
  nthOpcional [[1];[2]] 1 = Some [2].
  nthOpcional [true] 2    = None.
----- *)
```

```
Fixpoint nthOpcional {X : Type} (xs : list X) (n : nat) : Opcional X :=
match xs with
| []      => None
| x :: xs' => if iguales_nat n 0
               then Some x
               else nthOpcional xs' (pred n)
end.
```

**Example** prop\_nthOpcional1 : nthOpcional [4;5;6;7] 0 = Some 4.

**Proof.** reflexivity. Qed.

**Example** prop\_nthOpcional2 : nthOpcional [[1];[2]] 1 = Some [2].

**Proof.** reflexivity. Qed.

**Example** prop\_nthOpcional3 : nthOpcional [true] 2 = None.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 1.3.2. Definir la función
  primeroOpcional {X : Type} (xs : list X) : Opcional X
tal que (primeroOpcional xs) es el primer elemento de xs, si xs es no
vacía; o es None, en caso contrario. Por ejemplo,
  primeroOpcional [1;2]    = Some 1.
----- *)
```

```
    primeroOpcional [[1];[2]] = Some [1].
```

```
----- *)
```

**Definition** primeroOpcional {X : Type} (xs : list X) : Opcional X :=  
 match xs with  
 | [] => None  
 | x :: \_ => Some x  
**end.**

**Check** @primeroOpcional.

**Example** prop\_primeroOpcional1 : primeroOpcional [1;2] = Some 1.

**Proof.** reflexivity. **Qed.**

**Example** prop\_primeroOpcional2 : primeroOpcional [[1];[2]] = Some [1].

**Proof.** reflexivity. **Qed.**

```
(* =====  
  $ 2. Funciones como datos  
===== *)
```

```
(* =====  
  §§ 2.1. Funciones de orden superior  
===== *)
```

```
(* -----  
  Ejemplo 2.1.1. Definir la función  
    aplica3veces {X : Type} (f : X -> X) (n : X) : X  
  tal que (aplica3veces f) aplica 3 veces la función f. Por ejemplo,  
    aplica3veces menosDos 9      = 3.  
    aplica3veces negacion true   = false.  
----- *)
```

**Definition** aplica3veces {X : Type} (f : X -> X) (n : X) : X :=  
 f (f (f n)).

**Check** @aplica3veces.

```
(* ==> aplica3veces : forall X : Type, (X -> X) -> X -> X *)
```

**Example** prop\_aplica3veces: aplica3veces menosDos 9 = 3.

**Proof. reflexivity. Qed.**

**Example** prop\_aplica3veces': aplica3veces negacion true = false.

**Proof. reflexivity. Qed.**

```
(* =====
   §§ 2.2. Filtrado
   ===== *)
```

```
(* -----
   Ejemplo 2.2.1. Definir la función
       filtra {X : Type} (p : X -> bool) (xs : list X) : (list X)
   tal que (filtra p xs) es la lista de los elementos de xs que
   verifican p. Por ejemplo,
       filtra esPar [1;2;3;4] = [2;4].
   ----- *)
```

```
Fixpoint filtra {X : Type} (p : X -> bool) (xs : list X) : (list X) :=
match xs with
| []      => []
| x :: xs' => if p x
               then x :: (filtra p xs')
               else filtra p xs'
end.
```

**Example** prop\_filtra1: filtra esPar [1;2;3;4] = [2;4].

**Proof. reflexivity. Qed.**

```
(* -----
   Ejemplo 2.2.2. Definir la función
       unitarias {X : Type} (xss : list (list X)) : list (list X) :=
   tal que (unitarias xss) es la lista de listas unitarias de xss. Por
   ejemplo,
       unitarias [[1;2];[3];[4];[5;6;7];[];[8]] = [[3];[4];[8]]
   ----- *)
```

**Definition** esUnitaria {X : **Type**} (xs : **list** X) : **bool** :=  
iguales\_nat (longitud xs) 1.

**Definition** unitarias {X : **Type**} (xss : **list** (**list** X)) : **list** (**list** X) :=

filtra esUnitaria xss.

Compute (unitarias [[1; 2]; [3]; [4]; [5;6;7]; []; [8]]).  
 (\* = [[3]; [4]; [8]] : list (list nat)\*)

**Example** prop\_unitarias:

unitarias [[1; 2]; [3]; [4]; [5;6;7]; []; [8]]  
 = [[3]; [4]; [8]].

**Proof. reflexivity. Qed.**

```
(* -----
Ejercicio 2.2.3. Definir la función
  nImpares (xs : list nat) : nat
tal que nImpares xs) es el número de elementos impares de xs. Por
ejemplo,
  nImpares [1;0;3;1;4;5] = 4.
  nImpares [0;2;4]       = 0.
  nImpares nil           = 0.
----- *)
```

**Definition** nImpares (xs : list nat) : nat :=  
 longitud (filtra esImpar xs).

**Example** prop\_nImpares1: nImpares [1;0;3;1;4;5] = 4.

**Proof. reflexivity. Qed.**

**Example** prop\_nImpares2: nImpares [0;2;4] = 0.

**Proof. reflexivity. Qed.**

**Example** prop\_nImpares3: nImpares nil = 0.

**Proof. reflexivity. Qed.**

```
(* =====
§§ 2.3. Funciones anónimas
===== *)
```

```
(* -----
Ejemplo 2.3.1. Demostrar que
  aplica3veces (fun n => n * n) 2 = 256.
----- *)
```

**Example** prop\_anon\_fun' :

aplica3veces (fun n => n \* n) 2 = 256.

**Proof.** reflexivity. Qed.

```
(* -----
Ejemplo 2.3.2. Calcular
  filtra (fun xs => iguales_nat (longitud xs) 1)
    [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
----- *)
```

Compute (filtra (fun xs => iguales\_nat (longitud xs) 1)  
[ [1; 2]; [3]; [4]; [5;6;7]; [], [8] ]).

(\* = [[3]; [4]; [8]] : list (list nat)\*)

```
(* -----
Ejercicio 2.3.3. Definir la función
  filtra_pares_mayores7 (xs : list nat) : list nat
tal que (filtra_pares_mayores7 xs) es la lista de los elementos de xs
que son pares y mayores que 7. Por ejemplo,
  filtra_pares_mayores7 [1;2;6;9;10;3;12;8] = [10;12;8].
  filtra_pares_mayores7 [5;2;6;19;129]      = [].
----- *)
```

**Definition** filtra\_pares\_mayores7 (xs : list nat) : list nat :=  
filtra (fun x => esPar x && menor\_o\_igual 7 x) xs.

**Example** prop\_filtra\_pares\_mayores7\_1 :

filtra\_pares\_mayores7 [1;2;6;9;10;3;12;8] = [10;12;8].

**Proof.** reflexivity. Qed.

**Example** prop\_filtra\_pares\_mayores7\_2 :

filtra\_pares\_mayores7 [5;2;6;19;129] = [].

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 2.3.4. Definir la función
  partition {X : Type} (p : X -> bool) (xs : list X) : list X * list X
tal que (partition p xs) es el par de listas (ys,zs) donde xs es la
lista de los elementos de xs que cumplen p y zs la de las que no lo
```

```

    cumplen. Por ejemplo,
      partition esImpar [1;2;3;4;5]          = ([1;3;5], [2;4]).
      partition (fun x => false) [5;9;0] = ([], [5;9;0]).
    ----- *)

```

```

Definition partition {X : Type}
  (p : X -> bool)
  (xs : list X)
  : list X * list X :=
  (filtra p xs, filtra (fun x => negacion (p x)) xs).

```

**Example** prop\_partition1: partition esImpar [1;2;3;4;5] = ([1;3;5], [2;4]).  
**Proof.** reflexivity. **Qed.**

**Example** prop\_partition2: partition (fun x => false) [5;9;0] = ([], [5;9;0]).  
**Proof.** reflexivity. **Qed.**

```

(* =====
   §§ 2.4. Aplicación a todos los elementos (map)
   ===== *)

```

```

(* -----
   Ejercicio 2.4.1. Definir la función
     map {X Y:Type} (f : X -> Y) (xs:list X) : list Y
   tal que (map f xs) es la lista obtenida aplicando f a todos los
   elementos de xs. Por ejemplo,
     map (fun x => plus 3 x) [2;0;2] = [5;3;5].
     map esImpar [2;1;2;5] = [false;true;false;true].
     map (fun n => [evenb n;esImpar n]) [2;1;2;5]
       = [[true;false];[false;true];[true;false];[false;true]].
   ----- *)

```

```

Fixpoint map {X Y:Type} (f : X -> Y) (xs : list X) : list Y :=
match xs with
| []      => []
| x :: xs' => f x :: map f xs'
end.

```

**Example** prop\_map1:  
 map (fun x => plus 3 x) [2;0;2] = [5;3;5].

**Proof. reflexivity. Qed.**

**Example** prop\_map2:

```
map esImpar [2;1;2;5] = [false;true;false;true].
```

**Proof. reflexivity. Qed.**

**Example** prop\_map3:

```
map (fun n => [esPar n ; esImpar n]) [2;1;2;5]
= [[true;false];[false;true];[true;false];[false;true]].
```

**Proof. reflexivity. Qed.**

```
(* -----
   Ejercicio 2.4.2. Demostrar que
       map f (inversa l) = inversa (map f l).
   ----- *)
```

**Lemma** map\_conc: **forall** (X Y : **Type**) (f : X -> Y) (xs ys : **list** X),  
map f (xs ++ ys) = map f xs ++ map f ys.

**Proof.**

```
intros X Y f xs ys.
```

```
(* X : Type
   Y : Type
   f : X -> Y
   xs, ys : list X
   =====
   map f (xs ++ ys) = map f xs ++ map f ys *)
```

```
induction xs as [|x xs' HI].
```

```
+
```

```
(* X : Type
   Y : Type
   f : X -> Y
   ys : list X
   =====
   map f ([ ] ++ ys) = map f [ ] ++ map f ys *)
```

**reflexivity.**

```
+
```

```
(* X : Type
   Y : Type
   f : X -> Y
   x : X
   xs', ys : list X
   HI : map f (xs' ++ ys) =
       map f xs' ++ map f ys
```

```

=====
map f ((x :: xs') ++ ys) =
map f (x :: xs') ++ map f ys *)
simpl.
(* f x :: map f (xs' ++ ys) =
   f x :: (map f xs' ++ map f ys) *)
rewrite HI.
(* f x :: (map f xs' ++ map f ys) =
   f x :: (map f xs' ++ map f ys) *)
reflexivity.
Qed.

Theorem map_inversa : forall (X Y : Type) (f : X -> Y) (xs : list X),
  map f (inversa xs) = inversa (map f xs).
Proof.
intros X Y f xs.
(* X : Type
   Y : Type
   f : X -> Y
   xs : list X
   =====
   map f (inversa xs) = inversa (map f xs) *)
induction xs as [|x xs' HI].
+
(* X : Type
   Y : Type
   f : X -> Y
   =====
   map f (inversa [ ]) = inversa (map f [ ]) *)
reflexivity.
+
(* X : Type
   Y : Type
   f : X -> Y
   x : X
   xs' : list X
   HI : map f (inversa xs') = inversa (map f xs')
   =====
   map f (inversa (x :: xs')) =
   inversa (map f (x :: xs')) *)
simpl.
(* map f (inversa xs' ++ [x]) =
   inversa (map f xs') ++ [f x] *)
rewrite map_conc.
(* map f (inversa xs') ++ map f [x] =
   inversa (map f xs') ++ [f x] *)
rewrite HI.
(* inversa (map f xs') ++ map f [x] =

```



*inversa (map f xs') ++ [f x] \*)*

**reflexivity.**

**Qed.**

```
(* -----
Ejercicio 2.4.3. Definir la función
  conc_map {X Y : Type} (f : X -> list Y) (xs : list X) : (list Y)
tal que (conc_map f xs) es la concatenación de las listas obtenidas
aplicando f a l. Por ejemplo,
  conc_map (fun n => [n;n;n]) [1;5;4] = [1; 1; 1; 5; 5; 5; 4; 4; 4].
----- *)
```

```
Fixpoint conc_map {X Y : Type} (f : X -> list Y) (xs : list X) : (list Y) :=
match xs with
| []      => []
| x :: xs' => f x ++ conc_map f xs'
end.
```

**Example** prop\_conc\_map:

```
conc_map (fun n => [n;n;n]) [1;5;4]
= [1; 1; 1; 5; 5; 5; 4; 4; 4].
```

**Proof. reflexivity. Qed.**

```
(* -----
Ejercicio 2.4.4. Definir la función
  map_opcional {X Y : Type} (f : X -> Y) (o : Opcional X) : Opcional Y
tal que (map_opcional f o) es la aplicación de f a o. Por ejemplo,
  map_opcional S (Some 3) = Some 4
  map_opcional S None     = None
----- *)
```

```
Definition map_opcional {X Y : Type} (f : X -> Y) (o : Opcional X)
      : Opcional Y :=
```

```
match o with
| None    => None
| Some x  => Some (f x)
end.
```

Compute (map\_opcional S (Some 3)).

```
(* = Some 4 : Opcional nat*)
```

```
Compute (map_opcional S None).
(* = None : Opcional nat*)
```

**Example** prop\_map\_opcional1: map\_opcional S (Some 3) = Some 4.

**Proof.** reflexivity. Qed.

**Example** prop\_map\_opcional2: map\_opcional S None = None.

**Proof.** reflexivity. Qed.

```
(* =====
   §§ 2.5. Plegados (fold)
   ===== *)

(* -----
   Ejemplo 2.5.1. Definir la función
       fold {X Y:Type} (f: X -> Y -> Y) (xs : list X) (b : Y) : Y
   tal que (fold f xs b) es el plegado de xs con la operación f a partir
   del elemento b. Por ejemplo,
       fold mult [1;2;3;4] 1                = 24.
       fold conjuncion [true;true;false>true] true = false.
       fold conc  [[1];[];[2;3];[4]] []      = [1;2;3;4].
   ----- *)
```

```
Fixpoint fold {X Y:Type} (f: X -> Y -> Y) (xs : list X) (b : Y) : Y :=
  match xs with
  | nil    => b
  | x :: xs' => f x (fold f xs' b)
end.
```

**Check** (fold conjuncion).

```
(* ==> fold conjuncion : list bool -> bool -> bool *)
```

**Example** fold\_example1:

```
fold mult [1;2;3;4] 1 = 24.
```

**Proof.** reflexivity. Qed.

**Example** fold\_example2 :

```
fold conjuncion [true;true;false>true] true = false.
```

**Proof.** reflexivity. Qed.

**Example** fold\_example3 :  
**fold** conc [[1];[];[2;3];[4]] [] = [1;2;3;4].  
**Proof.** **reflexivity.** **Qed.**

```
(* =====
  §§ 2.6. Funciones que construyen funciones
  ===== *)
```

```
(* -----
  Ejemplo 2.6.1. Definir la función
    constante {X : Type} (x : X) : nat -> X
  tal que (constante x) es la función que a todos los naturales le
  asigna el x. Por ejemplo,
    (constante 5) 99 = 5.
  ----- *)
```

**Definition** constante {X : **Type**} (x : X) : **nat** -> X :=  
**fun** (k : **nat**) => x.

**Example** prop\_constante: (constante 5) 99 = 5.  
**Proof.** **reflexivity.** **Qed.**

```
(* -----
  Ejemplo 2.6.2. Calcular el tipo de plus.
  ----- *)
```

**Check** plus.

```
(* ==> nat -> nat -> nat *)
```

```
(* -----
  Ejemplo 2.6.3. Definir la función
    plus3 : nat -> nat
  tal que (plus3 x) es tres más x. Por ejemplo,
    plus3 4 = 7.
    aplica3veces plus3 0 = 9.
    aplica3veces (plus 3) 0 = 9.
  ----- *)
```

**Definition** plus3 := plus 3.

**Example** prop\_plus3a: plus3 4 = 7.

**Proof.** reflexivity. Qed.

**Example** prop\_plus3b: aplica3veces plus3 0 = 9.

**Proof.** reflexivity. Qed.

**Example** prop\_plus3c: aplica3veces (plus 3) 0 = 9.

**Proof.** reflexivity. Qed.

```
(* =====
  § 3. Ejercicios
  ===== *)
```

**Module** Exercises.

```
(* -----
  Ejercicio 3.1. Definir, usando fold, la función
    longitudF {X : Type} (xs : list X) : nat
  tal que (longitudF xs) es la longitud de xs. Por ejemplo,
    longitudF [4;7;0] = 3.
  ----- *)
```

**Definition** longitudF {X : **Type**} (xs : **list** X) : **nat** :=  
 fold (fun \_ n => S n) xs 0.

**Example** prop\_longitudF1: longitudF [4;7;0] = 3.

**Proof.** reflexivity. Qed.

```
(* -----
  Ejemplo 3.2. Demostrar que
    longitudF l = longitud l.
  ----- *)
```

**Theorem** longitudF\_longitud: forall X (xs : **list** X),  
 longitudF xs = longitud xs.

**Proof.**

intros X xs.

```
(* X : Type
  xs : list X
  =====
  longitudF xs = longitud xs *)
```

```

unfold longitudF.          (* fold (fun (_ : X) (n : nat) => S n) xs 0 =
                             longitud xs *)

induction xs as [|x xs' HI|.
-
                             (* X : Type
                             =====
                             fold (fun (_ : X) (n : nat) => S n) [ ] 0 =
                             longitud [ ] *)

reflexivity.
-
                             (* X : Type
                             x : X
                             xs' : list X
                             HI : fold (fun (_:X) (n:nat) => S n) xs' 0 =
                             longitud xs'
                             =====
                             fold (fun (_:X) (n:nat) => S n) (x::xs') 0 =
                             longitud (x :: xs') *)

simpl.                    (* S (fold (fun (_:X) (n:nat) => S n) xs' 0) =
                             S (longitud xs') *)

rewrite HI.
reflexivity.
Qed.

(* -----
   Ejercicio 3.2. Definir, usando fold, la función
       mapF {X Y : Type} (f : X -> Y) (xs : list X) : list Y
   tal que (mapF f xs) es la lista obtenida aplicando f a los
   elementos de l.
   ----- *)

Definition mapF {X Y : Type} (f : X -> Y) (xs : list X) : list Y :=
  fold (fun x t => (f x) :: t) xs [].

(* -----
   Ejercicio 3.4. Demostrar que mapF es equivalente a map.
   ----- *)

Theorem mapF_correct : forall (X Y : Type) (f : X -> Y) (xs : list X),
  mapF f xs = map f xs.

Proof.
  intros X Y f xs.          (* X : Type

```

```

Y : Type
f : X -> Y
xs : list X
=====
mapF f xs = map f xs *)
unfold mapF.
(* fold (fun (x:X) (t:list Y) => f x::t) xs []
   = map f xs *)

induction xs as [|x xs' HI|.
-
(* X : Type
   Y : Type
   f : X -> Y
   =====
   fold (fun (x:X) (t:list Y) => f x :: t) [] []
   = map f [ ] *)

reflexivity.
-
(* X : Type
   Y : Type
   f : X -> Y
   x : X
   xs' : list X
   HI : fold (fun (x:X) (t:list Y) => f x :: t)
            xs' [ ]
        = map f xs'
   =====
   fold (fun (x0:X) (t:list Y) => f x0 :: t)
        (x :: xs') [ ]
   = map f (x :: xs') *)
simpl.
(* f x :: fold (fun (x0:X) (t:list Y) =>
                f x0 :: t) xs' [ ]
   = f x :: map f xs' *)
rewrite HI.
reflexivity.
Qed.

(* -----
Ejemplo 3.5. Definir la función
  curry {X Y Z : Type} (f : X * Y -> Z) (x : X) (y : Y) : Z
tal que (curry f x y) es la versión curryficada de f. Por ejemplo,
  curry fst 3 5 = 3.
----- *)

```

**Definition** curry {X Y Z : Type}

(f : X \* Y -> Z) (x : X) (y : Y) : Z := f (x, y).

Compute (curry fst 3 5).

(\* = 3 : nat\*)

**Example** prop\_curry: curry fst 3 5 = 3.

**Proof.** reflexivity. Qed.

```
(* -----
  Ejercicio 3.6. Definir la función
    uncurry {X Y Z : Type} (f : X -> Y -> Z) (p : X * Y) : Z
  tal que (uncurry f p) es la versión incurryficada de f.
  ----- *)
```

**Definition** uncurry {X Y Z : Type}

(f : X -> Y -> Z) (p : X \* Y) : Z := f (fst p) (snd p).

Compute (uncurry mult (2,5)).

(\* = 10 : nat\*)

**Example** prop\_uncurry: uncurry mult (2,5) = 10.

**Proof.** reflexivity. Qed.

```
(* -----
  Ejercicio 3.7. Calcular el tipo de las funciones curry y uncurry.
  ----- *)
```

**Check** @curry.

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

**Check** @uncurry.

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

```
(* -----
  Ejercicio 3.8. Demostrar que
    curry (uncurry f) x y = f x y
  ----- *)
```

```
Theorem uncurry_curry : forall (X Y Z : Type)
  (f : X -> Y -> Z)
  x y,
  curry (uncurry f) x y = f x y.
```

**Proof.** **reflexivity.** **Qed.**

```
(* -----
  Ejercicio 3.9. Demostrar que
    uncurry (curry f) p = f p.
  ----- *)
```

```
Theorem curry_uncurry : forall (X Y Z : Type)
  (f : (X * Y) -> Z)
  (p : X * Y),
  uncurry (curry f) p = f p.
```

**Proof.**

```
intros.      (* X : Type
               Y : Type
               Z : Type
               f : X * Y -> Z
               p : X * Y
               =====
               uncurry (curry f) p = f p *)
destruct p.  (* uncurry (curry f) (x, y) = f (x, y) *)
reflexivity.
```

**Qed.**

**Module** Church.

```
(* -----
  Ejercicio 3.11.1. En los siguientes ejercicios se trabajará con la
  definición de Church de los números naturales: el número natural n es
  la función que toma como argumento una función f y devuelve como
  valor la aplicación de n veces la función f.

  Definir el tipo nat para los números naturales de Church.
  ----- *)
```

**Definition** nat := **forall** X : **Type**, (X -> X) -> X -> X.



```
(* -----
  Ejercicio 3.11.2. Definir la función
    uno : nat
  tal que uno es el número uno de Church.
  ----- *)
```

```
Definition uno : nat :=
fun (X : Type) (f : X -> X) (x : X) => f x.
```

```
(* -----
  Ejercicio 3.11.3. Definir la función
    dos : nat
  tal que dos es el número dos de Church.
  ----- *)
```

```
Definition dos : nat :=
fun (X : Type) (f : X -> X) (x : X) => f (f x).
```

```
(* -----
  Ejercicio 3.11.4. Definir la función
    cero : nat
  tal que cero es el número cero de Church.
  ----- *)
```

```
Definition cero : nat :=
fun (X : Type) (f : X -> X) (x : X) => x.
```

```
(* -----
  Ejercicio 3.11.5. Definir la función
    tres : nat
  tal que tres es el número tres de Church.
  ----- *)
```

```
Definition tres : nat := @aplica3veces.
```

```
(* -----
  Ejercicio 3.11.5. Definir la función
    suc (n : nat) : nat
  tal que (suc n) es el siguiente del número n de Church. Por ejemplo,
    suc cero = uno.
```

```
suc uno  = dos.
suc dos  = tres.
```

```
----- *)
```

**Definition** suc (n : nat) : nat :=  
 fun (X : Type) (f : X -> X) (x : X) => f (n X f x).

**Example** prop\_suc\_1: suc cero = uno.

**Proof.** reflexivity. Qed.

**Example** prop\_suc\_2: suc uno = dos.

**Proof.** reflexivity. Qed.

**Example** prop\_suc\_3: suc dos = tres.

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 3.11.6. Definir la función
  suma (n m : nat) : nat
tal que (suma n m) es la suma de n y m. Por ejemplo,
  suma cero uno          = uno.
  suma dos tres          = suma tres dos.
  suma (suma dos dos) tres = suma uno (suma tres tres).
----- *)
```

**Definition** suma (n m : nat) : nat :=  
 fun (X : Type) (f : X -> X) (x : X) => m X f (n X f x).

**Example** prop\_suma\_1 : suma cero uno = uno.

**Proof.** reflexivity. Qed.

**Example** prop\_suma\_2 : suma dos tres = suma tres dos.

**Proof.** reflexivity. Qed.

**Example** prop\_suma\_3 :

suma (suma dos dos) tres = suma uno (suma tres tres).

**Proof.** reflexivity. Qed.

```
(* -----
Ejercicio 3.11.7. Definir la función
```

```

    producto (n m : nat) : nat
    tal que (producto n m) es el producto de n y m. Por ejemplo,
    producto uno uno = uno.
    producto cero (suma tres tres) = cero.
    producto dos tres = suma tres tres.
    ----- *)

```

**Definition** producto (n m : nat) : nat :=  
 fun (X : Type) (f : X -> X) (x : X) => n X (m X f) x.

**Example** prop\_producto\_1: producto uno uno = uno.

**Proof.** reflexivity. Qed.

**Example** prop\_producto\_2: producto cero (suma tres tres) = cero.

**Proof.** reflexivity. Qed.

**Example** prop\_producto\_3: producto dos tres = suma tres tres.

**Proof.** reflexivity. Qed.

```

(* -----
    Ejercicio 3.11.8. Definir la función
    potencia (n m : nat) : nat
    tal que (potencia n m) es la potencia m-ésima de n. Por ejemplo,
    potencia dos dos = suma dos dos.
    potencia tres dos = suma (producto dos (producto dos dos)) uno.
    potencia tres cero = uno.
    ----- *)

```

**Definition** potencia (n m : nat) : nat :=  
 ( fun (X : Type) (f : X -> X) (x : X) => (m (X -> X) (n X) f) x ).

**Example** prop\_potencia\_1: potencia dos dos = suma dos dos.

**Proof.** reflexivity. Qed.

**Example** prop\_potencia\_2:

potencia tres dos = suma (producto dos (producto dos dos)) uno.

**Proof.** reflexivity. Qed.

**Example** prop\_potencia\_3: potencia tres cero = uno.

**Proof.** reflexivity. Qed.

**End** Church.

**End** Exercises.

```
(* =====  
  § Bibliografía  
  ===== *)
```

```
(*  
+ "Polymorphism and higher-order functions" de Peirce et als.  
  http://bit.ly/2Mj5gMf *)
```

# Tema 5

## Tácticas básicas de Coq

```
(* T5: Tácticas básicas de Coq *)
```

```
Set Warnings "-notation-overridden,-parsing".
```

```
Require Export T4_PolimorfismoyOS.
```

```
(* El contenido del tema es
```

```
1. La táctica 'apply'
```

```
2. La táctica 'apply ... with ...'
```

```
3. La táctica 'inversion'
```

```
4. Uso de tácticas sobre las hipótesis
```

```
5. Control de la hipótesis de inducción
```

```
6. Expansión de definiciones
```

```
7. Uso de 'destruct' sobre expresiones compuestas
```

```
8. Ejercicios
```

```
9. Resumen de tácticas básicas
```

```
*)
```

```
(* =====  
§ 1. La táctica 'apply'  
===== *)
```

```
(* -----  
Ejemplo 1.1. Demostrar que  
     $n = m \rightarrow$   
     $[n;o] = [n;p] \rightarrow$   
     $[n;o] = [m;p]$ .  
----- *)
```

```

(* Demostración sin apply *)
Theorem artificial_1a : forall (n m o p : nat),
  n = m ->
  [n;o] = [n;p] ->
  [n;o] = [m;p].
Proof.
  intros n m o p H1 H2. (* n, m, o, p : nat
                        H1 : n = m
                        H2 : [n; o] = [n; p]
                        =====
                        [n; o] = [m; p] *)
  rewrite <- H1.          (* [n; o] = [n; p] *)
  rewrite H2.             (* [n; p] = [n; p] *)
  reflexivity.
Qed.

```

```

(* Demostración con apply *)
Theorem artificial_1b : forall (n m o p : nat),
  n = m ->
  [n;o] = [n;p] ->
  [n;o] = [m;p].
Proof.
  intros n m o p H1 H2. (* n, m, o, p : nat
                        H1 : n = m
                        H2 : [n; o] = [n; p]
                        =====
                        [n; o] = [m; p] *)
  rewrite <- H1.          (* [n; o] = [n; p] *)
  apply H2.
Qed.

```

```

(* -----
   Nota. Uso de la táctica 'apply'.
   ----- *)

```

```

(* -----
   Ejemplo 1.2. Demostrar que
     n = m ->
     (forall (q r : nat), q = r -> [q;o] = [r;p]) ->
     [n;o] = [m;p].

```

```

----- *)

Theorem artificial2 : forall (n m o p : nat),
  n = m ->
  (forall (q r : nat), q = r -> [q;o] = [r;p]) ->
  [n;o] = [m;p].
Proof.
  intros n m o p H1 H2. (* n, m, o, p : nat
                           H1 : n = m
                           H2 : forall q r : nat, q = r -> [q; o] = [r; p]
                           =====
                           [n; o] = [m; p] *)

  apply H2.              (* n = m *)
  apply H1.

Qed.

(* -----
   Nota. Uso de la táctica 'apply' en hipótesis condicionales y
   razonamiento hacia atrás
   ----- *)

(* -----
   Ejemplo 1.3. Demostrar que
   (n,n) = (m,m) ->
   (forall (q r : nat), (q,q) = (r,r) -> [q] = [r]) ->
   [n] = [m].
   ----- *)

Theorem artificial2a : forall (n m : nat),
  (n,n) = (m,m) ->
  (forall (q r : nat), (q,q) = (r,r) -> [q] = [r]) ->
  [n] = [m].
Proof.
  intros n m H1 H2. (* n, m : nat
                       H1 : (n, n) = (m, m)
                       H2 : forall q r : nat, (q, q) = (r, r) -> [q] = [r]
                       =====
                       [n] = [m] *)

  apply H2.          (* (n, n) = (m, m) *)
  apply H1.

```

**Qed.**

```
(* -----
  Ejercicio 1.1. Demostrar, sin usar simpl, que
    (forall n, evenb n = true -> oddb (S n) = true) ->
    evenb 3 = true ->
    oddb 4 = true.
  ----- *)
```

**Theorem** artificial\_ex :

```
(forall n, esPar n = true -> esImpar (S n) = true) ->
esPar 3 = true ->
esImpar 4 = true.
```

**Proof.**

```
intros H1 H2. (* H1 : forall n : nat, esPar n = true -> esImpar (S n) = true
                H2 : esPar 3 = true
```

```
=====
                esImpar 4 = true *)
```

```
apply H1.      (* esPar 3 = true *)
```

```
apply H2.
```

**Qed.**

```
(* -----
  Ejemplo 1.4. Demostrar que
    true = iguales_nat n 5 ->
    iguales_nat (S (S n)) 7 = true.
  ----- *)
```

**Theorem** artificial3a: forall (n : nat),

```
true = iguales_nat n 5 ->
iguales_nat (S (S n)) 7 = true.
```

**Proof.**

```
intros n H. (* n : nat
              H : true = iguales_nat n 5
              =====
              iguales_nat (S (S n)) 7 = true *)
```

```
symmetry.      (* true = iguales_nat (S (S n)) 7 *)
```

```
simpl.        (* true = iguales_nat n 5 *)
```

```
apply H.
```

**Qed.**



```
(* -----
Nota. Necesidad de usar symmetry antes de apply.
----- *)
```

```
(* -----
Ejercicio 1.2. Demostrar
  forall (xs ys : list nat),
    xs = inversa ys -> ys = inversa xs.
----- *)
```

**Theorem** inversa2: **forall** (xs ys : **list nat**),  
 xs = inversa ys -> ys = inversa xs.

**Proof.**

```
intros xs ys H.          (* xs, ys : list nat
                           H : xs = inversa ys
                           =====
                           ys = inversa xs *)

rewrite H.                (* ys = inversa (inversa ys) *)
symmetry.                 (* inversa (inversa ys) = ys *)
apply inversa_involutiva.
```

**Qed.**

```
(* =====
§ 2. La táctica 'apply ... with ...'
===== *)
```

```
(* -----
Ejemplo 2.1. Demostrar que
  forall (a b c d e f : nat),
    [a;b] = [c;d] ->
    [c;d] = [e;f] ->
    [a;b] = [e;f].
----- *)
```

**Example** ejemplo\_con\_transitiva: **forall** (a b c d e f : **nat**),  
 [a;b] = [c;d] ->  
 [c;d] = [e;f] ->  
 [a;b] = [e;f].

**Proof.**

```

intros a b c d e f H1 H2. (* a, b, c, d, e, f : nat
                             H1 : [a; b] = [c; d]
                             H2 : [c; d] = [e; f]
                             =====
                             [a; b] = [e; f] *)
rewrite -> H1.             (* [c; d] = [e; f] *)
rewrite -> H2.             (* [e; f] = [e; f] *)
reflexivity.
Qed.

(* -----
   Ejemplo 2.2. Demostrar que
   forall (X : Type) (n m o : X),
     n = m -> m = o -> n = o.
   ----- *)

Theorem igualdad_transitiva: forall (X:Type) (n m o : X),
  n = m -> m = o -> n = o.
Proof.
  intros X n m o H1 H2. (* X : Type
                           n, m, o : X
                           H1 : n = m
                           H2 : m = o
                           =====
                           n = o *)
  rewrite -> H1.          (* m = o *)
  rewrite -> H2.          (* o = o *)
  reflexivity.
Qed.

(* -----
   Nota. El ejercicio 2.2 es una generalización del 2.1, sus
   demostraciones son isomorfas y se puede usar el 2.2 en la
   demostración del 2.1.
   ----- *)

(* -----
   Ejemplo 2.3. Demostrar que
   forall (X : Type) (n m o : X),
     n = m -> m = o -> n = o.
   ----- *)

```

```

----- *)

(* 1ª demostración *)
Example ejemplo_con_transitiva' : forall (a b c d e f : nat),
  [a;b] = [c;d] ->
  [c;d] = [e;f] ->
  [a;b] = [e;f].
Proof.
  intros a b c d e f H1 H2.

  (* a, b, c, d, e, f : nat
     H1 : [a; b] = [c; d]
     H2 : [c; d] = [e; f]
     =====
     [a; b] = [e; f] *)

  apply igualdad_transitiva with (m:=[c;d]).
  -
    apply H1.
  -
    apply H2.
Qed.

(* 2ª demostración *)
Example ejemplo_con_transitiva'' : forall (a b c d e f : nat),
  [a;b] = [c;d] ->
  [c;d] = [e;f] ->
  [a;b] = [e;f].
Proof.
  intros a b c d e f H1 H2.

  (* a, b, c, d, e, f : nat
     H1 : [a; b] = [c; d]
     H2 : [c; d] = [e; f]
     =====
     [a; b] = [e; f] *)

  apply igualdad_transitiva with [c;d].
  -
    apply H1.
  -
    apply H2.
Qed.

(* -----
   Nota. Uso de la táctica 'apply ... with ...'

```

```

----- *)
(* -----
  Ejercicio 2.1. Demostrar que
    forall (n m o p : nat),
      m = (menosDos o) ->
      (n + p) = m ->
      (n + p) = (menosDos o).
----- *)

```

**Example** ejercicio\_igualdad\_transitiva: **forall** (n m o p : **nat**),  
 m = (menosDos o) ->  
 (n + p) = m ->  
 (n + p) = (menosDos o).

**Proof.**

```

intros n m o p H1 H2.                (* n, m, o, p : nat
                                         H1 : m = menosDos o
                                         H2 : n + p = m
                                         =====
                                         n + p = menosDos o *)

apply igualdad_transitiva with m.
-                                     (* n + p = m *)
  apply H2.
-                                     (* m = menosDos o *)
  apply H1.

```

**Qed.**

```

(* =====
  § 3. La táctica 'inversion'
  ===== *)

```

```

(* -----
  Ejemplo 3.1. Demostrar que
    forall (n m : nat),
      S n = S m -> n = m.
----- *)

```

**Theorem** S\_inyectiva: **forall** (n m : **nat**),  
 S n = S m ->  
 n = m.

**Proof.**

```

intros n m H. (* n, m : nat
                  H : S n = S m
                  =====
                  n = m *)
inversion H. (* n, m : nat
                  H : S n = S m
                  H1 : n = m
                  =====
                  m = m *)

```

**reflexivity.**

**Qed.**

```

(* -----
   Nota. Uso de la táctica 'inversion'
   ----- *)

(* -----
   Ejemplo 3.2. Demostrar que
   forall (n m o : nat),
     [n; m] = [o; o] -> [n] = [m].
   ----- *)

```

**Theorem** inversion\_ej1: **forall** (n m o : **nat**),  
 [n; m] = [o; o] ->  
 [n] = [m].

**Proof.**

```

intros n m o H. (* n, m, o : nat
                  H : [n; m] = [o; o]
                  =====
                  [n] = [m] *)
inversion H. (* n, m, o : nat
                  H : [n; m] = [o; o]
                  H1 : n = o
                  H2 : m = o
                  =====
                  [o] = [o] *)

```

**reflexivity.**

**Qed.**

```
(* -----
  Ejemplo 3.3. Demostrar que
    forall (n m : nat),
      [n] = [m] ->
        n = m.
  ----- *)
```

**Theorem** inversion\_ej2: **forall** (n m : **nat**),  
 [n] = [m] ->  
 n = m.

**Proof.**

```
  intros n m H.          (* n, m : nat
                           H : [n] = [m]
                           =====
                           n = m *)
  inversion H as [Hnm].  (* n, m : nat
                           H : [n] = [m]
                           Hnm : n = m
                           =====
                           m = m *)
```

**reflexivity.**

**Qed.**

```
(* -----
  Nota. Nombramiento de las hipótesis generadas por inversión.
  ----- *)
```

```
(* -----
  Ejercicio 3.1. Demostrar que
    forall (X : Type) (x y z : X) (xs ys : list X),
      x :: y :: xs = z :: ys ->
        y :: xs = x :: ys ->
          x = y.
  ----- *)
```

**Example** inversion\_ej3 : **forall** (X : **Type**) (x y z : X) (xs ys : **list** X),  
 x :: y :: xs = z :: ys ->  
 y :: xs = x :: ys ->  
 x = y.

**Proof.**

```

intros X x y z xs ys H1 H2. (* X : Type
                                x, y, z : X
                                xs, ys : list X
                                H1 : x :: y :: xs = z :: ys
                                H2 : y :: xs = x :: ys
                                =====
                                x = y *)
inversion H1. (* X : Type
                                x, y, z : X
                                xs, ys : list X
                                H1 : x :: y :: xs = z :: ys
                                H2 : y :: xs = x :: ys
                                H0 : x = z
                                H3 : y :: xs = ys
                                =====
                                z = y *)
inversion H2. (* xs, ys : list X
                                H1 : x :: y :: xs = z :: ys
                                H2 : y :: xs = x :: ys
                                H0 : x = z
                                H3 : y :: xs = ys
                                H4 : y = x
                                H5 : xs = ys
                                =====
                                z = x *)
symmetry.
apply H0.
Qed.

(* -----
   Ejemplo 3.4. Demostrar que
   forall n:nat,
   iguales_nat 0 n = true -> n = 0.
   ----- *)

Theorem iguales_nat_0_n: forall n:nat,
  iguales_nat 0 n = true -> n = 0.
Proof.
  intros n. (* n : nat
               =====

```

```

                                iguales_nat 0 n = true -> n = 0 *)
destruct n as [| n'].
-
                                (*
                                =====
                                iguales_nat 0 0 = true -> 0 = 0 *)
                                (* H : iguales_nat 0 0 = true
                                =====
                                0 = 0 *)
                                reflexivity.
-
                                (* n' : nat
                                =====
                                iguales_nat 0 (S n') = true -> S n' = 0 *)
                                simpl.
                                (* n' : nat
                                =====
                                false = true -> S n' = 0 *)
                                intros H.
                                (* n' : nat
                                H : false = true
                                =====
                                S n' = 0 *)
                                inversion H.
Qed.

(* -----
Ejemplo 3.5. Demostrar que
  forall (n : nat),
    S n = 0 -> 2 + 2 = 5.
----- *)

Theorem inversion_ej4: forall (n : nat),
  S n = 0 ->
  2 + 2 = 5.
Proof.
  intros n H. (* n : nat
               H : S n = 0
               =====
               2 + 2 = 5 *)
  inversion H.
Qed.

(* -----
```



*Ejemplo 3.6. Demostrar que*

```
forall (n m : nat),
  false = true -> [n] = [m].
```

----- \*)

**Theorem** inversion\_ej5: **forall** (n m : **nat**),  
 false = true -> [n] = [m].

**Proof.**

```
intros n m H. (* n, m : nat
               H : false = true
               =====
               [n] = [m] *)
```

**inversion** H.

**Qed.**

(\* -----  
*Ejercicio 3.2. Demostrar que*  
 forall (X : Type) (x y z : X) (xs ys : list X),  
 x :: y :: xs = [] ->  
 y :: xs = z :: ys ->  
 x = z.  
 ----- \*)

**Example** inversion\_ej6 :

```
forall (X : Type) (x y z : X) (xs ys : list X),
  x :: y :: xs = [] ->
  y :: xs = z :: ys ->
  x = z.
```

**Proof.**

```
intros X x y z xs ys H. (* X : Type
                           x, y, z : X
                           xs, ys : list X
                           H : x :: y :: xs = [ ]
                           =====
                           y :: xs = z :: ys -> x = z *)
```

**inversion** H.

**Qed.**

(\* -----  
*Ejemplo 3.7. Demostrar que*

```
forall (A B : Type) (f: A -> B) (x y: A),
  x = y -> f x = f y.
```

```
----- *)
```

**Theorem** funcional: **forall** (A B : **Type**) (f: A -> B) (x y: A),  
 x = y -> f x = f y.

**Proof.**

```
intros A B f x y H. (* A : Type
                      B : Type
                      f : A -> B
                      x, y : A
                      H : x = y
                      =====
                      f x = f y *)
rewrite H.           (* f y = f y *)
reflexivity.
```

**Qed.**

```
(* =====
   § 4. Uso de tácticas sobre las hipótesis
   ===== *)
```

```
(* -----
   Ejemplo 4.1. Demostrar que
   forall (n m : nat) (b : bool),
     iguales_nat (S n) (S m) = b ->
     iguales_nat n m = b.
   ----- *)
```

**Theorem** S\_inj: **forall** (n m : **nat**) (b : **bool**),  
 iguales\_nat (S n) (S m) = b ->  
 iguales\_nat n m = b.

**Proof.**

```
intros n m b H. (* n, m : nat
                  b : bool
                  H : iguales_nat (S n) (S m) = b
                  =====
                  iguales_nat n m = b *)
simpl in H.      (* n, m : nat
                  b : bool
```

```

      H : iguales_nat n m = b
      =====
      iguales_nat n m = b *)

```

**apply** H.

**Qed.**

```

(* -----
   Nota. Uso de táctica 'simpl in ...'
   ----- *)

```

```

(* -----
   Ejemplo 4.1. Demostrar que
   forall (n : nat),
     (iguales_nat n 5 = true -> iguales_nat (S (S n)) 7 = true) ->
     true = iguales_nat n 5 ->
     true = iguales_nat (S (S n)) 7.
   ----- *)

```

**Theorem** artificial3': **forall** (n : **nat**),  
 (iguales\_nat n 5 = **true** -> iguales\_nat (S (S n)) 7 = **true**) ->  
**true** = iguales\_nat n 5 ->  
**true** = iguales\_nat (S (S n)) 7.

**Proof.**

```

intros n H1 H2. (* n : nat
                  H1 : iguales_nat n 5 = true ->
                      iguales_nat (S (S n)) 7 = true
                  H2 : true = iguales_nat n 5
                  =====
                  true = iguales_nat (S (S n)) 7 *)
symmetry in H2. (* n : nat
                  H1 : iguales_nat n 5 = true ->
                      iguales_nat (S (S n)) 7 = true
                  H2 : iguales_nat n 5 = true
                  =====
                  true = iguales_nat (S (S n)) 7 *)
apply H1 in H2. (* n : nat
                  H1 : iguales_nat n 5 = true ->
                      iguales_nat (S (S n)) 7 = true
                  H2 : iguales_nat (S (S n)) 7 = true
                  =====

```

```

      true = iguales_nat (S (S n)) 7 *)
symmetry in H2. (* n : nat
      H1 : iguales_nat n 5 = true ->
            iguales_nat (S (S n)) 7 = true
      H2 : true = iguales_nat (S (S n)) 7
      =====
      true = iguales_nat (S (S n)) 7 *)

apply H2.
Qed.

(* -----
   Nota. Uso de las tácticas 'apply H1 in H2' y 'symmetry in H'.
   ----- *)

(* -----
   Ejercicio 4.1. Demostrar
       forall n m : nat,
         n + n = m + m ->
           n = m.

   Nota: Usar suma_s_Sm.
   ----- *)

Theorem suma_n_n_inyectiva:
  forall n m : nat,
    n + n = m + m ->
      n = m.
Proof.
  intros n.
  induction n as [| n' HI].
  -
    intros m H1.
    destruct m.
    (* n : nat
       =====
       forall m : nat, n + n = m + m -> n = m *)

    (*
       =====
       forall m : nat, 0 + 0 = m + m -> 0 = m *)
    (* m : nat
       H1 : 0 + 0 = m + m
       =====
       0 = m *)

```

```

+
(* H1 : 0 + 0 = 0 + 0
=====
0 = 0 *)

reflexivity.

+
(* m : nat
H1 : 0 + 0 = S m + S m
=====
0 = S m *)

inversion H1.

-
(* n' : nat
HI : forall m : nat, n' + n' = m + m
-> n' = m
=====
forall m : nat, S n' + S n' = m + m
-> S n' = m *)

intros m H2.

(* n' : nat
HI : forall m : nat, n' + n' = m + m
-> n' = m

m : nat
H2 : S n' + S n' = m + m
=====
S n' = m *)

destruct m.

+
(* n' : nat
HI : forall m : nat, n' + n' = m + m
-> n' = m

H2 : S n' + S n' = 0 + 0
=====
S n' = 0 *)

inversion H2.

+
(* n' : nat
HI : forall m : nat, n' + n' = m + m
-> n' = m

m : nat
H2 : S n' + S n' = S m + S m
=====
S n' = S m *)

inversion H2.

```

```

m : nat
H2 : S n' + S n' = S m + S m
H0 : n' + S n' = m + S m
=====
S n' = S m *)
rewrite <- suma_n_Sm in H0. (* n' : nat
HI : forall m : nat, n' + n' = m + m
-> n' = m

m : nat
H2 : S n' + S n' = S m + S m
H0 : S (n' + n') = m + S m
=====
S n' = S m *)
symmetry in H0. (* n' : nat
HI : forall m : nat, n' + n' = m + m
-> n' = m

m : nat
H2 : S n' + S n' = S m + S m
H0 : m + S m = S (n' + n')
=====
S n' = S m *)
rewrite <- suma_n_Sm in H0. (* n' : nat
HI : forall m : nat, n' + n' = m + m
-> n' = m

m : nat
H2 : S n' + S n' = S m + S m
H0 : S (m + m) = S (n' + n')
=====
S n' = S m *)
inversion H0. (* n' : nat
HI : forall m : nat, n' + n' = m + m
-> n' = m

m : nat
H2 : S n' + S n' = S m + S m
H0 : S (m + m) = S (n' + n')
H1 : m + m = n' + n'
=====
S n' = S m *)
symmetry in H1. (* n' : nat
HI : forall m : nat, n' + n' = m + m

```

```

                                -> n' = m
m : nat
H2 : S n' + S n' = S m + S m
H0 : S (m + m) = S (n' + n')
H1 : n' + n' = m + m
=====
S n' = S m *)
apply HI in H1.
(* n' : nat
  HI : forall m : nat, n' + n' = m + m
      -> n' = m

  m : nat
  H2 : S n' + S n' = S m + S m
  H0 : S (m + m) = S (n' + n')
  H1 : n' = m
  =====
  S n' = S m *)
rewrite <- H1.
(* n' : nat
  HI : forall m : nat, n' + n' = m + m
      -> n' = m

  m : nat
  H2 : S n' + S n' = S m + S m
  H0 : S (m + m) = S (n' + n')
  H1 : n' = m
  =====
  S n' = S n' *)

reflexivity.
Qed.

(* =====
   § 5. Control de la hipótesis de inducción
   ===== *)

(* -----
   Ejemplo 5.1. Demostrar que
     forall n m : nat,
       doble n = doble m -> n = m.
   ----- *)

(* 1ª intento *)
Theorem doble_inyectiva_FAILED : forall n m : nat,
```

```

    doble n = doble m ->
    n = m.
Proof.
  intros n m.
    (* n, m : nat
    =====
    doble n = doble m -> n = m *)
  induction n as [| n' HI].
  -
    (* m : nat
    =====
    doble 0 = doble m -> 0 = m *)
  simpl.
    (* m : nat
    =====
    0 = doble m -> 0 = m *)
  intros H.
    (* m : nat
    H : 0 = doble m
    =====
    0 = m *)
  destruct m as [| m'].
  +
    (* H : 0 = doble 0
    =====
    0 = 0 *)
  reflexivity.
  +
    (* m' : nat
    H : 0 = doble (S m')
    =====
    0 = S m' *)
  inversion H.
  -
    (* n', m : nat
    HI : doble n' = doble m -> n' = m
    =====
    doble (S n') = doble m -> S n' = m *)
  intros H.
    (* n', m : nat
    HI : doble n' = doble m -> n' = m
    H : doble (S n') = doble m
    =====
    S n' = m *)
  destruct m as [| m'].
  +
    (* n' : nat
    HI : doble n' = doble 0 -> n' = 0
    H : doble (S n') = doble 0

```



```

=====
S n' = 0 *)
simpl in H. (* n' : nat
              HI : doble n' = doble 0 -> n' = 0
              H : S (S (doble n')) = 0
              =====
              S n' = 0 *)

inversion H.
+
(* n', m' : nat
  HI : doble n' = doble (S m') -> n' = S m'
  H : doble (S n') = doble (S m')
  =====
  S n' = S m' *)
apply funcional. (* n', m' : nat
                   HI : doble n' = doble (S m') -> n' = S m'
                   H : doble (S n') = doble (S m')
                   =====
                   n' = m' *)

Abort.

(* 2º intento *)
Theorem doble_inyectiva: forall n m,
  doble n = doble m ->
  n = m.
Proof.
intros n. (* n : nat
            =====
            forall m : nat, doble n = doble m -> n = m *)
induction n as [| n' HI].
-
(*
=====
forall m : nat, doble 0 = doble m -> 0 = m *)
simpl.
intros m H. (* m : nat
              H : 0 = doble m
              =====
              0 = m *)
destruct m as [| m'].
+
(* H : 0 = doble 0
=====

```



```

inversion H.      (* n' : nat
                    HI : forall m : nat, doble n' = doble m -> n' = m
                    m' : nat
                    H : S (S (doble n')) = doble (S m')
                    H1 : doble n' = doble m'
                    =====
                    doble n' = doble n' *)

reflexivity.

Qed.

(* -----
   Nota. Uso de la estrategia de generalización.
   ----- *)

(* -----
   Ejercicio 5.1. Demostrar que
   forall n m : nat,
     iguales_nat n m = true -> n = m.
   ----- *)

Theorem iguales_nat_true : forall n m : nat,
  iguales_nat n m = true -> n = m.

Proof.
  induction n as [|n' HIn'].
  -
    (*
      =====
      forall m : nat, iguales_nat 0 m = true
      -> 0 = m *)

    induction m as [|m' HIm'].
    +
      (*
        =====
        iguales_nat 0 0 = true -> 0 = 0 *)

      reflexivity.

    +
      (* m' : nat
         HIm' : iguales_nat 0 m' = true -> 0 = m'
         =====
         iguales_nat 0 (S m') = true -> 0 = S m' *)

      simpl.
      intros H.      (* m' : nat
                       HIm' : iguales_nat 0 m' = true -> 0 = m'

```

```

      H : false = true
      =====
      0 = S m' *)

inversion H.
-
      (* n' : nat
      HIn' : forall m:nat, iguales_nat n' m = true
              -> n' = m
      =====
      forall m : nat, iguales_nat (S n') m = true
              -> S n' = m *)

induction m as [|m' HIm'].
+
      (* n' : nat
      HIn' : forall m:nat, iguales_nat n' m = true
              -> n' = m
      =====
      iguales_nat (S n') 0 = true -> S n' = 0 *)
      (* false = true -> S n' = 0 *)
      (* n' : nat
      HIn' : forall m:nat, iguales_nat n' m = true
              -> n' = m

      H : false = true
      =====
      S n' = 0 *)

inversion H.
+
      (* n' : nat
      HIn' : forall m:nat, iguales_nat n' m = true
              -> n' = m

      m' : nat
      HIm' : iguales_nat (S n') m' = true
              -> S n' = m'
      =====
      iguales_nat (S n') (S m') = true
              -> S n' = S m' *)
      (* iguales_nat n' m' = true -> S n' = S m' *)
      (* n' : nat
      HIn' : forall m:nat, iguales_nat n' m = true
              -> n' = m

      m' : nat
      HIm' : iguales_nat (S n') m' = true
              -> S n' = m'

```

```

      H : iguales_nat n' m' = true
      =====
      S n' = S m' *)
apply HIn' in H.
      (* n' : nat
      HIn' : forall m:nat, iguales_nat n' m = true
          -> n' = m

      m' : nat
      HIm' : iguales_nat (S n') m' = true
          -> S n' = m'
      H : n' = m'
      =====
      S n' = S m' *)
      (* S m' = S m' *)

rewrite H.
reflexivity.

Qed.

(* -----
   Ejemplo 5.2. Demostrar que
     forall n m : nat,
       doble n = doble m ->
         n = m.
   ----- *)

(* 1º intento *)
Theorem doble_inyectiva_2a: forall n m : nat,
  doble n = doble m ->
    n = m.

Proof.
  intros n m.
  (* n, m : nat
  =====
  doble n = doble m -> n = m *)

  induction m as [| m' HI].
  -
    (* n : nat
    =====
    doble n = doble 0 -> n = 0 *)
    (* doble n = 0 -> n = 0 *)
    (* n : nat
    H : doble n = 0
    =====
    n = 0 *)
    simpl.
    intros H.

```

```

destruct n as [| n'].
+
    (* H : doble 0 = 0
       =====
       0 = 0 *)

    reflexivity.
+
    (* n' : nat
       H : doble (S n') = 0
       =====
       S n' = 0 *)

    simpl in H.
    (* n' : nat
       H : S (S (doble n')) = 0
       =====
       S n' = 0 *)

    inversion H.
-
    (* n, m' : nat
       HI : doble n = doble m' -> n = m'
       =====
       doble n = doble (S m') -> n = S m' *)

    intros H.
    (* n, m' : nat
       HI : doble n = doble m' -> n = m'
       H : doble n = doble (S m')
       =====
       n = S m' *)

    destruct n as [| n'].
    +
        (* m' : nat
           HI : doble 0 = doble m' -> 0 = m'
           H : doble 0 = doble (S m')
           =====
           0 = S m' *)

        simpl in H.
        (* m' : nat
           HI : doble 0 = doble m' -> 0 = m'
           H : 0 = S (S (doble m'))
           =====
           0 = S m' *)

        inversion H.
    +
        (* n', m' : nat
           HI : doble (S n') = doble m' -> S n' = m'
           H : doble (S n') = doble (S m')
           =====

```

```

      S n' = S m' *)
    apply funcional.
  Abort.

(* 2º intento *)
Theorem doble_inyectiva_2 : forall n m,
  doble n = doble m ->
  n = m.
Proof.
  intros n m.
  generalize dependent n.
  induction m as [| m' HI].
  -
    simpl.
    intros n H.
    destruct n as [| n'].
    +
      reflexivity.
    +
      simpl in H.
    inversion H.
  -
    (* n, m : nat
       =====
       doble n = doble m -> n = m *)
    (* m : nat
       =====
       forall n : nat, doble n = doble m -> n = m *)
    (*
       =====
       forall n : nat, doble n = doble 0 -> n = 0 *)
    (* forall n : nat, doble n = 0 -> n = 0 *)
    (* n : nat
       H : doble n = 0
       =====
       n = 0 *)
    (* H : doble 0 = 0
       =====
       0 = 0 *)
    (* n' : nat
       H : doble (S n') = 0
       =====
       S n' = 0 *)
    (* n' : nat
       H : S (S (doble n')) = 0
       =====
       S n' = 0 *)
    (* m' : nat
       HI : forall n : nat, doble n = doble m' -> n = m'

```

```

=====
forall n : nat, doble n = doble (S m')
  -> n = S m' *)

intros n H.
(* m' : nat
   HI : forall n : nat, doble n = doble m' -> n = m'
   n : nat
   H : doble n = doble (S m')
   =====
   n = S m' *)

destruct n as [| n'].
+
(* m' : nat
   HI : forall n : nat, doble n = doble m' -> n = m'
   H : doble 0 = doble (S m')
   =====
   0 = S m' *)

simpl in H.
(* m' : nat
   HI : forall n : nat, doble n = doble m' -> n = m'
   H : 0 = S (S (doble m'))
   =====
   0 = S m' *)

inversion H.
+
(* m' : nat
   HI : forall n : nat, doble n = doble m' -> n = m'
   n' : nat
   H : doble (S n') = doble (S m')
   =====
   S n' = S m' *)

apply funcional.
(* n' = m' *)
apply HI.
(* doble n' = doble m' *)
simpl in H.
(* m' : nat
   HI : forall n : nat, doble n = doble m' -> n = m'
   n' : nat
   H : S (S (doble n')) = S (S (doble m'))
   =====
   doble n' = doble m' *)

inversion H.
(* m' : nat
   HI : forall n : nat, doble n = doble m' -> n = m'
   n' : nat
   H : S (S (doble n')) = S (S (doble m'))
   H1 : doble n' = doble m'

```



```

=====
double n' = double n' *)

reflexivity.
Qed.

(* -----
   Nota. Uso de la táctica 'generalize dependent n'.
   ----- *)

(* -----
   Ejemplo 5.3. Demostrar que
       forall x y : id,
         iguales_id x y = true -> x = y.
   ----- *)

Theorem iguales_id_true: forall x y : id,
  iguales_id x y = true -> x = y.
Proof.
  intros [m] [n].          (* m, n : nat
                           =====
                           iguales_id (Id m) (Id n) = true -> Id m = Id n *)
  simpl.
  intros H.                (* iguales_nat m n = true -> Id m = Id n *)
                           (* m, n : nat
                           H : iguales_nat m n = true
                           =====
                           Id m = Id n *)

  assert (H' : m = n).
  -
    (* m, n : nat
    H : iguales_nat m n = true
    =====
    m = n *)

    apply iguales_nat_true. (* iguales_nat m n = true *)
    apply H.

  -
    (* m, n : nat
    H : iguales_nat m n = true
    H' : m = n
    =====
    Id m = Id n *)

    rewrite H'.
    reflexivity.
    (* Id n = Id n *)

```

**Qed.**

```
(* -----
  Ejercicio 5.2. Demostrar, por inducción sobre l,
    forall (n : nat) (X : Type) (xs : list X),
      longitud xs = n ->
        nthOpcional xs n = None.
  ----- *)
```

**Theorem** nthOpcional\_None: **forall** (n : **nat**) (X : **Type**) (xs : **list** X),  
 longitud xs = n ->  
 nthOpcional xs n = None.

**Proof.**

```
intros n X xs. (* n : nat
                  X : Type
                  xs : list X
                  =====
                  longitud xs = n -> nthOpcional xs n = None *)

generalize dependent n. (* X : Type
                           xs : list X
                           =====
                           forall n : nat,
                             longitud xs = n -> nthOpcional xs n = None *)

induction xs as [|x xs' HI].
- (* X : Type
   =====
   forall n : nat,
     longitud [] = n -> nthOpcional [] n = None *)

  reflexivity.
- (* X : Type
   x : X
   xs' : list X
   HI : forall n : nat,
        longitud xs' = n ->
        nthOpcional xs' n = None
   =====
   forall n : nat,
     longitud (x :: xs') = n ->
     nthOpcional (x :: xs') n = None *)

  destruct n as [|n'].
```

```

+
(* X : Type
  x : X
  xs' : list X
  HI : forall n : nat,
    longitud xs' = n ->
    nthOpcional xs' n = None
=====
  longitud (x :: xs') = 0 ->
  nthOpcional (x :: xs') 0 = None *)

intros H.

(* X : Type
  x : X
  xs' : list X
  HI : forall n : nat,
    longitud xs' = n ->
    nthOpcional xs' n = None
  H : longitud (x :: xs') = 0
=====
  nthOpcional (x :: xs') 0 = None *)

simpl in H.

(* X : Type
  x : X
  xs' : list X
  HI : forall n : nat,
    longitud xs' = n ->
    nthOpcional xs' n = None
  H : S (longitud xs') = 0
=====
  nthOpcional (x :: xs') 0 = None *)

inversion H.

+
(* X : Type
  x : X
  xs' : list X
  HI : forall n : nat,
    longitud xs' = n ->
    nthOpcional xs' n = None
  n' : nat
=====
  longitud (x :: xs') = S n' ->
  nthOpcional (x :: xs') (S n') = None *)

simpl.
(* S (longitud xs') = S n' ->
  nthOpcional xs' n' = None *)

```

```

intros H.
(* X : Type
   x : X
   xs' : list X
   HI : forall n : nat,
        longitud xs' = n ->
        nthOpcional xs' n = None
   n' : nat
   H : S (longitud xs') = S n'
   =====
   nthOpcional xs' n' = None *)
apply HI.
inversion H.
(* X : Type
   x : X
   xs' : list X
   HI : forall n : nat,
        longitud xs' = n ->
        nthOpcional xs' n = None
   n' : nat
   H : S (longitud xs') = S n'
   H1 : longitud xs' = n'
   =====
   longitud xs' = longitud xs' *)

reflexivity.
Qed.

(* =====
   § 6. Expansión de definiciones
   ===== *)

(* -----
   Ejemplo 6.1. Definir la función
   cuadrado : nat -> nat
   tal que (cuadrado n) es el cuadrado de n.
   ----- *)

Definition cuadrado (n:nat) : nat := n * n.

(* -----
   Ejemplo 6.2. Demostrar que
   forall n m : nat,

```

```
cuadrado (n * m) = cuadrado n * cuadrado m.
```

```
----- *)
```

**Lemma** cuadrado\_mult : **forall** n m : **nat**,  
cuadrado (n \* m) = cuadrado n \* cuadrado m.

**Proof.**

**intros** n m.

```
(* n, m : nat
```

```
=====
```

```
cuadrado (n * m) =  
cuadrado n * cuadrado m *)
```

**unfold** cuadrado.

```
(* (n * m) * (n * m) =  
(n * n) * (m * m) *)
```

**rewrite** producto\_asociativa.

```
(* ((n * m) * n) * m =  
(n * n) * (m * m) *)
```

**assert** (H : (n \* m) \* n = (n \* n) \* m).

```
-
```

```
(* (n * m) * n = (n * n) * m) *)
```

**rewrite** producto\_commutativa.

```
(* n * (n * m) = (n * n) * m *)
```

**apply** producto\_asociativa.

```
-
```

```
(* n, m : nat
```

```
H : (n * m) * n = (n * n) * m
```

```
=====
```

```
((n * m) * n) * m =  
(n * n) * (m * m) *)
```

**rewrite** H.

```
(* ((n * n) * m) * m =  
(n * n) * (m * m) *)
```

**rewrite** producto\_asociativa.

```
(* ((n * n) * m) * m =  
(n * n) * (m * m) *)
```

**reflexivity.**

**Qed.**

```
(* -----  
Nota. Uso de la táctica 'unfold'  
----- *)
```

```
(* -----  
Ejemplo 6.4. Definir la función  
const5 : nat -> nat  
tal que (const5 x) es el número 5.  
----- *)
```

**Definition** const5 (x: nat) : nat := 5.

```
(* -----
  Ejemplo 6.5. Demostrar que
    forall m : nat,
      const5 m + 1 = const5 (m + 1) + 1.
  ----- *)
```

**Fact** prop\_const5 : forall m : nat,  
const5 m + 1 = const5 (m + 1) + 1.

**Proof.**

```
  intros m.      (* m : nat
                  =====
                  const5 m + 1 = const5 (m + 1) + 1 *)
  simpl.         (* 6 = 6 *)
  reflexivity.
```

**Qed.**

```
(* -----
  Nota. Expansión automática de la definición de const5.
  ----- *)
```

```
(* -----
  Ejemplo 6.6. Se coconsidera la siguiente definición
  Definition const5b (x:nat) : nat :=
    match x with
    | 0   => 5
    | S _ => 5
    end.

  Demostrar que
    forall m : nat,
      const5b m + 1 = const5b (m + 1) + 1.
  ----- *)
```

**Definition** const5b (x:nat) : nat :=  
match x with  
| 0 => 5  
| S \_ => 5  
end.

```

(* 1ª intento *)
Fact prop_const5b_1: forall m : nat,
  const5b m + 1 = const5b (m + 1) + 1.
Proof.
  intros m. (* m : nat
    =====
    const5b m + 1 = const5b (m + 1) + 1 *)
  simpl. (* const5b m + 1 = const5b (m + 1) + 1 *)
Abort.

(* 1ª demostración *)
Fact prop_const5b_2: forall m : nat,
  const5b m + 1 = const5b (m + 1) + 1.
Proof.
  intros m. (* m : nat
    =====
    const5b m + 1 = const5b (m + 1) + 1 *)
  destruct m.
  - (*
    =====
    const5b 0 + 1 = const5b (0 + 1) + 1 *)
    simpl. (* 6 = 6 *)
    reflexivity.
  - (* m : nat
    =====
    const5b (S m) + 1 = const5b (S m + 1) + 1 *)
    simpl. (* 6 = 6 *)
    reflexivity.
Qed.

(* 2ª demostración *)
Fact prop_const5b_3: forall m : nat,
  const5b m + 1 = const5b (m + 1) + 1.
Proof.
  intros m. (* m : nat
    =====
    const5b m + 1 = const5b (m + 1) + 1 *)
  unfold const5b. (* m : nat
    =====

```

```

        match m with
        | 0 | _ => 5
        end + 1 = match m + 1 with
        | 0 | _ => 5
        end + 1 *)

destruct m.

-
    (*
    =====
    5 + 1 = match 0 + 1 with
    | 0 | _ => 5
    end + 1 *)

    reflexivity.

-
    (* m : nat
    =====
    5 + 1 = match S m + 1 with
    | 0 | _ => 5
    end + 1 *)

    reflexivity.

Qed.

(* =====
§ 7. Uso de 'destruct' sobre expresiones compuestas
===== *)

(* -----
Ejemplo 7.1. Se considera la siguiente definición
Definition const_false (n : nat) : bool :=
  if iguales_nat n 3 then false
  else if iguales_nat n 5 then false
  else false.

Demostrar que
forall n : nat,
  const_false n = false.
----- *)

Definition const_false (n : nat) : bool :=
  if iguales_nat n 3 then false
  else if iguales_nat n 5 then false
  else false.

```



**Theorem** const\_false\_false : forall n : nat,  
const\_false n = false.

**Proof.**

```

intros n.                                (* n : nat
                                           =====
                                           const_false n = false *)

unfold const_false.                      (* (if iguales_nat n 3 then false
                                           else if iguales_nat n 5 then false
                                           else false) =
                                           false *)

destruct (iguales_nat n 3).
-                                           (* n : nat
                                           =====
                                           false = false *)

  reflexivity.
-                                           (* n : nat
                                           =====
                                           (if iguales_nat n 5 then false
                                           else false) = false *)

  destruct (iguales_nat n 5).
+                                           (* n : nat
                                           =====
                                           false = false *)

    reflexivity.
+                                           (* n : nat
                                           =====
                                           false = false *)

      reflexivity.

```

**Qed.**

```

(* -----
Ejemplo 7.2. Se considera la siguiente definición
Definition ej (n : nat) : bool :=
  if      iguales_nat n 3 then true
  else if iguales_nat n 5 then true
  else      false.

```

Demostrar que  
forall n : nat,

```

    ej n = true -> esImpar n = true.
    ----- *)

Definition ej (n : nat) : bool :=
  if      iguales_nat n 3 then true
  else if iguales_nat n 5 then true
  else      false.

(* 1º intento *)
Theorem ej_impar_a: forall n : nat,
  ej n = true ->
  esImpar n = true.
Proof.
  intros n H.
  unfold ej in H. (* n : nat

  (* n : nat
    H : ej n = true
    =====
    esImpar n = true *)

    H : (if iguales_nat n 3
          then true
          else if iguales_nat n 5
              then true
              else false)
        = true
    =====
    esImpar n = true *)

  destruct (iguales_nat n 3).
  -
  (* n : nat
    H : true = true
    =====
    esImpar n = true *)

Abort.

(* 2º intento *)
Theorem ej_impar : forall n : nat,
  ej n = true ->
  esImpar n = true.
Proof.
  intros n H.
  (* n : nat
    H : ej n = true

```

```
unfold ej in H.
```

```
destruct (iguales_nat n 3) eqn: H3.
```

```
-
```

```
apply iguales_nat_true in H3.
```

```
rewrite H3.
```

```
reflexivity.
```

```
-
```

```
destruct (iguales_nat n 5) eqn:H5.
```

```
+
```

```
apply iguales_nat_true in H5.
```

```
=====
esImpar n = true *)
(* n : nat
H : (if iguales_nat n 3
    then true
    else if iguales_nat n 5
        then true else false)
= true
=====
esImpar n = true *)
(* n : nat
H3 : iguales_nat n 3 = true
H : true = true
=====
esImpar n = true *)
(* n : nat
H3 : n = 3
H : true = true
=====
esImpar n = true *)
(* esImpar 3 = true *)

(* n : nat
H3 : iguales_nat n 3 = false
H : (if iguales_nat n 5
    then true else false)
= true
=====
esImpar n = true *)

(* n : nat
H3 : iguales_nat n 3 = false
H5 : iguales_nat n 5 = true
H : true = true
=====
esImpar n = true *)
(* n : nat
H3 : iguales_nat n 3 = false
H5 : n = 5
```

```

rewrite H5.
reflexivity.
+
inversion H.
Qed.

(* -----
Nota. Uso de la táctica 'destruct e eqn: H'.
----- *)

(* -----
Ejercicio 7.1. Demostrar que desempareja y empareja son inversas; es decir,
forall X Y (ps : list (X * Y)) xs ys,
  desempareja ps = (xs, ys) ->
  empareja xs ys = ps.
----- *)

Theorem empareja_desempareja: forall X Y (ps : list (X * Y)) xs ys,
  desempareja ps = (xs, ys) ->
  empareja xs ys = ps.
Proof.
  intros X Y ps.

  (* X : Type
   Y : Type
   ps : list (X * Y)
   =====
   forall (xs : list X) (ys : list Y),
     desempareja ps = (xs, ys) ->
     empareja xs ys = ps *)

  induction ps as [| (x,y) ps' HI].
  -
    (* X : Type
     Y : Type
     =====

```

**intros** xs ys H.

**simpl in** H.

**inversion** H.

**simpl.**  
**reflexivity.**

```

forall (xs : list X) (ys : list Y),
  desempareja [ ] = (xs, ys) ->
  empareja xs ys = [ ] *)
(* X : Type
   Y : Type
   xs : list X
   ys : list Y
   H : desempareja [ ] = (xs, ys)
   =====
   empareja xs ys = [ ] *)
(* X : Type
   Y : Type
   xs : list X
   ys : list Y
   H : ([ ], [ ]) = (xs, ys)
   =====
   empareja xs ys = [ ] *)
(* X : Type
   Y : Type
   xs : list X
   ys : list Y
   H : ([ ], [ ]) = (xs, ys)
   H1 : [ ] = xs
   H2 : [ ] = ys
   =====
   empareja [ ] [ ] = [ ] *)
(* [ ] = [ ] *)

(* X : Type
   Y : Type
   x : X
   y : Y
   ps' : list (X * Y)
   HI : forall (xs:list X) (ys:list Y),
        desempareja ps' = (xs, ys)
        -> empareja xs ys = ps'
   =====
   forall (xs : list X) (ys : list Y),
     desempareja ((x,y)::ps') = (xs,ys)
     -> empareja xs ys = (x,y)::ps' *)

```

**intros** xs ys H.

```
(* X : Type
   Y : Type
   x : X
   y : Y
   ps' : list (X * Y)
   HI : forall (xs:list X) (ys:list Y),
        desempareja ps' = (xs, ys) ->
        empareja xs ys = ps'
   xs : list X
   ys : list Y
   H : desempareja ((x,y)::ps') = (xs,ys)
   =====
   empareja xs ys = (x, y) :: ps' *)
```

**destruct** (desempareja ps') eqn: E.

```
(* Y : Type
   x : X
   y : Y
   ps' : list (X * Y)
   l : list X
   l0 : list Y
   E : desempareja ps' = (l, l0)
   HI : forall (xs:list X) (ys:list Y),
        (l, l0) = (xs, ys) ->
        empareja xs ys = ps'
   xs : list X
   ys : list Y
   H : desempareja ((x,y)::ps') = (xs,ys)
   =====
   empareja xs ys = (x, y) :: ps' *)
```

**simpl in** H.

```
(* X : Type
   Y : Type
   x : X
   y : Y
   ps' : list (X * Y)
   l : list X
   l0 : list Y
   E : desempareja ps' = (l, l0)
   HI : forall (xs: ist X) (ys:list Y),
        (l, l0) = (xs, ys) ->
        empareja xs ys = ps'
   xs : list X
```

**rewrite** E **in** H.

**inversion** H.

```

ys : list Y
H : match desempareja ps' with
    | (xs, ys) => (x :: xs, y :: ys)
    end = (xs, ys)
=====
empareja xs ys = (x, y) :: ps' *)
(* X : Type
   Y : Type
   x : X
   y : Y
   ps' : list (X * Y)
   l : list X
   l0 : list Y
   E : desempareja ps' = (l, l0)
   HI : forall (xs:list X) (ys:list Y),
        (l, l0) = (xs, ys) ->
        empareja xs ys = ps'
   xs : list X
   ys : list Y
   H : (x :: l, y :: l0) = (xs, ys)
   =====
   empareja xs ys = (x, y) :: ps' *)
(* X : Type
   Y : Type
   x : X
   y : Y
   ps' : list (X * Y)
   l : list X
   l0 : list Y
   E : desempareja ps' = (l, l0)
   HI : forall (xs:list X) (ys:list Y),
        (l, l0) = (xs, ys) ->
        empareja xs ys = ps'
   xs : list X
   ys : list Y
   H : (x :: l, y :: l0) = (xs, ys)
   H1 : x :: l = xs
   H2 : y :: l0 = ys
   =====
   empareja (x :: l) (y :: l0) =

```

**simpl.**

**rewrite** HI.

+

**reflexivity.**

+

```

      (x, y) :: ps' *)
(* (x, y) :: empareja l l0 =
    (x, y) :: ps' *)

(* X : Type
   Y : Type
   x : X
   y : Y
   ps' : list (X * Y)
   l : list X
   l0 : list Y
   E : desempareja ps' = (l, l0)
   HI : forall (xs:list X) (ys:list Y),
        (l, l0) = (xs, ys) ->
        empareja xs ys = ps'
   xs : list X
   ys : list Y
   H : (x :: l, y :: l0) = (xs, ys)
   H1 : x :: l = xs
   H2 : y :: l0 = ys
   =====
   (x, y) :: ps' = (x, y) :: ps' *)

(* X : Type
   Y : Type
   x : X
   y : Y
   ps' : list (X * Y)
   l : list X
   l0 : list Y
   E : desempareja ps' = (l, l0)
   HI : forall (xs:list X) (ys:list Y),
        (l, l0) = (xs, ys) ->
        empareja xs ys = ps'
   xs : list X
   ys : list Y
   H : (x :: l, y :: l0) = (xs, ys)
   H1 : x :: l = xs
   H2 : y :: l0 = ys
   =====

```



```

                                (l, l0) = (l, l0)
*)
    reflexivity.
Qed.

(* -----
   Ejercicio 7.2. Demostrar que
   forall (f : bool -> bool) (b : bool),
       f (f (f b)) = f b.
   ----- *)

Theorem bool_tres_veces:
  forall (f : bool -> bool) (b : bool),
    f (f (f b)) = f b.
Proof.
  intros f b.
                                (* f : bool -> bool
                                   b : bool
                                   =====
                                   f (f (f b)) = f b *)

  destruct b.
  -
                                (* f : bool -> bool
                                   =====
                                   f (f (f true)) = f true *)

    destruct (f true) eqn:H1.
    +
                                (* f : bool -> bool
                                   H1 : f true = true
                                   =====
                                   f (f true) = true *)
                                (* f true = true *)

      rewrite H1.
      apply H1.
    +
                                (* f : bool -> bool
                                   H1 : f true = false
                                   =====
                                   f (f false) = false *)

    destruct (f false) eqn:H2.
    *
                                (* f : bool -> bool
                                   H1 : f true = false
                                   H2 : f false = true
                                   =====
                                   f true = false *)

```

```

    apply H1.
  *
    (* f : bool -> bool
       H1 : f true = false
       H2 : f false = false
       =====
       f false = false *)

    apply H2.
  -
    destruct (f false) eqn:H3.
  +
    (* f : bool -> bool
       H3 : f false = true
       =====
       f (f true) = true *)

    destruct (f true) eqn:H4.
  *
    (* f : bool -> bool
       H3 : f false = true
       H4 : f true = true
       =====
       f true = true *)

    apply H4.
  *
    (* f : bool -> bool
       H3 : f false = true
       H4 : f true = false
       =====
       f false = true *)

    apply H3.
  +
    (* f : bool -> bool
       H3 : f false = false
       =====
       f (f false) = false *)

    rewrite H3.
    apply H3.

Qed.

(* =====
   § 8. Ejercicios
   ===== *)

```

```
(* -----
  Ejercicio 8.1. Demostrar que
    forall (n m : nat),
      iguales_nat n m = iguales_nat m n.
  ----- *)
```

**Theorem** iguales\_nat\_simetrica: **forall** n m : **nat**,  
 iguales\_nat n m = iguales\_nat m n.

**Proof.**

```
  intros n m.
  destruct (iguales_nat n m) eqn:H1.
  -
    apply iguales_nat_true in H1.
    rewrite H1.
    symmetry.
    apply iguales_nat_refl.
  -
    destruct (iguales_nat m n) eqn:H2.
    +
      apply iguales_nat_true in H2.
      rewrite H2 in H1.
    (* n, m : nat
      =====
      iguales_nat n m = iguales_nat m n *)
    (* n, m : nat
      H1 : iguales_nat n m = true
      =====
      true = iguales_nat m n *)
    (* n, m : nat
      H1 : n = m
      =====
      true = iguales_nat m n *)
    (* true = iguales_nat m m *)
    (* iguales_nat m m = true *)
    (* n, m : nat
      H1 : iguales_nat n m = false
      =====
      false = iguales_nat m n *)
    (* n, m : nat
      H1 : iguales_nat n m = false
      H2 : iguales_nat m n = true
      =====
      false = true *)
    (* n, m : nat
      H1 : iguales_nat n m = false
      H2 : m = n
      =====
      false = true *)
    (* n, m : nat
```

```

H1 : iguales_nat n n = false
H2 : m = n
=====
false = true *)

rewrite iguales_nat_refl in H1. (* n, m : nat
H1 : true = false
H2 : m = n
=====
false = true *)

inversion H1.
+
(* n, m : nat
H1 : iguales_nat n m = false
H2 : iguales_nat m n = false
=====
false = false *)

reflexivity.
Qed.

(* -----
Ejercicio 8.2. Demostrar que
  forall n m p : nat,
    iguales_nat n m = true ->
    iguales_nat m p = true ->
    iguales_nat n p = true.
----- *)

Theorem iguales_nat_trans: forall n m p : nat,
  iguales_nat n m = true ->
  iguales_nat m p = true ->
  iguales_nat n p = true.

Proof.
  intros n m p H1 H2. (* n, m, p : nat
H1 : iguales_nat n m = true
H2 : iguales_nat m p = true
=====
iguales_nat n p = true *)

  apply iguales_nat_true in H1. (* n, m, p : nat
H1 : n = m
H2 : iguales_nat m p = true
=====

```

```

                                iguales_nat n p = true *)
apply iguales_nat_true in H2. (* n, m, p : nat
                                H1 : n = m
                                H2 : m = p
                                =====
                                iguales_nat n p = true *)
rewrite H1.                    (* iguales_nat m p = true *)
rewrite H2.                    (* iguales_nat p p = true *)
apply iguales_nat_refl.
Qed.

(* -----
   Ejercicio 8.3. Definir las hipótesis sobre xs e ys para que se cumpla
   la propiedad
       desempareja (empareja xs ys) = (xs,ys).
   y demostrarla.
   ----- *)

(* En la prueba se usará el siguiente lema *)
Lemma longitud_cero: forall (X : Type) (xs : list X),
  longitud xs = 0 -> xs = [].
Proof.
  intros X xs H.              (* X : Type
                               xs : list X
                               H : longitud xs = 0
                               =====
                               xs = [ ] *)

  destruct xs as [|x xs'].
  -
    (* X : Type
       H : longitud [ ] = 0
       =====
       [ ] = [ ] *)

    reflexivity.
  -
    (* X : Type
       x : X
       xs' : list X
       H : longitud (x :: xs') = 0
       =====
       x :: xs' = [ ] *)

  simpl in H.                (* X : Type

```

```

x : X
xs' : list X
H : S (longitud xs') = 0
=====
x :: xs' = [ ] *)

inversion H.
Qed.

Theorem desempareja_empareja: forall (X : Type) (xs ys: list X),
  longitud xs = longitud ys ->
  desempareja (empareja xs ys) = (xs,ys).
Proof.
  intros X xs.
    (* X : Type
       xs : list X
       =====
       forall ys : list X,
         longitud xs = longitud ys ->
         desempareja (empareja xs ys) = (xs, ys) *)

  induction xs as [|x xs' HI1].
  -
    (* X : Type
       =====
       forall ys : list X,
         longitud [ ] = longitud ys ->
         desempareja (empareja [ ] ys) = ([ ], ys) *)

  intros ys H.
    (* X : Type
       ys : list X
       H : longitud [ ] = longitud ys
       =====
       desempareja (empareja [ ] ys) = ([ ], ys) *)

  simpl in H.
    (* X : Type
       ys : list X
       H : 0 = longitud ys
       =====
       desempareja (empareja [ ] ys) = ([ ], ys) *)

  symmetry in H.
    (* X : Type
       ys : list X
       H : longitud ys = 0
       =====
       desempareja (empareja [ ] ys) = ([ ], ys) *)

  apply longitud_cero in H.
    (* X : Type

```

```

rewrite H.
simpl.
reflexivity.

```

```

-

```

```

intros ys.

```

```

destruct ys as [|y ys'].

```

```

+

```

```

ys : list X
H : ys = [ ]
=====
desempareja (empareja [] ys) = ([], ys) *)
(* desempareja (empareja [] []) = ([], []) *)
(* ([], []) = ([], []) *)

(* X : Type
x : X
xs' : list X
HI1 : forall ys : list X,
      longitud xs' = longitud ys ->
      desempareja (empareja xs' ys)
      = (xs', ys)
=====
forall ys : list X,
longitud (x :: xs') = longitud ys ->
desempareja (empareja (x :: xs') ys)
= (x :: xs', ys) *)

(* X : Type
x : X
xs' : list X
HI1 : forall ys : list X,
      longitud xs' = longitud ys ->
      desempareja (empareja xs' ys)
      = (xs', ys)
ys : list X
=====
longitud (x :: xs') = longitud ys ->
desempareja (empareja (x :: xs') ys)
= (x :: xs', ys) *)

(* X : Type
x : X
xs' : list X
HI1 : forall ys : list X,
      longitud xs' = longitud ys ->
      desempareja (empareja xs' ys)
      = (xs', ys)
=====

```

**intros** H.

```

longitud (x :: xs') = longitud [ ] ->
desempareja (empareja (x :: xs') [ ])
= (x :: xs', [ ]) *)
(* X : Type
   x : X
   xs' : list X
   HI1 : forall ys : list X,
        longitud xs' = longitud ys ->
        desempareja (empareja xs' ys)
        = (xs', ys)
   H : longitud (x :: xs') = longitud [ ]
   =====
   desempareja (empareja (x :: xs') [ ])
   = (x :: xs', [ ]) *)

```

**simpl in** H.

```

(* X : Type
   x : X
   xs' : list X
   HI1 : forall ys : list X,
        longitud xs' = longitud ys ->
        desempareja (empareja xs' ys)
        = (xs', ys)
   H : S (longitud xs') = 0
   =====
   desempareja (empareja (x :: xs') [ ])
   = (x :: xs', [ ]) *)

```

**inversion** H.

+

```

(* X : Type
   x : X
   xs' : list X
   HI1 : forall ys : list X,
        longitud xs' = longitud ys ->
        desempareja (empareja xs' ys)
        = (xs', ys)
   y : X
   ys' : list X
   =====
   longitud (x :: xs') = longitud (y :: ys') ->
   desempareja (empareja (x::xs') (y::ys'))
   = (x :: xs', y :: ys') *)

```

**intros** H.

```

(* X : Type

```



```

x : X
xs' : list X
HI1 : forall ys : list X,
      longitud xs' = longitud ys ->
      desempareja (empareja xs' ys)
      = (xs', ys)

y : X
ys' : list X
H : longitud (x :: xs') = longitud (y :: ys')
=====
desempareja (empareja (x::xs') (y::ys'))
= (x :: xs', y :: ys') *)

inversion H.
(* X : Type
x : X
xs' : list X
HI1 : forall ys : list X,
      longitud xs' = longitud ys ->
      desempareja (empareja xs' ys)
      = (xs', ys)

y : X
ys' : list X
H : longitud (x :: xs') = longitud (y :: ys')
H1 : longitud xs' = longitud ys'
=====
desempareja (empareja (x::xs') (y::ys'))
= (x :: xs', y :: ys') *)

apply HI1 in H1.
(* X : Type
x : X
xs' : list X
HI1 : forall ys : list X,
      longitud xs' = longitud ys ->
      desempareja (empareja xs' ys)
      = (xs', ys)

y : X
ys' : list X
H : longitud (x :: xs') = longitud (y :: ys')
H1 : desempareja (empareja xs' ys')
      = (xs', ys')
=====
desempareja (empareja (x::xs') (y::ys'))

```

```

    = (x :: xs', y :: ys') *)
  simpl.
    (* match desempareja (empareja xs' ys') with
       | (xs, ys) => (x :: xs, y :: ys)
       end
       = (x :: xs', y :: ys') *)
  rewrite H1.
  reflexivity.
Qed.

(* -----
   Ejercicio 8.4. Demostrar que
   forall (X : Type) (p : X -> bool) (x : X) (xs ys : list X),
     filtra p xs = x :: ys ->
     p x = true.
   ----- *)

Theorem prop_filtra:
  forall (X : Type) (p : X -> bool) (x : X) (xs ys : list X),
    filtra p xs = x :: ys ->
    p x = true.
Proof.
  intros X p x xs ys.
    (* X : Type
       p : X -> bool
       x : X
       xs, ys : list X
       =====
       filtra p xs = x :: ys -> p x = true *)

  induction xs as [|x' xs' HI].
  -
    (* X : Type
       p : X -> bool
       x : X
       ys : list X
       =====
       filtra p [ ] = x :: ys -> p x = true *)
    simpl.
    intros H.
    (* [ ] = x :: ys -> p x = true *)
    (* X : Type
       p : X -> bool
       x : X
       ys : list X
       H : [ ] = x :: ys

```

**inversion** H.

-

**destruct** (p x') eqn:Hx'.

+

**simpl.**

**rewrite** Hx'.

**intros** H.

**inversion** H.

```

=====
p x = true *)

(* X : Type
   p : X -> bool
   x, x' : X
   xs', ys : list X
   HI : filtra p xs' = x :: ys -> p x = true
   =====
   filtra p (x'::xs') = x::ys -> p x = true *)

(* X : Type
   p : X -> bool
   x, x' : X
   xs', ys : list X
   HI : filtra p xs' = x :: ys -> p x = true
   Hx' : p x' = true
   =====
   filtra p (x'::xs') = x::ys -> p x = true *)
(* (if p x' then x' :: filtra p xs'
    else filtra p xs')
   = x :: ys -> p x = true *)
(* x' :: filtra p xs' = x :: ys -> p x = true *)
(* X : Type
   p : X -> bool
   x, x' : X
   xs', ys : list X
   HI : filtra p xs' = x :: ys -> p x = true
   Hx' : p x' = true
   H : x' :: filtra p xs' = x :: ys
   =====
   p x = true *)
(* X : Type
   p : X -> bool
   x, x' : X
   xs', ys : list X
   HI : filtra p xs' = x :: ys -> p x = true
   Hx' : p x' = true
   H : x' :: filtra p xs' = x :: ys
   H1 : x' = x

```

```

H2 : filtra p xs' = ys
=====
p x = true *)
rewrite H1 in Hx'.
(* X : Type
   p : X -> bool
   x, x' : X
   xs', ys : list X
   H1 : filtra p xs' = x :: ys -> p x = true
   Hx' : p x = true
   H : x' :: filtra p xs' = x :: ys
   H1 : x' = x
   H2 : filtra p xs' = ys
   =====
   p x = true *)

apply Hx'.
+
(* X : Type
   p : X -> bool
   x, x' : X
   xs', ys : list X
   H1 : filtra p xs' = x :: ys -> p x = true
   Hx' : p x' = false
   =====
   filtra p (x'::xs') = x::ys -> p x = true *)
simpl.
(* (if p x' then x' :: filtra p xs'
    else filtra p xs')
   = x :: ys -> p x = true *)
rewrite Hx'.
apply H1.
Qed.

(* -----
Ejercicio 8.5.1. Definir, por recursión, la función
  todos {X : Type} (p : X -> bool) (xs : list X) : bool
tal que (todos p xs) se verifica si todos los elementos de xs cumplen
p. Por ejemplo,
  todos esImpar [1;3;5;7;9]      = true
  todos negacion [false;false]  = true
  todos esPar [0;2;4;5]         = false
  todos (iguales_nat 5) []      = true
----- *)

```

```

Fixpoint todos {X : Type} (p : X -> bool) (xs : list X) : bool :=
  match xs with
  | nil      => true
  | x::xs'   => conjuncion (p x) (todos p xs')
  end.

```

```

Compute (todos esImpar [1;3;5;7;9]).
(* = true : bool*)
Compute (todos negacion [false;false]).
(* = true : bool*)
Compute (todos esPar [0;2;4;5]).
(* = false : bool*)
Compute (todos (iguales_nat 5) []).
(* = true : bool *)

```

```

(* -----
   Ejercicio 8.5.2. Definir, por recursión, la función
       existe
   tal que (existe p xs) se verifica si algún elemento de xs cumple
   p. Por ejemplo,
       existe (iguales_nat 5) [0;2;3;6]           = false
       existe (conjuncion true) [true;true;false] = true
       existe esImpar [1;0;0;0;0;3]              = true
       existe esPar []                             = false
   ----- *)

```

```

Fixpoint existe {X : Type} (p : X -> bool) (xs : list X) : bool :=
  match xs with
  | nil      => false
  | x::xs'   => disyuncion (p x) (existe p xs')
  end.

```

```

Compute (existe (iguales_nat 5) [0;2;3;6]).
(* = false : bool *)
Compute (existe (conjuncion true) [true;true;false]).
(* = true : bool *)
Compute (existe esImpar [1;0;0;0;0;3]).
(* = true : bool *)
Compute (existe esPar []).

```

```
(* = false : bool *)

(* -----
   Ejercicio 8.5.3. Redefinir, usando todos y negb, la función existe2 y
   demostrar su equivalencia con existe.
   ----- *)
```

**Definition** existe2 {X : Type} (p : X -> bool) (xs : list X) : bool :=  
negacion (todos (fun y => negacion (p y)) xs).

```
Compute (existe2 (iguales_nat 5) [0;2;3;6]).
(* = false : bool *)
Compute (existe2 (conjuncion true) [true;true;false]).
(* = true : bool *)
Compute (existe2 esImpar [1;0;0;0;0;3]).
(* = true : bool *)
Compute (existe2 esPar []).
(* = false : bool *)
```

**Theorem** equiv\_existe: forall (X : Type) (p : X -> bool) (xs : list X),  
existe p xs = existe2 p xs.

**Proof.**

```
intros X p xs.
(* X : Type
   p : X -> bool
   xs : list X
   =====
   existe p xs = existe2 p xs *)

induction xs as [|x xs' HI].
-
(* X : Type
   p : X -> bool
   =====
   existe p [] = existe2 p [] *)
unfold existe2.
(* existe p [] =
   negacion (todos (fun y : X => negacion (p y))
                  []) *)
simpl.
reflexivity.
-
(* X : Type
   p : X -> bool
   x : X
```

```

xs' : list X
HI : existe p xs' = existe2 p xs'
=====
existe p (x :: xs') = existe2 p (x :: xs') *)

destruct (p x) eqn:Hx.
+
(* X : Type
   p : X -> bool
   x : X
   xs' : list X
   HI : existe p xs' = existe2 p xs'
   Hx : p x = true
   =====
   existe p (x :: xs') = existe2 p (x :: xs') *)
unfold existe2.
(* existe p (x :: xs') =
   negacion (todos (fun y : X => negacion (p y))
                  (x :: xs')) *)
simpl.
(* p x || existe p xs' =
   negacion
     (negacion
      (p x) &&
      todos (fun y : X => negacion (p y)) xs') *)
rewrite Hx.
(* true || existe p xs' =
   negacion
     (negacion true &&
      todos (fun y : X => negacion (p y)) xs')*)
simpl.
reflexivity.
+
(* X : Type
   p : X -> bool
   x : X
   xs' : list X
   HI : existe p xs' = existe2 p xs'
   Hx : p x = false
   =====
   existe p (x :: xs') = existe2 p (x :: xs') *)
unfold existe2.
(* existe p (x :: xs') =
   negacion
     (todos (fun y : X => negacion (p y))
            (x :: xs')) *)
simpl.
(* p x || existe p xs' =

```

```

negacion
  (negacion (p x) &&
   todos (fun y : X => negacion (p y)) xs') *)
rewrite Hx.
(* false || existe p xs' =
negacion (
  negacion false &&
  todos (fun y : X => negacion (p y)) xs') *)
simpl.
(* existe p xs' =
negacion
  (todos (fun y : X => negacion (p y)) xs') *)
rewrite HI.
(* existe2 p xs' =
negacion
  (todos (fun y : X => negacion (p y)) xs') *)
unfold existe2.
(* negacion
  (todos (fun y : X => negacion (p y)) xs') =
negacion
  (todos (fun y : X => negacion (p y)) xs') *)
reflexivity.
Qed.

(* =====
§ 9. Resumen de tácticas básicas
===== *)

(* Las tácticas básicas utilizadas hasta ahora son
+ apply H:
+ si el objetivo coincide con la hipótesis H, lo demuestra;
+ si H es una implicación,
+ si el objetivo coincide con la conclusión de H, lo sustituye por
  su premisa y
+ si el objetivo coincide con la premisa de H, lo sustituye por
  su conclusión.

+ apply ... with ...: Especifica los valores de las variables que no
  se pueden deducir por emparejamiento.

+ apply H1 in H2: Aplica la igualdad de la hipótesis H1 a la
  hipótesis H2.

+ assert (H: P): Incluye la demostración de la propiedad P y continúa

```



- la demostración añadiendo como premisa la propiedad  $P$  con nombre  $H$ .*
- + *destruct  $b$ : Distingue dos casos según que  $b$  sea  $\text{True}$  o  $\text{False}$ .*
  - + *destruct  $n$  as  $[| n1]$ : Distingue dos casos según que  $n$  sea  $0$  o sea  $S\ n1$ .*
  - + *destruct  $p$  as  $[n\ m]$ : Sustituye el par  $p$  por  $(n,m)$ .*
  - + *destruct  $e$  eqn:  $H$ : Distingue casos según el valor de la expresión  $e$  y lo añade al contexto la hipótesis  $H$ .*
  - + *generalize dependent  $x$ : Mueve la variable  $x$  (y las que dependan de ella) del contexto a una hipótesis explícita en el objetivo.*
  - + *induction  $n$  as  $[|n1\ IHn1]$ : Inicia una demostración por inducción sobre  $n$ . El caso base en  $\sim n\ 0 \sim$ . El paso de la inducción consiste en suponer la propiedad para  $\sim n1 \sim$  y demostrarla para  $\sim S\ n1 \sim$ . El nombre de la hipótesis de inducción es  $\sim IHn1 \sim$ .*
  - + *intros vars: Introduce las variables del cuantificador universal y, como premisas, los antecedentes de las implicaciones.*
  - + *inversion: Aplica que los constructores son disjuntos e inyectivos.*
  - + *reflexivity: Demuestra el objetivo si es una igualdad trivial.*
  - + *rewrite  $H$ : Sustituye el término izquierdo de  $H$  por el derecho.*
  - + *rewrite  $\leftarrow H$ : Sustituye el término derecho de  $H$  por el izquierdo.*
  - + *simpl: Simplifica el objetivo.*
  - + *simpl in  $H$ : Simplifica la hipótesis  $H$ .*
  - + *symmetry: Cambia un objetivo de la forma  $s = t$  en  $t = s$ .*
  - + *symmetry in  $H$ : Cambia la hipótesis  $H$  de la forma  $\sim st \sim$  en  $\sim ts \sim$ .*
  - + *unfold  $f$ : Expande la definición de la función  $f$ .*
- \*)

```
(* =====  
  § Bibliografía  
===== *)
```

```
(*  
+ "More Basic Tactics" de Peirce et als. http://bit.ly/2LYFTlZ *)
```

# Tema 6

## Lógica en Coq

```
(* T6: Lógica en Coq *)
```

```
Set Warnings "-notation-overridden,-parsing".
```

```
Require Export T5_Tactics.
```

```
(* El contenido del tema es
```

```
1. Introducción
```

```
2. Conectivas lógicas
```

```
1. Conjunción
```

```
2. Disyunción
```

```
3. Falsedad y negación
```

```
4. Verdad
```

```
5. Equivalencia lógica
```

```
6. Cuantificación existencial
```

```
3. Programación con proposiciones
```

```
4. Aplicando teoremas a argumentos
```

```
5. Coq vs. teoría de conjuntos
```

```
1. Extensionalidad funcional
```

```
2. Proposiciones y booleanos
```

```
3. Lógica clásica vs. constructiva
```

```
Bibliografía
```

```
*)
```

```
(* =====  
§ 1. Introducción  
===== *)
```

```
(* -----
Ejemplo 1.1. Calcular el tipo de las siguientes expresiones
  3 = 3.
  3 = 4.
  forall n m : nat, n + m = m + n.
  forall n : nat, n = 2.
----- *)
```

```
Check 3 = 3.
(* ==> Prop *)
```

```
Check 3 = 4.
(* ==> Prop *)
```

```
Check forall n m : nat, n + m = m + n.
(* ==> Prop *)
```

```
Check forall n : nat, n = 2.
(* ==> Prop *)
```

```
(* -----
Nota. El tipo de las fórmulas es Prop.
----- *)
```

```
(* -----
Ejemplo 1.2.3. Demostrar que 2 más dos es 4.
----- *)
```

```
Theorem suma_2_y_2:
  2 + 2 = 4.
```

```
Proof. reflexivity. Qed.
```

```
(* -----
Nota. Usa la proposición '2 + 2 = 4'.
----- *)
```

```
(* -----
Ejemplo 1.2.2. Definir la proposición
  prop_suma: Prop
----- *)
```

*que afirma que la suma de 2 y 2 es 4.*

----- \*)

**Definition** prop\_suma: **Prop** := 2 + 2 = 4.

(\* -----  
*Ejemplo 1.2.3. Calcular el tipo de prop\_suma*  
 ----- \*)

**Check** prop\_suma.

(\* ==> prop\_suma : Prop \*)

(\* -----  
*Ejemplo 1.2.4. Usando prop\_suma, demostrar que la suma de 2 y 2 es 4.*  
 ----- \*)

**Theorem** prop\_suma\_es\_verdadera:

prop\_suma.

**Proof.** reflexivity. **Qed.**

(\* -----  
*Ejemplo 1.3.1. Definir la proposición*  
*es\_tres (n : nat) : Prop*  
*tal que (es\_tres n) se verifica si n es el número 3.*  
 ----- \*)

**Definition** es\_tres (n : **nat**) : **Prop** :=  
 n = 3.

(\* -----  
*Ejemplo 1.3.2. Calcular el tipo de las siguientes expresiones*  
*es\_tres.*  
*es\_tres 3.*  
*es\_tres 5.*  
 ----- \*)

**Check** es\_tres.

(\* ==> nat -> Prop \*)

**Check** es\_tres 3.

```
(* ==> Prop *)
```

**Check** es\_tres 5.

```
(* ==> Prop *)
```

```
(* -----
  Nota. Ejemplo de proposición parametrizada.
  ----- *)
```

```
(* -----
  Ejemplo 1.4.1. Definir la función
    inyectiva {A B : Type} (f : A -> B) : Prop :=
  tal que (inyectiva f) se verifica si f es inyectiva.
  ----- *)
```

**Definition** inyectiva {A B : **Type**} (f : A -> B) : **Prop** :=  
**forall** x y : A, f x = f y -> x = y.

```
(* -----
  Ejemplo 1.4.2. Demostrar que la funcion sucesor es inyectiva; es
  decir,
    inyectiva S.
  ----- *)
```

**Lemma** suc\_iny: inyectiva S.

**Proof.**

```
intros n m H. (* n, m : nat
                H : S n = S m
                =====
                n = m *)
```

```
inversion H. (* n, m : nat
                H : S n = S m
                H1 : n = m
                =====
                m = m *)
```

**reflexivity.**

**Qed.**

```
(* -----
  Ejemplo 1.5. Calcular los tipos de las siguientes expresiones
```

```

    3 = 5.
    eq 3 5.
    eq 3.
    @eq.
    ----- *)

Check (3 = 5).
(* ==> Prop *)

Check (eq 3 5).
(* ==> Prop *)

Check (eq 3).
(* ==> nat -> Prop *)

Check @eq.
(* ==> forall A : Type, A -> A -> Prop *)

(* -----
    Notas.
    1. La expresión (x = y) es una abreviatura de (eq x y).
    2. Se escribe @eq en lugar de eq para ver los argumentos implícitos.
    ----- *)

(* =====
    § 2. Conectivas lógicas
    ===== *)

(* =====
    §§ 2.1. Conjunción
    ===== *)

(* -----
    Ejemplo 2.1.1. Demostrar que
    3 + 4 = 7 /\ 2 * 2 = 4.
    ----- *)

Example ej_conjuncion: 3 + 4 = 7 /\ 2 * 2 = 4.
Proof.
  split.

```

```

- (*
    =====
    3 + 4 = 7 *)
reflexivity.
- (*
    =====
    2 * 2 = 4 *)
reflexivity.
Qed.

(* -----
Notas.
1. El símbolo de conjunción se escribe con /\
2. La táctica 'split' sustituye el objetivo (P /\ Q) por los
subobjetivos P y Q.
----- *)

(* -----
Ejemplo 2.1.2. Demostrar que
  forall A B : Prop, A -> B -> A /\ B.
----- *)

Lemma conj_intro: forall A B : Prop, A -> B -> A /\ B.
Proof.
  intros A B HA HB. (* A, B : Prop
                      HA : A
                      HB : B
                      =====
                      A /\ B *)
split.
- (* A, B : Prop
    HA : A
    HB : B
    =====
    A *)
  apply HA.
- (* A, B : Prop
    HA : A
    HB : B
    =====

```



```

                                B *)
  apply HB.
Qed.

(* -----
   Ejemplo 2.1.3. Demostrar, con con_intro, que
    $3 + 4 = 7 \wedge 2 * 2 = 4$ .
   ----- *)

```

**Example** ej\_conjuncion':  $3 + 4 = 7 \wedge 2 * 2 = 4$ .

**Proof.**

```

  apply conj_intro.
-      (* 3 + 4 = 7 *)
  reflexivity.
-      (* 2 * 2 = 4 *)
  reflexivity.
Qed.

```

```

(* -----
   Ejercicio 2.1.1. Demostrar que
   forall n m : nat, n + m = 0 -> n = 0 /\ m = 0.
   ----- *)

```

**Example** ejercicio\_conj:

**forall** n m : **nat**,  $n + m = 0 \rightarrow n = 0 \wedge m = 0$ .

**Proof.**

```

  intros n m H.
                                (* n, m : nat
                                H : n + m = 0
                                =====
                                n = 0 /\ m = 0 *)

  apply conj_intro.
-
                                (* n, m : nat
                                H : n + m = 0
                                =====
                                n = 0 *)

  destruct n.
+
                                (* m : nat
                                H : 0 + m = 0
                                =====
                                0 = 0 *)

```

```

reflexivity.
+
(* n, m : nat
   H : S n + m = 0
   =====
   S n = 0 *)

simpl in H.
(* n, m : nat
   H : S (n + m) = 0
   =====
   S n = 0 *)

inversion H.
-
(* n, m : nat
   H : n + m = 0
   =====
   m = 0 *)

destruct m.
+
(* n : nat
   H : n + 0 = 0
   =====
   0 = 0 *)

reflexivity.
+
(* n, m : nat
   H : n + S m = 0
   =====
   S m = 0 *)

rewrite suma_conmutativa in H.
(* n, m : nat
   H : S m + n = 0
   =====
   S m = 0 *)

simpl in H.
(* n, m : nat
   H : S (m + n) = 0
   =====
   S m = 0 *)

inversion H.

Qed.

(* -----
   Ejemplo 2.1.4. Demostrar que
   forall n m : nat, n = 0 /\ m = 0 -> n + m = 0.
   ----- *)

```

**Lemma** ej\_conjuncion2 :

**forall** n m : **nat**, n = 0 /\ m = 0 -> n + m = 0.

**Proof.**

```

intros n m H.          (* n, m : nat
                        H : n = 0 /\ m = 0
                        =====
                        n + m = 0 *)

destruct H as [Hn Hm]. (* n, m : nat
                        Hn : n = 0
                        Hm : m = 0
                        =====
                        n + m = 0 *)

rewrite Hn.             (* 0 + m = 0 *)
rewrite Hm.             (* 0 + 0 = 0 *)
reflexivity.

```

**Qed.**

```

(* -----
   Nota. Uso de la táctica 'destruct H as [HA HB]' que sustituye la
   hipótesis H de la forma (A /\ B) por las hipótesis HA (que afirma
   que A es verdad) y HB (que afirma que B es verdad).
   ----- *)

```

```

(* -----
   Ejemplo 2.1.5. Demostrar que
   forall n m : nat, n = 0 /\ m = 0 -> n + m = 0.
   ----- *)

```

**Lemma** ej\_conjuncion2' :

**forall** n m : **nat**, n = 0 /\ m = 0 -> n + m = 0.

**Proof.**

```

intros n m [Hn Hm].    (* n, m : nat
                        Hn : n = 0
                        Hm : m = 0
                        =====
                        n + m = 0 *)

rewrite Hn.             (* 0 + m = 0 *)
rewrite Hm.             (* 0 + 0 = 0 *)
reflexivity.

```

**Qed.**

```
(* -----
Nota. La táctica 'intros x [HA HB]', cuando el objetivo es de la
forma (forall x, A /\ B -> C), introduce la variable x y las
hipótesis HA y HB afirmando la certeza de A y de B, respectivamente.
----- *)
```

```
(* -----
Ejemplo 2.1.6. Demostrar que
forall n m : nat, n = 0 -> m = 0 -> n + m = 0.
----- *)
```

**Lemma** ej\_conjuncion2'' :

**forall** n m : **nat**, n = 0 -> m = 0 -> n + m = 0.

**Proof.**

```
intros n m Hn Hm. (* n, m : nat
                    Hn : n = 0
                    Hm : m = 0
                    =====
                    n + m = 0 *)
rewrite Hn.        (* 0 + m = 0 *)
rewrite Hm.        (* 0 + 0 = 0 *)
reflexivity.
```

**Qed.**

```
(* -----
Ejemplo 2.1.7. Demostrar que
forall n m : nat, n + m = 0 -> n * m = 0.
----- *)
```

**Lemma** ej\_conjuncion3 :

**forall** n m : **nat**, n + m = 0 -> n \* m = 0.

**Proof.**

```
intros n m H.      (* n, m : nat
                    H : n + m = 0
                    =====
                    n * m = 0 *)
assert (H' : n = 0 /\ m = 0).
-                (* n, m : nat
                  H : n + m = 0
```

```

=====
n = 0 /\ m = 0 *)
(* n + m = 0 *)

apply ejercicio_conj.
apply H.
-
(* n, m : nat
   H : n + m = 0
   H' : n = 0 /\ m = 0
=====
   n * m = 0 *)
destruct H' as [Hn Hm].
(* n, m : nat
   H : n + m = 0
   Hn : n = 0
   Hm : m = 0
=====
   n * m = 0 *)
rewrite Hn.
reflexivity.
Qed.

(* -----
   Ejemplo 2.1.8. Demostrar que
     forall P Q : Prop,
       P /\ Q -> P.
   ----- *)

Lemma conj_e1 : forall P Q : Prop,
  P /\ Q -> P.
Proof.
  intros P Q [HP HQ]. (* P, Q : Prop
                        HP : P
                        HQ : Q
=====
                        P *)

  apply HP.
Qed.

(* -----
   Ejercicio 2.1.2. Demostrar que
     forall P Q : Prop,
       P /\ Q -> Q.

```

```

----- *)

Lemma conj_e2: forall P Q : Prop,
  P /\ Q -> Q.
Proof.
  intros P Q [HP HQ]. (* P, Q : Prop
                        HP : P
                        HQ : Q
                        =====
                        Q *)

  apply HQ.
Qed.

(* -----
   Ejemplo 2.1.9. Demostrar que
   forall P Q : Prop,
   P /\ Q -> Q /\ P.
   ----- *)

Theorem conj_conmutativa: forall P Q : Prop,
  P /\ Q -> Q /\ P.
Proof.
  intros P Q [HP HQ]. (* P, Q : Prop
                        HP : P
                        HQ : Q
                        =====
                        Q /\ P *)

  split.
  - (* P, Q : Prop
      HP : P
      HQ : Q
      =====
      Q *)

    apply HQ.
  - (* P, Q : Prop
      HP : P
      HQ : Q
      =====
      P *)

    apply HP.

```

**Qed.**

```
(* -----
  Ejercicio 2.1.3. Demostrar que
    forall P Q R : Prop,
      P /\ (Q /\ R) -> (P /\ Q) /\ R.
  ----- *)
```

**Theorem** conj\_asociativa : **forall** P Q R : **Prop**,  
 P /\ (Q /\ R) -> (P /\ Q) /\ R.

**Proof.**

```
  intros P Q R [HP [HQ HR]]. (* P, Q, R : Prop
                                HP : P
                                HQ : Q
                                HR : R
                                =====
                                (P /\ Q) /\ R *)
```

**split.**

```
- (* P /\ Q *)
```

**split.**

```
+ (* P *)
```

**apply** HP.

```
+ (* Q *)
```

**apply** HQ.

```
- (* R *)
```

**apply** HR.

**Qed.**

```
(* -----
  Nota. Uso de la táctica 'intros P Q R [HP [HQ HR]]'.
  ----- *)
```

```
(* -----
  Ejemplo 2.1.10. Calcular el tipo de la expresión
    and
  ----- *)
```

**Check** and.

```
(* ==> and : Prop -> Prop -> Prop *)
```

```

(* -----
   Nota.  $(x \wedge y)$  es una abreviatura de  $(\text{and } x \ y)$ .
   ----- *)

(* =====
   §§ 2.2. Disyunción
   ===== *)

(* -----
   Ejemplo 2.2.1. Demostrar que
   forall n m : nat, n = 0  $\wedge$  m = 0 -> n * m = 0.
   ----- *)

(* 1ª demostración *)
Lemma disy_ej1:
  forall n m : nat, n = 0  $\wedge$  m = 0 -> n * m = 0.
Proof.
  intros n m H.
  destruct H as [Hn | Hm].
  -
    (* n, m : nat
       Hn : n = 0
       =====
       n * m = 0 *)
    rewrite Hn.
    reflexivity.
  -
    (* n, m : nat
       Hm : m = 0
       =====
       n * m = 0 *)
    rewrite Hm.
    rewrite <- mult_n_0.
    reflexivity.
Qed.

(* 2ª demostración *)
Lemma disy_ej:
  forall n m : nat, n = 0  $\wedge$  m = 0 -> n * m = 0.
Proof.
  intros n m [Hn | Hm].
  -
    (* n, m : nat

```



```

                                Hn : n = 0
                                =====
                                n * m = 0 *)
rewrite Hn.                    (* 0 * m = 0 *)
reflexivity.
-                               (* n, m : nat
                               Hm : m = 0
                               =====
                               n * m = 0 *)
rewrite Hm.                    (* n * 0 = 0 *)
rewrite <- mult_n_0.           (* 0 = 0 *)
reflexivity.
Qed.

(* -----
Notas.
1. La táctica 'destruct H as [Hn | Hm]', cuando la hipótesis H es de
la forma (A \ / B), la divide en dos casos: uno con hipótesis HA
(afirmando la certeza de A) y otro con la hipótesis HB (afirmando
la certeza de B).
2. La táctica 'intros x [HA | HB]', cuando el objetivo es de la
forma (forall x, A \ / B -> C), introduce la variable x y dos casos:
uno con hipótesis HA (afirmando la certeza de A) y otro con la
hipótesis HB (afirmando la certeza de B).
----- *)

(* -----
Ejemplo 2.2.2. Demostrar que
forall A B : Prop, A -> A \ / B.
----- *)

Lemma disy_intro: forall A B : Prop, A -> A \ / B.
Proof.
  intros A B HA. (* A, B : Prop
                  HA : A
                  =====
                  A \ / B *)
  left.          (* A *)
  apply HA.
Qed.

```

```
(* -----
Nota. La táctica 'left' sustituye el objetivo de la forma (A \ / B)
por A.
----- *)
```

```
(* -----
Ejemplo 2.2.3. Demostrar que
  forall n : nat, n = 0 \ / n = S (pred n).
----- *)
```

**Lemma** cero\_o\_sucesor:

**forall** n : **nat**, n = 0 \ / n = S (pred n).

**Proof.**

**intros** [|n].

```
- (*
    =====
    0 = 0 \ / 0 = S (Nat.pred 0) *)
left. (* 0 = 0 *)
reflexivity.
- (* n : nat
    =====
    S n = 0 \ / S n = S (Nat.pred (S n)) *)
right. (* S n = S (Nat.pred (S n)) *)
reflexivity.
```

**Qed.**

```
(* -----
Nota. La táctica 'right' sustituye el objetivo de la forma (A \ / B)
por B.
----- *)
```

```
(* -----
Ejercicio 2.2.1. Demostrar que
  forall n m, n * m = 0 -> n = 0 \ / m = 0.
----- *)
```

**Lemma** mult\_eq\_0 :

**forall** n m, n \* m = 0 -> n = 0 \ / m = 0.

**Proof.**

```

intros n m H.          (* n, m : nat
                        H : n * m = 0
                        =====
                        n = 0 \ / m = 0 *)

destruct n as [|n'|].
-
  (* m : nat
  H : 0 * m = 0
  =====
  0 = 0 \ / m = 0 *)

  left.
  reflexivity.
-
  (* n', m : nat
  H : S n' * m = 0
  =====
  S n' = 0 \ / m = 0 *)

  destruct m as [|m'|].
+
  (* n' : nat
  H : S n' * 0 = 0
  =====
  S n' = 0 \ / 0 = 0 *)

  right.
  reflexivity.
+
  (* n', m' : nat
  H : S n' * S m' = 0
  =====
  S n' = 0 \ / S m' = 0 *)

  simpl in H.
  (* n', m' : nat
  H : S (m' + n' * S m') = 0
  =====
  S n' = 0 \ / S m' = 0 *)

  inversion H.
Qed.

(* -----
   Ejercicio 2.2.2. Demostrar que
   forall P Q : Prop,
     P \ / Q -> Q \ / P.
   ----- *)

```

**Theorem** disy\_commutativa: **forall** P Q : **Prop**,

$P \setminus / Q \rightarrow Q \setminus / P.$

**Proof.**

**intros** P Q [HP | HQ].

```
- (* P, Q : Prop
   HP : P
   =====
   Q \ / P *)
right. (* P *)
```

**apply** HP.

```
- (* P, Q : Prop
   HQ : Q
   =====
   Q \ / P *)
left. (* Q *)
```

**apply** HQ.

**Qed.**

```
(* -----
   Ejemplo 2.2.4. Calcular el tipo de la expresión
   or
   ----- *)
```

**Check** or.

```
(* ==> or : Prop -> Prop -> Prop *)
```

```
(* -----
   Nota. (x \ / y) es una abreviatura de (or x y).
   ----- *)
```

```
(* =====
   §§ 2.3. Falsedad y negación
   ===== *)
```

**Module** DefNot.

```
(* -----
   Ejemplo 2.3.1. Definir la función
   not (P : Prop) : Prop
   tal que (not P) es la negación de P
   ----- *)
```

**Definition** not (P:Prop) : Prop :=  
 P -> False.

(\* -----  
 Ejemplo 2.3.2. Definir ( $\sim x$ ) como abreviatura de (not x).  
 ----- \*)

**Notation** " $\sim x$ " := (not x) : type\_scope.

(\* -----  
 Nota. Esta es la forma como está definida la negación en Coq.  
 ----- \*)

**End** DefNot.

(\* -----  
 Ejemplo 2.3.3. Demostrar que  
 forall (P:Prop),  
 False -> P.  
 ----- \*)

**Theorem** ex\_falso\_quodlibet: forall (P:Prop),  
 False -> P.

**Proof.**

intros P H. (\* P : Prop  
 H : False  
 =====  
 P \*)

destruct H.

**Qed.**

(\* -----  
 Nota. En latín, "ex falso quodlibet" significa "de lo falso (se sigue) cualquier cosa".  
 ----- \*)

(\* -----  
 Ejercicio 2.3.1. Demostrar que  
 forall (P:Prop),  
 P -> P.  
 ----- \*)

```

~ P -> (forall (Q:Prop), P -> Q).
----- *)

Fact negacion_elim: forall (P:Prop),
  ~ P -> (forall (Q:Prop), P -> Q).
Proof.
  unfold not.      (*)
                    =====
                    forall P : Prop, (P -> False) -> forall Q : Prop, P -> Q *)
  intros P H1.      (* P : Prop
                    H1 : P -> False
                    =====
                    forall Q : Prop, P -> Q *)
  intros Q H2.      (* P : Prop
                    H1 : P -> False
                    Q : Prop
                    H2 : P
                    =====
                    Q *)
  apply H1 in H2. (* P : Prop
                    H1 : P -> False
                    Q : Prop
                    H2 : False
                    =====
                    Q *)

  destruct H2.
Qed.

(* -----
  Ejemplo 2.3.4. Demostrar que
  ~(0 = 1).
  ----- *)

Theorem cero_no_es_uno: ~(0 = 1).
Proof.
  intros H.        (* H : 0 = 1
                    =====
                    False *)

  inversion H.
Qed.

```

```
(* -----
  Nota. La expresión (x <> y) es una abreviatura de ~(x = y).
  ----- *)
```

**Theorem** cero\_no\_es\_uno' : 0 <> 1.

**Proof.**

```
  intros H.      (* H : 0 = 1
                  =====
                  False *)
```

inversion H.

**Qed.**

```
(* -----
  Ejemplo 2.3.5. Demostrar que
    ~ False
  ----- *)
```

**Theorem** not\_False :

~ False.

**Proof.**

```
  unfold not. (*
              =====
              False -> False *)
  intros H.   (* H : False
              =====
              False *)
```

destruct H.

**Qed.**

```
(* -----
  Ejemplo 2.3.6. Demostrar que
    forall P Q : Prop,
      (P /\ ~P) -> Q.
  ----- *)
```

**Theorem** contradiccion\_implica\_cualquiera: forall P Q : Prop,

(P /\ ~P) -> Q.

**Proof.**

```

intros P Q [HP HNP]. (* P, Q : Prop
                        HP : P
                        HNP : ~ P
                        =====
                        Q *)
unfold not in HNP. (* P, Q : Prop
                      HP : P
                      HNP : P -> False
                      =====
                      Q *)
apply HNP in HP. (* P, Q : Prop
                   HP : False
                   HNP : P -> False
                   =====
                   Q *)

destruct HP.
Qed.

(* -----
   Ejemplo 2.3.7. Demostrar que
   forall P : Prop,
     P -> ~~P.
   ----- *)

Theorem doble_neg: forall P : Prop,
  P -> ~~P.
Proof.
  intros P H. (* P : Prop
                H : P
                =====
                ~ ~ P *)
  unfold not. (* (P -> False) -> False *)
  intros G.   (* P : Prop
                H : P
                G : P -> False
                =====
                False *)
  apply G.    (* P *)
  apply H.
Qed.

```



```
(* -----
Ejercicio 2.3.2. Demostrar que
  forall (P Q : Prop),
    (P -> Q) -> (~Q -> ~P).
----- *)
```

**Theorem** contrapositiva: **forall** (P Q : **Prop**),  
 (P -> Q) -> (~Q -> ~P).

**Proof.**

```
unfold not. (*
=====
  forall P Q : Prop,
    (P -> Q) -> (Q -> False) -> P -> False *)
intros P Q H1 H2 H3. (* P, Q : Prop
  H1 : P -> Q
  H2 : Q -> False
  H3 : P
=====
  False *)
apply H1 in H3. (* P, Q : Prop
  H1 : P -> Q
  H2 : Q -> False
  H3 : Q
=====
  False *)
apply H2 in H3. (* P, Q : Prop
  H1 : P -> Q
  H2 : Q -> False
  H3 : False
=====
  False *)
apply H3.
Qed.
```

```
(* -----
Ejercicio 2.3.3. Demostrar que
  forall P : Prop,
    ~ (P /\ ~P).
----- *)
```

**Theorem** no\_contradiccion: **forall** P : **Prop**,  
 $\sim (P \wedge \sim P)$ .

**Proof.**

```

unfold not.          (*
=====
      forall P : Prop, P /\ (P -> False) -> False *)
intros P [H1 H2]. (* P : Prop
      H1 : P
      H2 : P -> False
=====
      False *)
apply H2.            (* P *)
apply H1.

```

**Qed.**

```

(* -----
   Ejemplo 2.3.8. Demostrar que
     forall b : bool,
       b <> true -> b = false.
   ----- *)

```

(\* 1ª demostración \*)

**Theorem** no\_verdadero\_es\_falso: **forall** b : **bool**,  
 $b \neq \text{true} \rightarrow b = \text{false}$ .

**Proof.**

```

intros [] H.
-
      (* H : true <> true
      =====
      true = false *)
unfold not in H.      (* H : true = true -> False
      =====
      true = false *)
apply ex_falso_quodlibet. (* H : true = true -> False
      =====
      False *)
apply H.              (* true = true *)
reflexivity.
-
      (* H : false <> true
      =====

```

```

false = false *)

reflexivity.
Qed.

(* 2ª demostración *)
Theorem no_verdadero_es_falso': forall b : bool,
  b <> true -> b = false.
Proof.
  intros [] H.
  -
    (* H : true <> true
    =====
    true = false *)
    unfold not in H. (* H : true = true -> False
    =====
    true = false *)
    exfalso. (* H : true = true -> False
    =====
    False *)
    apply H. (* true = true *)
    reflexivity.
  -
    (* H : false <> true
    =====
    false = false *)

    reflexivity.
Qed.

(* -----
Notas.
1. Uso de 'apply ex_falso_quodlibet' en la primera demostración.
2. Uso de 'exfalso' en la segunda demostración.
3. La táctica 'exfalso' sustituye el objetivo por falso.
----- *)

(* =====
§§ 2.4. Verdad
===== *)

(* -----
Ejemplo 2.4.1. Demostrar que la proposición True es verdadera.
----- *)

```

**Lemma** True\_es\_verdadera : True.

**Proof.**

**apply** I.

**Qed.**

```
(* -----
   Nota. Uso del constructor I.
   ----- *)

(* =====
   §§ 2.5. Equivalencia lógica
   ===== *)
```

**Module** DefIff.

```
(* -----
   Ejemplo 2.5.1. Definir la función
       iff (P Q : Prop) : Prop
   tal que (iff P Q) es la equivalencia de P y Q.
   ----- *)
```

**Definition** iff (P Q : **Prop**) : **Prop** := (P -> Q) /\ (Q -> P).

```
(* -----
   Ejemplo 2.5.2. Definir (P <-> Q) como una abreviatura de (iff P Q).
   ----- *)
```

**Notation** "P <-> Q" := (iff P Q)  
                           (at level 95, no associativity)  
                           : type\_scope.

**End** DefIff.

```
(* -----
   Ejemplo 2.5.3. Demostrar que
       forall P Q : Prop,
         (P <-> Q) -> (Q <-> P).
   ----- *)
```

**Theorem** iff\_sim : forall P Q : Prop,  
 (P <-> Q) -> (Q <-> P).

**Proof.**

```

intros P Q [HPQ HQP]. (* P, Q : Prop
                        HPQ : P -> Q
                        HQP : Q -> P
                        =====
                        Q <-> P *)

split.
-
  (* P, Q : Prop
  HPQ : P -> Q
  HQP : Q -> P
  =====
  Q -> P *)

apply HQP.
-
  (* P, Q : Prop
  HPQ : P -> Q
  HQP : Q -> P
  =====
  P -> Q *)

```

**apply** HPQ.

**Qed.**

```

(* -----
  Ejemplo 2.5.4. Demostrar que
    forall b : bool,
      b <> true <-> b = false.
  ----- *)

```

**Lemma** not\_true\_iff\_false : forall b : bool,  
 b <> true <-> b = false.

**Proof.**

```

intros b.
(* b : bool
=====
b <> true <-> b = false *)

split.
-
  (* b : bool
  =====
  b <> true -> b = false *)

```

```

apply no_verdadero_es_falso.
-
      (* b : bool
      =====
      b = false -> b <> true *)

intros H.
      (* b : bool
      H : b = false
      =====
      b <> true *)

rewrite H.
intros H'.
      (* false <> true *)
      (* b : bool
      H : b = false
      H' : false = true
      =====
      False *)

inversion H'.
Qed.

(* -----
   Ejercicio 2.3.4. Demostrar que
   forall P : Prop,
     P <-> P.
   ----- *)

Lemma iff_refl_aux: forall P : Prop,
  P -> P.
Proof.
  intros P H. (* P : Prop
                H : P
                =====
                P *)

  apply H.
Qed.

Theorem iff_refl: forall P : Prop,
  P <-> P.
Proof.
  split.
-
      (* P : Prop
      =====

```

```

                                P -> P *)
  apply iff_refl_aux.
-
                                (* P : Prop
                                =====
                                P -> P *)

  apply iff_refl_aux.
Qed.

(* -----
  Ejercicio 2.3.5. Demostrar que
    forall P Q R : Prop,
      (P <-> Q) -> (Q <-> R) -> (P <-> R).
  ----- *)

Theorem iff_trans: forall P Q R : Prop,
  (P <-> Q) -> (Q <-> R) -> (P <-> R).
Proof.
  intros P Q R [HPQ HQP] [HQR HRQ]. (* P, Q, R : Prop
                                     HPQ : P -> Q
                                     HQP : Q -> P
                                     HQR : Q -> R
                                     HRQ : R -> Q
                                     =====
                                     P <-> R *)

  split.
-
                                (* P, Q, R : Prop
                                HPQ : P -> Q
                                HQP : Q -> P
                                HQR : Q -> R
                                HRQ : R -> Q
                                =====
                                P -> R *)

  intros HP.
                                (* P, Q, R : Prop
                                HPQ : P -> Q
                                HQP : Q -> P
                                HQR : Q -> R
                                HRQ : R -> Q
                                HP : P
                                =====
                                R *)

```

```

apply HQR.
apply HPQ.
apply HP.
-
(* P, Q, R : Prop
   HPQ : P -> Q
   HQP : Q -> P
   HQR : Q -> R
   HRQ : R -> Q
   =====
   R -> P *)

intros HR.
(* P, Q, R : Prop
   HPQ : P -> Q
   HQP : Q -> P
   HQR : Q -> R
   HRQ : R -> Q
   HR : R
   =====
   P *)

apply HQP.
apply HRQ.
apply HR.
Qed.

(* -----
   Ejercicio 2.3.6. Demostrar que
   forall P Q R : Prop,
     P /\ (Q /\ R) <=> (P /\ Q) /\ (P /\ R).
   ----- *)

Theorem distributiva_disy_conj: forall P Q R : Prop,
  P /\ (Q /\ R) <=> (P /\ Q) /\ (P /\ R).
Proof.
  split.
-
(* P, Q, R : Prop
   =====
   P /\ (Q /\ R) -> (P /\ Q) /\ (P /\ R) *)

intros [HP | [HQ HR]].
+
(* P, Q, R : Prop
   HP : P
   =====

```



```

                                (P \ / Q) /\ (P \ / R) *)

split.
*
                                (* P, Q, R : Prop
                                HP : P
                                =====
                                P \ / Q *)
                                (* P *)

left.
apply HP.
*
                                (* P, Q, R : Prop
                                HP : P
                                =====
                                P \ / R *)
                                (* P *)

left.
apply HP.
+
                                (* P, Q, R : Prop
                                HQ : Q
                                HR : R
                                =====
                                (P \ / Q) /\ (P \ / R) *)

split.
*
                                (* P, Q, R : Prop
                                HQ : Q
                                HR : R
                                =====
                                P \ / Q *)
                                (* Q *)

right.
apply HQ.
*
                                (* P, Q, R : Prop
                                HQ : Q
                                HR : R
                                =====
                                P \ / R *)
                                (* R *)

right.
apply HR.
-
                                (* P, Q, R : Prop
                                =====
                                (P \ / Q) /\ (P \ / R) -> P \ / (Q /\ R) *)

intros [[HP1|HQ] [HP2|HR]].
+
                                (* P, Q, R : Prop
                                HP1, HP2 : P

```

```

=====
P \ / (Q /\ R) *)
(* P *)

left.
apply HP1.
+
=====
P \ / (Q /\ R) *)
(* P *)

left.
apply HP1.
+
=====
P \ / (Q /\ R) *)
(* P *)

left.
apply HP2.
+
=====
P \ / (Q /\ R) *)
(* Q /\ R *)

right.
split.
*
=====
Q *)

apply HQ.
*
=====
R *)

apply HR.
Qed.

(* -----

```

*Nota. Se importa la librería `Coq.Setoids.Setoid` para usar las tácticas `reflexivity` y `rewrite` con `iff`.*

----- \*)

**Require Import** Coq.Setoids.Setoid.

(\* -----  
*Ejemplo 2.5.5. Demostrar que*  
 $\text{forall } n \ m : \text{nat}, n * m = 0 \leftrightarrow n = 0 \ \vee \ m = 0.$   
 ----- \*)

**Lemma** mult\_0 : **forall** n m : **nat**,  $n * m = 0 \leftrightarrow n = 0 \ \vee \ m = 0.$

**Proof.**

**split.**

- (\* n, m : nat  
 =====  
 $n * m = 0 \rightarrow n = 0 \ \vee \ m = 0$  \*)

**apply** mult\_eq\_0.

- (\* n, m : nat  
 =====  
 $n = 0 \ \vee \ m = 0 \rightarrow n * m = 0$  \*)

**apply** disy\_ej.

**Qed.**

(\* -----  
*Ejemplo 2.5.6. Demostrar que*  
 $\text{forall } P \ Q \ R : \text{Prop},$   
 $P \ \vee \ (Q \ \vee \ R) \leftrightarrow (P \ \vee \ Q) \ \vee \ R.$   
 ----- \*)

**Lemma** disy\_asociativa :

**forall** P Q R : **Prop**,  $P \ \vee \ (Q \ \vee \ R) \leftrightarrow (P \ \vee \ Q) \ \vee \ R.$

**Proof.**

**intros** P Q R. (\* P, Q, R : Prop  
 =====  
 $P \ \vee \ (Q \ \vee \ R) \leftrightarrow (P \ \vee \ Q) \ \vee \ R$  \*)

**split.**

- (\* P, Q, R : Prop  
 =====  
 $P \ \vee \ (Q \ \vee \ R) \rightarrow (P \ \vee \ Q) \ \vee \ R$  \*)

```

intros [H | [H | H]].
+
  (* P, Q, R : Prop
    H : P
    =====
    (P \ / Q) \ / R *)

  left.
  (* P \ / Q *)
  left.
  (* P *)
  apply H.
+
  (* P, Q, R : Prop
    H : Q
    =====
    (P \ / Q) \ / R *)

  left.
  (* P \ / Q *)
  right.
  (* Q *)
  apply H.
+
  (* P, Q, R : Prop
    H : R
    =====
    (P \ / Q) \ / R *)

  right.
  (* R *)
  apply H.
-
  (* P, Q, R : Prop
    =====
    (P \ / Q) \ / R -> P \ / (Q \ / R) *)

intros [[H | H] | H].
+
  (* P, Q, R : Prop
    H : P
    =====
    (P \ / Q) \ / R *)

  left.
  (* P *)
  apply H.
+
  (* P, Q, R : Prop
    H : Q
    =====
    P \ / (Q \ / R) *)

  right.
  (* Q \ / R *)
  left.
  (* Q *)
  apply H.
+
  (* P, Q, R : Prop
    H : R
  
```

```

=====
      P \/\ (Q \/\ R) *)
right.      (* Q \/\ R *)
right.      (* R *)
apply H.

Qed.

(* -----
   Ejemplo 2.5.7. Demostrar que
   forall n m p : nat,
     n * m * p = 0 <-> n = 0 \/\ m = 0 \/\ p = 0.
   ----- *)

Lemma mult_0_3: forall n m p : nat,
  n * m * p = 0 <-> n = 0 \/\ m = 0 \/\ p = 0.
Proof.
  intros n m p.      (* n, m, p : nat
                       =====
                       n * (m * p) = 0 <-> n = 0 \/\ (m = 0 \/\ p = 0) *)
  rewrite mult_0.      (* n * m = 0 \/\ p = 0 <->
                       n = 0 \/\ (m = 0 \/\ p = 0) *)
  rewrite mult_0.      (* (n = 0 \/\ m = 0) \/\ p = 0 <->
                       n = 0 \/\ (m = 0 \/\ p = 0) *)
  rewrite disy_asociativa. (* (n = 0 \/\ m = 0) \/\ p = 0 <->
                             (n = 0 \/\ m = 0) \/\ p = 0 *)
  reflexivity.
Qed.

(* -----
   Nota. Uso de reflexivity y rewrite con iff.
   ----- *)

(* -----
   Ejemplo 2.5.8. Demostrar que
   forall n m : nat,
     n * m = 0 -> n = 0 \/\ m = 0.
   ----- *)

Lemma ej_apply_iff: forall n m : nat,
  n * m = 0 -> n = 0 \/\ m = 0.

```

**Proof.**

```

intros n m H. (* n, m : nat
                  H : n * m = 0
                  =====
                  n = 0 \ / m = 0 *)
apply mult_0. (* n * m = 0 *)
apply H.

```

**Qed.**

```

(* -----
   Nota. Uso de apply sobre iff.
   ----- *)

```

```

(* =====
   §§ 2.6. Cuantificación existencial
   ===== *)

```

```

(* -----
   Ejemplo 2.6.1. Demostrar que
   exists n : nat, 4 = n + n.
   ----- *)

```

**Lemma** cuatro\_es\_par: **exists** n : **nat**, 4 = n + n.

**Proof.**

```

exists 2.      (*
                  =====
                  4 = 2 + 2 *)

```

**reflexivity.**

**Qed.**

```

(* -----
   Nota. La táctica 'exists a' sustituye el objetivo de la forma
   (exists x, P(x)) por P(a).
   ----- *)

```

```

(** Conversely, if we have an existential hypothesis [exists x, P] in
    the context, we can destruct it to obtain a witness [x] and a
    hypothesis stating that [P] holds of [x]. *)

```

```

(* -----

```

*Ejemplo 2.6.2. Demostrar que*

*forall n : nat,  
(exists m, n = 4 + m) -> (exists o, n = 2 + o).*

----- \*)

*(\* 1ª demostración \*)*

**Theorem** ej\_existe\_2a: **forall** n : **nat**,  
  (**exists** m, n = 4 + m) ->  
  (**exists** o, n = 2 + o).

**Proof.**

**intros** n H.  
  **destruct** H **as** [a Ha].  
  **exists** (2 + a).  
  **apply** Ha.

**Qed.**

*(\* 2ª demostración \*)*

**Theorem** ej\_existe\_2b: **forall** n : **nat**,  
  (**exists** m, n = 4 + m) ->  
  (**exists** o, n = 2 + o).

**Proof.**

**intros** n [a Ha].  
  **exists** (2 + a).  
  **apply** Ha.

**Qed.**

*(\* -----*

*Notas.*

1. 'destruct H [a Ha]' sustituye la hipótesis (H : exists x, P(x)) por (Ha : P(a)).
2. 'intros x [a Ha]' sustituye el objetivo (forall x, (exists y P(y)) -> Q(x)) por Q(x) y le añade la hipótesis (Ha : P(a)).

----- \*)

*(\* -----*

*Ejercicio 2.6.1. Demostrar que*

*forall (X:Type) (P : X -> Prop),  
(forall x, P x) -> ~ (exists x, ~ P x)*

----- \*)

**Theorem** paraTodo\_no\_existe\_no: **forall** (X:Type) (P : X -> Prop),  
 (**forall** x, P x) -> ~ (**exists** x, ~ P x).

**Proof.**

```

intros X P H1 [a Ha]. (* X : Type
                        P : X -> Prop
                        H1 : forall x : X, P x
                        a : X
                        Ha : ~ P a
                        =====
                        False *)
apply Ha.              (* P a *)
apply H1.

```

**Qed.**

```

(* -----
   Ejercicio 2.6.2. Demostrar que
   forall (X : Type) (P Q : X -> Prop),
   (exists x, P x /\ Q x) <-> (exists x, P x) /\ (exists x, Q x).
   ----- *)

```

**Theorem** dist\_existe: **forall** (X : Type) (P Q : X -> Prop),  
 (**exists** x, P x /\ Q x) <-> (**exists** x, P x) /\ (**exists** x, Q x).

**Proof.**

```

intros X P Q.          (* X : Type
                        P, Q : X -> Prop
                        =====
                        (exists x:X, P x /\ Q x) <->
                        (exists x:X, P x) /\ (exists x:X, Q x) *)

```

**split.**

```

- (* X : Type
   P, Q : X -> Prop
   =====
   (exists x : X, P x /\ Q x) ->
   (exists x:X, P x) /\ (exists x:X, Q x) *)

```

**intros** [a [HPa | HQa]].

+

```

(* X : Type
   P, Q : X -> Prop
   a : X
   HPa : P a

```



```

=====
(exists x:X, P x) \/\ (exists x:X, Q x) *)
(* exists x : X, P x *)
  (* P a *)

left.
exists a.
apply HPa.
+
(* X : Type
  P, Q : X -> Prop
  a : X
  HQa : Q a
=====
(exists x:X, P x) \/\ (exists x:X, Q x) *)
(* exists x : X, Q x *)
  (* Q a *)

right.
exists a.
apply HQa.
-
(* X : Type
  P, Q : X -> Prop
=====
(exists x:X, P x) \/\ (exists x:X, Q x) ->
exists x : X, P x \/\ Q x *)

intros [[a HPa] | [a HQa]].
+
(* X : Type
  P, Q : X -> Prop
  a : X
  HPa : P a
=====
exists x : X, P x \/\ Q x *)
  (* P a \/\ Q a *)

exists a.
left.
apply HPa.
+
(* X : Type
  P, Q : X -> Prop
  a : X
  HQa : Q a
=====
exists x : X, P x \/\ Q x *)
  (* P a \/\ Q a *)

exists a.
right.
apply HQa.
Qed.

```

```
(* =====
§ 3. Programación con proposiciones
===== *)
```

```
(* -----
Ejemplo 3.1.1. Definir la función
  En {A : Type} (x : A) (xs : list A) : Prop :=
  tal que (En x xs) se verifica si x pertenece a xs.
----- *)
```

```
Fixpoint En {A : Type} (x : A) (xs : list A) : Prop :=
match xs with
| []      => False
| x' :: xs' => x' = x \ / En x xs'
end.
```

```
(* -----
Ejemplo 3.1.2. Demostrar que
  En 4 [1; 2; 3; 4; 5].
----- *)
```

**Example** En\_ejemplo\_1 : En 4 [1; 2; 3; 4; 5].

**Proof**.

```
simpl.      (* 1 = 4 \ / 2 = 4 \ / 3 = 4 \ / 4 = 4 \ / 5 = 4 \ / False *)
right.     (* 2 = 4 \ / 3 = 4 \ / 4 = 4 \ / 5 = 4 \ / False *)
right.     (* 3 = 4 \ / 4 = 4 \ / 5 = 4 \ / False *)
right.     (* 4 = 4 \ / 5 = 4 \ / False *)
left.      (* 4 = 4 *)
reflexivity.
```

**Qed**.

```
(* -----
Ejemplo 3.1.3. Demostrar que
  forall n : nat,
    En n [2; 4] -> exists n', n = 2 * n'.
----- *)
```

**Example** En\_ejemplo\_2: **forall** n : **nat**,  
 En n [2; 4] -> **exists** n', n = 2 \* n'.

**Proof**.

```

simpl.                                (*
=====
forall n : nat,
  2 = n \ / 4 = n \ / False ->
  exists n' : nat, n = n' + (n' + 0) *)

intros n [H | [H | []]].
-
  (* n : nat
     H : 2 = n
=====
     exists n' : nat, n = n' + (n' + 0) *)
  (* n = 1 + (1 + 0) *)
  exists 1.
  rewrite <- H.
  reflexivity.
-
  (* n : nat
     H : 4 = n
=====
     exists n' : nat, n = n' + (n' + 0) *)
  (* n = 2 + (2 + 0) *)
  exists 2.
  rewrite <- H.
  reflexivity.

Qed.

(* -----
   Nota. Uso del patrón vacío para descartar el último caso.
   ----- *)

(* -----
   Ejemplo 3.2. Demostrar que
   forall (A B : Type) (f : A -> B) (xs : list A) (x : A),
     En x xs ->
     En (f x) (map f xs).
   ----- *)

Lemma En_map: forall (A B : Type) (f : A -> B) (xs : list A) (x : A),
  En x xs ->
  En (f x) (map f xs).

Proof.
intros A B f xs x.                (* A : Type
                                     B : Type

```

```

      f : A -> B
      xs : list A
      x : A
      =====
      En x xs -> En (f x) (map f xs) *)

induction xs as [|x' xs' HI|.
-
      (* A : Type
      B : Type
      f : A -> B
      x : A
      =====
      En x [ ] -> En (f x) (map f [ ]) *)
      (* False -> False *)

      simpl.
      intros [].

-
      (* A : Type
      B : Type
      f : A -> B
      x' : A
      xs' : list A
      x : A
      HI : En x xs' -> En (f x) (map f xs')
      =====
      En x (x'::xs') ->
      En (f x) (map f (x'::xs')) *)
      (* x' = x \ / En x xs' ->
      f x' = f x \ / En (f x) (map f xs') *)

      simpl.

      intros [H | H].
      +
      (* A : Type
      B : Type
      f : A -> B
      x' : A
      xs' : list A
      x : A
      HI : En x xs' -> En (f x) (map f xs')
      H : x' = x
      =====
      f x' = f x \ / En (f x) (map f xs') *)
      (* f x = f x \ / En (f x) (map f xs') *)
      (* f x = f x *)

      rewrite H.
      left.
      reflexivity.

```

```

+
(* A : Type
   B : Type
   f : A -> B
   x' : A
   xs' : list A
   x : A
   HI : En x xs' -> En (f x) (map f xs')
   H : En x xs'
   =====
   f x' = f x /\ En (f x) (map f xs') *)
right.
apply HI.
apply H.
Qed.

(* -----
   Ejercicio 3.1. Demostrar que
   forall (A B : Type) (f : A -> B) (xs : list A) (y : B),
     En y (map f xs) <->
     exists x, f x = y /\ En x xs.
   ----- *)

Lemma En_map_iff: forall (A B : Type) (f : A -> B) (xs : list A) (y : B),
  En y (map f xs) <->
  exists x, f x = y /\ En x xs.
Proof.
  intros A B f xs y.
  (* A : Type
     B : Type
     f : A -> B
     xs : list A
     y : B
     =====
     En y (map f xs) <->
     (exists x : A, f x = y /\ En x xs) *)

  induction xs as [|x xs' HI].
  -
  (* A : Type
     B : Type
     f : A -> B
     y : B
     =====

```

**simpl.**

**split.**

+

**intros [].**

+

**intros [a [H []]].**

-

**simpl.**

**split.**

+

```

      En y (map f [ ]) <->
      (exists x : A, f x = y /\ En x [ ]) *)
(* En y (map f [ ]) <->
      (exists x : A, f x = y /\ En x [ ]) *)

(* A : Type
   B : Type
   f : A -> B
   y : B
   =====
   False ->
   exists x : A, f x = y /\ False *)

(* A : Type
   B : Type
   f : A -> B
   y : B
   =====
   (exists x : A, f x = y /\ False) ->
   False *)

(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
        (exists x:A, f x = y /\ En x xs')
   =====
   En y (map f (x :: xs')) <->
   (exists x0 : A,
    f x0 = y /\ En x0 (x :: xs')) *)
(* f x = y /\ En y (map f xs') <->
   (exists x0 : A,
    f x0 = y /\ (x = x0 /\ En x0 xs')) *)

(* A : Type
   B : Type
   f : A -> B

```

```

x : A
xs' : list A
y : B
HI : En y (map f xs') <->
      (exists x:A, f x = y /\ En x xs')
=====
f x = y /\ En y (map f xs') ->
exists x0 : A,
  f x0 = y /\ (x = x0 /\ En x0 xs') *)

intros [H1 | H2].
*

(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
         (exists x:A, f x = y /\ En x xs')
   H1 : f x = y
   =====
   exists x0 : A,
     f x0 = y /\ (x = x0 /\ En x0 xs') *)
(* f x = y /\ (x = x /\ En x xs') *)

exists x.
split.
--

(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
         (exists x:A, f x = y /\ En x xs')
   H1 : f x = y
   =====
   f x = y *)

apply H1.
--

```

```

xs' : list A
y : B
HI : En y (map f xs') <->
      (exists x:A, f x = y /\ En x xs')
H1 : f x = y
=====
x = x /\ En x xs' *)
(* x = x *)

left.
reflexivity.
*

(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
         (exists x:A, f x = y /\ En x xs')
   H2 : En y (map f xs')
   =====
   exists x0 : A,
     f x0 = y /\ (x = x0 /\ En x0 xs') *)
(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
         (exists x:A, f x = y /\ En x xs')
   H2 : exists x : A, f x = y /\ En x xs'
   =====
   exists x0 : A,
     f x0 = y /\ (x = x0 /\ En x0 xs') *)
destruct H2 as [a [Ha1 Ha2]]. (* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->

```



```

                                (exists x:A, f x = y /\ En x xs')
a : A
Ha1 : f a = y
Ha2 : En a xs'
=====
exists x0 : A,
  f x0 = y /\ (x = x0 \/ En x0 xs') *)

exists a.

split.
--
(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
       (exists x:A, f x = y /\ En x xs')
   a : A
   Ha1 : f a = y
   Ha2 : En a xs'
   =====
   f a = y *)

apply Ha1.
--
(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
       (exists x:A, f x = y /\ En x xs')
   a : A
   Ha1 : f a = y
   Ha2 : En a xs'
   =====
   x = a \/ En a xs' *)

right.
apply Ha2.

```

+

```
intros [a [Ha1 [Ha2 | Ha3]]].
```

```
*
```

```
left.
rewrite Ha2.
rewrite Ha1.
reflexivity.
```

```
*
```

```
(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
        (exists x : A,
         f x = y /\ En x xs')
   =====
   (exists x0 : A,
    f x0 = y /\ (x = x0 /\ En x0 xs')) ->
   f x = y /\ En y (map f xs') *)
```

```
(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
        (exists x:A, f x = y /\ En x xs')
   a : A
   Ha1 : f a = y
   Ha2 : x = a
   =====
   f x = y /\ En y (map f xs') *)
```

```
(* f x = y *)
```

```
(* f a = y *)
```

```
(* y = y *)
```

```
(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
        (exists x:A, f x = y /\ En x xs')
   a : A
```

```

    right.
    apply HI.
    exists a.
    split.
    --
    --
    apply Ha1.
    --
    --
    apply Ha3.
    Qed.

```

```

Ha1 : f a = y
Ha3 : En a xs'
=====
f x = y /\ En y (map f xs') *)
(* En y (map f xs') *)
(* exists x0 : A, f x0 = y /\ En x0 xs' *)
    (* f a = y /\ En a xs' *)

(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
        (exists x:A, f x = y /\ En x xs')
   a : A
   Ha1 : f a = y
   Ha3 : En a xs'
   =====
   f a = y *)

(* A : Type
   B : Type
   f : A -> B
   x : A
   xs' : list A
   y : B
   HI : En y (map f xs') <->
        (exists x:A, f x = y /\ En x xs')
   a : A
   Ha1 : f a = y
   Ha3 : En a xs'
   =====
   En a xs' *)

```

```

(* -----
   Ejercicio 3.2. Demostrar que

```

```

forall A (xs ys : list A) (a : A),
  En a (xs ++ ys) <-> En a xs \/ En a ys.
----- *)

```

**Lemma** En\_conc\_1: **forall** A (xs ys : **list** A) (a : A),  
 En a (xs ++ ys) -> En a xs \/ En a ys.

**Proof.**

**induction** xs **as** [|x xs' HI].

-

**simpl.**

**intros** ys a H.

**right.**

**apply** H.

-

**simpl.**

**intros** ys a [H1 | H2].

+

(\* A : Type

=====

forall (ys : list A) (a : A),

En a ([ ] ++ ys) -> En a [ ] \/ En a ys \*)

(\* forall (ys : list A) (a : A),

En a ys -> False \/ En a ys \*)

(\* A : Type

ys : list A

a : A

H : En a ys

=====

False \/ En a ys \*)

(\* En a ys \*)

(\* A : Type

x : A

xs' : list A

HI : forall (ys : list A) (a : A),

En a (xs' ++ ys) ->

En a xs' \/ En a ys

=====

forall (ys : list A) (a : A),

En a ((x :: xs') ++ ys) ->

En a (x :: xs') \/ En a ys \*)

(\* forall (ys : list A) (a : A),

x = a \/ En a (xs' ++ ys) ->

(x = a \/ En a xs') \/ En a ys \*)

(\* A : Type

x : A

xs' : list A

HI : forall (ys : list A) (a : A),

```

                                En a (xs' ++ ys) ->
                                En a xs' \/\ En a ys
ys : list A
a : A
H1 : x = a
=====
(x = a \/\ En a xs') \/\ En a ys *)
left.
left.
apply H1.
+
(* A : Type
x : A
xs' : list A
HI : forall (ys : list A) (a : A),
    En a (xs' ++ ys) ->
    En a xs' \/\ En a ys
ys : list A
a : A
H2 : En a (xs' ++ ys)
=====
(x = a \/\ En a xs') \/\ En a ys *)
rewrite <- disy_asociativa. (* x = a \/\ (En a xs' \/\ En a ys) *)
right. (* En a xs' \/\ En a ys *)
apply HI. (* En a (xs' ++ ys) *)
apply H2.

Qed.

Lemma En_conc_2: forall A (xs ys : list A) (a : A),
  En a xs \/\ En a ys -> En a (xs ++ ys).
Proof.
  induction xs as [|x xs' HI].
  -
    (* A : Type
    =====
    forall (ys : list A) (a : A),
      En a [ ] \/\ En a ys -> En a ([ ] ++ ys) *)
    (* forall (ys : list A) (a : A),
      False \/\ En a ys -> En a ys *)
    simpl.
    intros ys a [[] | H].
    (* A : Type
    ys : list A
    a : A

```

```

      H : En a ys
      =====
      En a ys *)

apply H.
-
      (* A : Type
      x : A
      xs' : list A
      HI : forall (ys : list A) (a : A),
           En a xs' \ / En a ys ->
           En a (xs' ++ ys)
      =====
      forall (ys : list A) (a : A),
           En a (x :: xs') \ / En a ys ->
           En a ((x :: xs') ++ ys) *)

simpl.
      (* forall (ys : list A) (a : A),
           (x = a \ / En a xs') \ / En a ys ->
           x = a \ / En a (xs' ++ ys) *)

intros ys a [[H1 | H2] | H3].
+
      (* A : Type
      x : A
      xs' : list A
      HI : forall (ys : list A) (a : A),
           En a xs' \ / En a ys ->
           En a (xs' ++ ys)
      ys : list A
      a : A
      H1 : x = a
      =====
      x = a \ / En a (xs' ++ ys) *)

      (* x = a *)

left.
apply H1.
+
      (* A : Type
      x : A
      xs' : list A
      HI : forall (ys : list A) (a : A),
           En a xs' \ / En a ys ->
           En a (xs' ++ ys)
      ys : list A
      a : A
      H2 : En a xs'

```

```

right.
apply HI.
left.
apply H2.
+
right.
apply HI.
right.
apply H3.
Qed.

=====
x = a \/\ En a (xs' ++ ys) *)
(* En a (xs' ++ ys) *)
(* En a xs' \/\ En a ys *)
(* En a xs' *)

(* A : Type
x : A
xs' : list A
HI : forall (ys : list A) (a : A),
      En a xs' \/\ En a ys ->
      En a (xs' ++ ys)
ys : list A
a : A
H3 : En a ys
=====
x = a \/\ En a (xs' ++ ys) *)
(* En a (xs' ++ ys) *)
(* En a xs' \/\ En a ys *)
(* En a ys *)

Lemma En_conc: forall A (xs ys : list A) (a : A),
  En a (xs ++ ys) <-> En a xs \/\ En a ys.
Proof.
split.
-
  (* A : Type
  xs, ys : list A
  a : A
  =====
  En a (xs ++ ys) -> En a xs \/\ En a ys *)
  apply En_conc_1.
-
  (* A : Type
  xs, ys : list A
  a : A
  =====
  En a xs \/\ En a ys -> En a (xs ++ ys) *)
  apply En_conc_2.
Qed.

```

```
(* -----
Ejercicio 3.3.1. Definir la propiedad
  Todos {T : Type} (P : T -> Prop) (xs : list T) : Prop
tal que (Todos P xs) se verifica si todos los elementos de xs cumplen
la propiedad P.
----- *)
```

```
Fixpoint Todos {T : Type} (P : T -> Prop) (xs : list T) : Prop :=
match xs with
| nil      => True
| x :: xs' => P x /\ Todos P xs'
end.
```

```
(* -----
Ejercicio 3.3.2. Demostrar que
  forall T (P : T -> Prop) (xs : list T),
    (forall x, En x xs -> P x) <->
    Todos P xs.
----- *)
```

```
Lemma Todos_En_1:
forall T (P : T -> Prop) (xs : list T),
  (forall x, En x xs -> P x) ->
  Todos P xs.
```

**Proof.**

```
induction xs as [|x' xs' HI].
```

```
-
```

```
simpl.
```

```
intros.
```

```
apply I.
```

```
-
```

```
(* T : Type
   P : T -> Prop
   =====
   (forall x : T, En x [ ] -> P x) ->
   Todos P [ ] *)
(* (forall x : T, False -> P x) -> True *)
(* T : Type
   P : T -> Prop
   H : forall x : T, False -> P x
   =====
   True *)

(* T : Type
```



```

P : T -> Prop
x' : T
xs' : list T
HI : (forall x : T, En x xs' -> P x) ->
      Todos P xs'
=====
(forall x : T, En x (x' :: xs') -> P x) ->
  Todos P (x' :: xs') *)
simpl.
(* (forall x : T, x' = x /\ En x xs' -> P x) ->
    P x' /\ Todos P xs' *)
intros H.
(* T : Type
   P : T -> Prop
   x' : T
   xs' : list T
   HI : (forall x : T, En x xs' -> P x) ->
         Todos P xs'
   H : forall x : T, x' = x /\ En x xs' -> P x
   =====
   P x' /\ Todos P xs' *)

split.
+
(* T : Type
   P : T -> Prop
   x' : T
   xs' : list T
   HI : (forall x : T, En x xs' -> P x) ->
         Todos P xs'
   H : forall x : T, x' = x /\ En x xs' -> P x
   =====
   P x' *)
apply H.
left.
reflexivity.
+
(* T : Type
   P : T -> Prop
   x' : T
   xs' : list T
   HI : (forall x : T, En x xs' -> P x) ->
         Todos P xs'
   H : forall x : T, x' = x /\ En x xs' -> P x
   =====

```

```

apply HI.
intros x H1.

apply H.
right.
apply H1.
Qed.

Lemma Todos_En_2:
  forall T (P : T -> Prop) (xs : list T),
    Todos P xs ->
      (forall x, En x xs -> P x).
Proof.
  induction xs as [|x' xs' HI].
  -
    (* T : Type
       P : T -> Prop
       =====
       Todos P [ ] ->
         forall x : T, En x [ ] -> P x *)
    (* True -> forall x : T, False -> P x *)
    (* T : Type
       P : T -> Prop
       =====
       forall x : T, False -> P x *)

    simpl.
    intros [].

    intros x [].
  -
    (* T : Type
       P : T -> Prop
       x' : T
       xs' : list T
       HI : Todos P xs' ->
         Todos P xs' *)
    (* forall x : T, En x xs' -> P x *)
    (* T : Type
       P : T -> Prop
       x' : T
       xs' : list T
       HI : En x xs'
       =====
       P x *)
    (* x' = x \ / En x xs' *)
    (* En x xs' *)

```

**simpl.**

**intros** [H1 H2] x [H3 | H4].

+

**rewrite** <- H3.

**apply** H1.

+

**apply** HI.

\*

```

      forall x : T, En x xs' -> P x
=====
  Todos P (x' :: xs') ->
    forall x : T, En x (x' :: xs') -> P x *)
(* P x' /\ Todos P xs' ->
  forall x : T, x' = x \/ En x xs' -> P x *)

(* T : Type
  P : T -> Prop
  x' : T
  xs' : list T
  HI : Todos P xs' ->
      forall x : T, En x xs' -> P x
  H1 : P x'
  H2 : Todos P xs'
  x : T
  H3 : x' = x
=====
  P x *)
(* P x' *)

(* T : Type
  P : T -> Prop
  x' : T
  xs' : list T
  HI : Todos P xs' ->
      forall x : T, En x xs' -> P x
  H1 : P x'
  H2 : Todos P xs'
  x : T
  H4 : En x xs'
=====
  P x *)

(* T : Type
  P : T -> Prop
  x' : T
  xs' : list T
  HI : Todos P xs' ->
      forall x : T, En x xs' -> P x

```

```

H1 : P x'
H2 : Todos P xs'
x : T
H4 : En x xs'
=====
Todos P xs' *)

apply H2.
*
(* T : Type
P : T -> Prop
x' : T
xs' : list T
H1 : Todos P xs' ->
    forall x : T, En x xs' -> P x
H1 : P x'
H2 : Todos P xs'
x : T
H4 : En x xs'
=====
En x xs' *)

apply H4.
Qed.

Lemma Todos_En:
  forall T (P : T -> Prop) (xs : list T),
    (forall x, En x xs -> P x) <->
      Todos P xs.
Proof.
  split.
  -
    (* T : Type
P : T -> Prop
xs : list T
=====
(forall x : T, En x xs -> P x) -> Todos P xs *)
    apply Todos_En_1.
  -
    (* T : Type
P : T -> Prop
xs : list T
=====
Todos P xs -> forall x : T, En x xs -> P x *)
    apply Todos_En_2.

```

**Qed.**

```
(* -----
  Ejercicio 3.4.1. Definir la propiedad
    combina_par_impar (Pimpar Ppar : nat -> Prop) : nat -> Prop
  tal que (combina_par_impar Pimpar Ppar) es una función que asigna a n
  (Pimpar n) si n es impar y (Ppar n) si n es par.
  ----- *)
```

**Definition** combina\_par\_impar (Pimpar Ppar : **nat** -> **Prop**) : **nat** -> **Prop** :=  
**fun** n => (esImpar n = **true** -> Pimpar n) /\  
 (esImpar n = **false** -> Ppar n).

```
(* -----
  Ejercicio 3.4.2. Demostrar que
    forall (Pimpar Ppar : nat -> Prop) (n : nat),
      (esImpar n = true -> Pimpar n) ->
      (esImpar n = false -> Ppar n) ->
      combina_par_impar Pimpar Ppar n.
  ----- *)
```

**Theorem** combina\_par\_impar\_intro :  
**forall** (Pimpar Ppar : **nat** -> **Prop**) (n : **nat**),  
 (esImpar n = **true** -> Pimpar n) ->  
 (esImpar n = **false** -> Ppar n) ->  
 combina\_par\_impar Pimpar Ppar n.

**Proof.**

```
intros Pimpar Ppar n H1 H2. (* Pimpar, Ppar : nat -> Prop
                               n : nat
                               H1 : esImpar n = true -> Pimpar n
                               H2 : esImpar n = false -> Ppar n
                               =====
                               combina_par_impar Pimpar Ppar n *)
unfold combina_par_impar. (* (esImpar n = true -> Pimpar n) /\
                               (esImpar n = false -> Ppar n) *)
split.
-
  (* Pimpar, Ppar : nat -> Prop
     n : nat
     H1 : esImpar n = true -> Pimpar n
     H2 : esImpar n = false -> Ppar n
```

```

=====
esImpar n = true -> Pimpar n *)

apply H1.
-
(* Pimpar, Ppar : nat -> Prop
   n : nat
   H1 : esImpar n = true -> Pimpar n
   H2 : esImpar n = false -> Ppar n
   =====
   esImpar n = false -> Ppar n *)

apply H2.
Qed.

(* -----
   Ejercicio 3.4.3. Demostrar que
   forall (Pimpar Ppar : nat -> Prop) (n : nat),
     combina_par_impar Pimpar Ppar n ->
     esImpar n = true ->
     Pimpar n.
   ----- *)

Theorem combina_par_impar_elim_impar:
forall (Pimpar Ppar : nat -> Prop) (n : nat),
  combina_par_impar Pimpar Ppar n ->
  esImpar n = true ->
  Pimpar n.

Proof.
intros Pimpar Ppar n H1 H2. (* Pimpar, Ppar : nat -> Prop
                               n : nat
                               H1 : combina_par_impar Pimpar Ppar n
                               H2 : esImpar n = true
                               =====
                               Pimpar n *)
unfold combina_par_impar in H1. (* Pimpar, Ppar : nat -> Prop
                                   n : nat
                                   H1 : (esImpar n = true -> Pimpar n) /\
                                       (esImpar n = false -> Ppar n)
                                   H2 : esImpar n = true
                                   =====
                                   Pimpar n *)
destruct H1 as [H3 H4]. (* Pimpar, Ppar : nat -> Prop

```

```

n : nat
H3 : esImpar n = true -> Pimpar n
H4 : esImpar n = false -> Ppar n
H2 : esImpar n = true
=====
Pimpar n *)
(* esImpar n = true *)

apply H3.
apply H2.
Qed.

(* -----
Ejercicio 3.4.4. Demostrar que
  forall (Pimpar Ppar : nat -> Prop) (n : nat),
    combina_par_impar Pimpar Ppar n ->
      esImpar n = false ->
        Ppar n.
----- *)

Theorem combina_par_impar_elim_par:
forall (Pimpar Ppar : nat -> Prop) (n : nat),
  combina_par_impar Pimpar Ppar n ->
    esImpar n = false ->
      Ppar n.

Proof.
intros Pimpar Ppar n H1 H2. (* Pimpar, Ppar : nat -> Prop
n : nat
H1 : combina_par_impar Pimpar Ppar n
H2 : esImpar n = false
=====
Ppar n *)

unfold combina_par_impar in H1. (* Pimpar, Ppar : nat -> Prop
n : nat
H1 : (esImpar n = true -> Pimpar n) /\
      (esImpar n = false -> Ppar n)
H2 : esImpar n = false
=====
Ppar n *)

destruct H1 as [H3 H4]. (* Pimpar, Ppar : nat -> Prop
n : nat
H3 : esImpar n = true -> Pimpar n
H4 : esImpar n = false -> Ppar n
=====
Ppar n *)

```

```

H4 : esImpar n = false -> Ppar n
H2 : esImpar n = false
=====
Ppar n *)
(* esImpar n = false *)

apply H4.
apply H2.
Qed.

(* =====
§ 4. Aplicando teoremas a argumentos
===== *)

(* -----
Ejemplo 4.1. Evaluar la expresión
  Check suma_conmutativa.
----- *)

Check suma_conmutativa.
(* ==> forall n m : nat, n + m = m + n *)

(* -----
Notas.
1. En Coq, las demostraciones son objetos de primera clase.
2. Coq devuelve el tipo de suma_conmutativa como el de cualquier
   expresión.
3. El identificador suma_conmutativa representa un objeto prueba de
   (forall n m : nat, n + m = m + n).
4. Un término de tipo (nat -> nat -> nat) transforma dos naturales en
   un natural.
5. Análogamente, un término de tipo (n = m -> n + n = m + m)
   transforma un argumento de tipo (n = m) en otro de tipo
   (n + n = m + m).
----- *)

(** Operationally, this analogy goes even further: by applying a
    theorem, as if it were a function, to hypotheses with matching
    types, we can specialize its result without having to resort to

```



*intermediate assertions. For example, suppose we wanted to prove the following result: \*)*

```
(* -----
Ejemplo 4.2. Demostrar que
  forall x y z : nat,
    x + (y + z) = (z + y) + x.
----- *)
```

*(\* 1º intento \*)*

```
Lemma suma_conmutativa3a :
  forall x y z : nat,
    x + (y + z) = (z + y) + x.
```

**Proof.**

```
  intros x y z. (* x, y, z : nat
=====
                    x + (y + z) = (z + y) + x *)
  rewrite suma_conmutativa. (* (y + z) + x = (z + y) + x *)
  rewrite suma_conmutativa. (* x + (y + z) = (z + y) + x *)
```

Abort.

*(\* 2º intento \*)*

```
Lemma suma_conmutativa3b :
  forall x y z,
    x + (y + z) = (z + y) + x.
```

**Proof.**

```
  intros x y z. (* x, y, z : nat
=====
                    x + (y + z) = z + y + x *)
  rewrite suma_conmutativa. (* (y + z) + x = (z + y) + x *)
  assert (H : y + z = z + y).
- (* x, y, z : nat
=====
                    y + z = z + y *)
  rewrite suma_conmutativa. (* z + y = z + y *)
  reflexivity.
- (* x, y, z : nat
    H : y + z = z + y
=====
                    (y + z) + x = (z + y) + x *)
```

```

    rewrite H.                                (* (z + y) + x = (z + y) + x *)
    reflexivity.
Qed.

(* 3º intento *)
Lemma suma_conmutativa3c:
  forall x y z,
    x + (y + z) = (z + y) + x.
Proof.
  intros x y z.                               (* x, y, z : nat
                                              =====
                                              x + (y + z) = (z + y) + x *)
  rewrite suma_conmutativa.                    (* (y + z) + x = (z + y) + x *)
  rewrite (suma_conmutativa y z).              (* (z + y) + x = (z + y) + x *)
  reflexivity.
Qed.

(* -----
   Nota. Indicación en (rewrite (suma_conmutativa y z)) de los
   argumentos con los que se aplica, análogamente a las funciones
   polimórficas.
   ----- *)

(* -----
   Ejemplo 4.3. Demostrar que
   forall {n : nat} {ns : list nat},
     En n (map (fun m => m * 0) ns) ->
     n = 0.
   ----- *)

(* Lema auxiliar *)
Lemma producto_n_0:
  forall n : nat, n * 0 = 0.
Proof.
  induction n as [|n' HI].
  -
    (*
    =====
    0 * 0 = 0 *)
    reflexivity.
  -
    (* n' : nat

```

```

                                HI : n' * 0 = 0
                                =====
                                S n' * 0 = 0 *)
    simpl.                      (* n' * 0 = 0 *)
    apply HI.
Qed.

(* 1ª demostración *)
Example ej_aplicacion_de_lemma_1:
  forall {n : nat} {ns : list nat},
    En n (map (fun m => m * 0) ns) ->
      n = 0.
Proof.
  intros n ns H.              (* n : nat
                                ns : list nat
                                H : En n (map (fun m : nat => m * 0) ns)
                                =====
                                n = 0 *)
  rewrite En_map_iff in H.     (* n : nat
                                ns : list nat
                                H : exists x : nat, x * 0 = n /\ En x ns
                                =====
                                n = 0 *)
  destruct H as [m [Hm _]].   (* n : nat
                                ns : list nat
                                m : nat
                                Hm : m * 0 = n
                                =====
                                n = 0 *)
  rewrite producto_n_0 in Hm.  (* n : nat
                                ns : list nat
                                m : nat
                                Hm : 0 = n
                                =====
                                n = 0 *)
  symmetry.                   (* 0 = n *)
  apply Hm.
Qed.

(* 2ª demostración *)

```

**Example** ej\_aplicacion\_de\_lema:

```
forall {n : nat} {ns : list nat},
  En n (map (fun m => m * 0) ns) ->
  n = 0.
```

**Proof.**

```
intros n ns H.
```

```
(* n : nat
   ns : list nat
   H : En n (map (fun m : nat => m * 0) ns)
   =====
   n = 0 *)
```

```
destruct (conj_el _ _
  (En_map_iff _ _ _ _))
  H)
  as [m [Hm _]].
```

```
(* n : nat
   ns : list nat
   H : En n (map (fun m : nat => m * 0) ns)
   m : nat
   Hm : m * 0 = n
   =====
   n = 0 *)
```

```
rewrite producto_n_0 in Hm.
```

```
(* n : nat
   ns : list nat
   H : En n (map (fun m : nat => m * 0) ns)
   m : nat
   Hm : 0 = n
   =====
   n = 0 *)
```

```
symmetry.
```

```
(* 0 = n *)
```

```
apply Hm.
```

**Qed.**

```
(* -----
   Nota. Aplicación de teoremas a argumentos con
   (conj_el _ _ (En_map_iff _ _ _ _)) H)
   ----- *)
```

```
(* =====
   § 5. Coq vs. teoría de conjuntos
   ===== *)
```

```

(* -----
Notas.
1. En lugar de decir que un elemento pertenece a un conjunto se puede
   decir que verifica la propiedad que define al conjunto.
----- *)

(* =====
§§ 5.1. Extensionalidad funcional
===== *)

(* -----
Ejemplo 5.1.1. Demostrar que
plus 3 = plus (pred 4).
----- *)

```

**Example** igualdad\_de\_funciones\_ej1:

suma 3 = suma (pred 4).

**Proof.**

reflexivity.

**Qed.**

```

(* -----
Ejemplo 5.1.2. Definir el axioma de extensionalidad funcional que
afirma que dos funciones son iguales cuando tienen los mismos
valores.
----- *)

```

**Axiom** extensionalidad\_funcional : forall {X Y: Type}

{f g : X -> Y},

(forall (x:X), f x = g x) -> f = g.

```

(* -----
Ejemplo 5.1.3. Demostrar que
(fun x => suma x 1) = (fun x => suma 1 x).
----- *)

```

**Example** igualdad\_de\_funciones\_ej2 :

(fun x => suma x 1) = (fun x => suma 1 x).

**Proof.**

apply extensionalidad\_funcional. (\*

```

=====
forall x : nat, suma x 1 = suma 1 x *)
(* x : nat
=====
suma x 1 = suma 1 x *)

intros x.

apply suma_conmutativa.
Qed.

(* -----
Notas.
1. No se puede demostrar sin el axioma.
2. Hay que ser cuidadoso en la definición de axiomas, porque se
   pueden introducir inconsistencias.
----- *)

(* -----
Ejemplo 5.1.4. Calcular los axiomas usados en la prueba de
igualdad_de_funciones_ej2
----- *)

Print Assumptions igualdad_de_funciones_ej2.
(* ==>
Axioms:
extensionalidad_funcional :
  forall (X Y : Type) (f g : X -> Y),
    (forall x : X, f x = g x) -> f = g *)

(* -----
Ejercicio 5.1.1. Se considera la siguiente definición iterativa de la
función inversa
  Fixpoint inversaIaux {X} (xs ys : list X) : list X :=
    match xs with
    | []      => ys
    | x :: xs' => inversaIaux xs' (x :: ys)
    end.

  Definition inversaI {X} (xs : list X) : list X :=
    inversaIaux xs [].

Demostrar que

```

```

    forall X : Type,
      @inversaI X = @inversa X.
    ----- *)

Fixpoint inversaIaux {X} (xs ys : list X) : list X :=
  match xs with
  | []      => ys
  | x :: xs' => inversaIaux xs' (x :: ys)
  end.

Definition inversaI {X} (xs : list X) : list X :=
  inversaIaux xs [].

Lemma inversaI_correcta_aux:
  forall (X : Type) (xs ys : list X),
    inversaIaux xs ys = inversa xs ++ ys.
Proof.
  intros X xs.
  (* X : Type
     xs : list X
     =====
     forall ys : list X,
       inversaIaux xs ys = inversa xs ++ ys *)

  induction xs as [|x xs' HI].
  -
    (* X : Type
       =====
       forall ys : list X,
         inversaIaux [ ] ys = inversa [ ] ++ ys *)
    simpl.
    intros.
    (* forall ys : list X, ys = ys *)
    (* X : Type
       ys : list X
       =====
       ys = ys *)

    reflexivity.
  -
    (* X : Type
       x : X
       xs' : list X
       HI : forall ys : list X,
           inversaIaux xs' ys = inversa xs' ++ ys
       =====
       forall ys : list X,

```

```

      inversaIaux (x :: xs') ys =
      inversa (x :: xs') ++ ys *)
intros ys.
(* X : Type
   x : X
   xs' : list X
   HI : forall ys : list X,
       inversaIaux xs' ys = inversa xs' ++ ys
   ys : list X
   =====
   inversaIaux (x :: xs') ys =
   inversa (x :: xs') ++ ys *)
simpl.
(* inversaIaux xs' (x :: ys) =
   (inversa xs' ++ [x]) ++ ys *)
rewrite <- conc_asociativa. (* inversaIaux xs' (x :: ys) =
   inversa xs' ++ ([x] ++ ys) *)
apply HI.
Qed.

Lemma inversaI_correcta:
  forall X : Type,
    @inversaI X = @inversa X.
Proof.
  intros X.
  (* X : Type
   =====
   inversaI = inversa *)
  apply extensionalidad_funcional. (* forall x : list X,
   inversaI x = inversa x *)
  intros.
  (* X : Type
   x : list X
   =====
   inversaI x = inversa x *)
  unfold inversaI.
  (* inversaIaux x [ ] = inversa x *)
  rewrite inversaI_correcta_aux. (* inversa x ++ [ ] = inversa x *)
  apply conc_nil.
Qed.

(* =====
   §§ 5.2. Proposiciones y booleanos
   ===== *)

```



```
(* -----
  Ejemplo 5.2.1. Demostrar que
    forall k : nat,
      esPar (doble k) = true.
  ----- *)
```

**Theorem** esPar\_doble:

```
forall k : nat,
  esPar (doble k) = true.
```

**Proof.**

```
intros k. (* k : nat
           =====
           esPar (doble k) = true *)

induction k as [|k' HI].
- (*
   =====
   esPar (doble 0) = true *)

  reflexivity.
- (* k' : nat
   HI : esPar (doble k') = true
   =====
   esPar (doble (S k')) = true *)

  simpl.
  apply HI.
```

**Qed.**

```
(* -----
  Ejercicio 5.2.1. Demostrar que
    forall n : nat,
      exists k : nat, n = if esPar n
                          then doble k
                          else S (doble k).
  ----- *)
```

**Theorem** esPar\_doble\_aux :

```
forall n : nat,
  exists k : nat, n = if esPar n
                      then doble k
                      else S (doble k).
```

**Proof.**

**induction** n **as** [|n' HI].

-

(\*

=====

exists k : nat,

0 = (if esPar 0  
then doble k  
else S (doble k)) \*)

**exists** 0.

(\* 0 = (if esPar 0  
then doble 0  
else S (doble 0)) \*)

**reflexivity.**

-

(\* n' : nat

HI : exists k : nat,

n' = (if esPar n'  
then doble k  
else S (doble k))

=====

exists k : nat,

S n' = (if esPar (S n')  
then doble k  
else S (doble k)) \*)

**destruct** (esPar n') eqn:H.

+

(\* n' : nat

H : esPar n' = true

HI : exists k : nat, n' = doble k

=====

exists k : nat,

S n' = (if esPar (S n')  
then doble k  
else S (doble k)) \*)

**rewrite** esPar\_S.

(\* exists k : nat,

S n' = (if negacion (esPar n')  
then doble k  
else S (doble k)) \*)

**rewrite** H.

(\* exists k : nat,

S n' = (if negacion true  
then doble k  
else S (doble k)) \*)

**simpl.**

(\* exists k : nat, S n' = S (doble k) \*)

```

destruct HI as [k' Hk']. (* n' : nat
                           H : esPar n' = true
                           k' : nat
                           Hk' : n' = doble k'
                           =====
                           exists k : nat, S n' = S (doble k) *)
exists k'.
rewrite Hk'.
reflexivity.
+
(* n' : nat
   H : esPar n' = false
   HI : exists k : nat, n' = S (doble k)
   =====
   exists k : nat,
     S n' = (if esPar (S n')
              then doble k
              else S (doble k)) *)
rewrite esPar_S.
(* exists k : nat,
   S n' = (if negacion (esPar n')
            then doble k
            else S (doble k)) *)
rewrite H.
(* exists k : nat,
   S n' = (if negacion false
            then doble k
            else S (doble k)) *)
simpl.
(* exists k : nat, S n' = doble k *)
destruct HI as [k' Hk']. (* n' : nat
                           H : esPar n' = false
                           k' : nat
                           Hk' : n' = S (doble k')
                           =====
                           exists k : nat, S n' = doble k *)
exists (1 + k').
(* S n' = doble (1 + k') *)
rewrite Hk'.
(* S (S (doble k')) = doble (1 + k') *)
reflexivity.

Qed.

(* -----
   Ejemplo 5.2.2. Demostrar que
   forall n : nat,

```

*esPar n = true <-> exists k, n = doble k.*

*Es decir, que la computación booleana (esPar n) refleja la proposición (exists k, n = doble k).*

----- \*)

**Theorem** esPar\_bool\_prop:

**forall** n : **nat**,

esPar n = **true** <-> **exists** k, n = doble k.

**Proof.**

**intros** n.

(\* n : nat

=====

esPar n = true <-> (exists k : nat, n = doble k) \*)

**split.**

-

(\* n : nat

=====

esPar n = true -> exists k : nat, n = doble k \*)

**intros** H.

(\* n : nat

H : esPar n = true

=====

exists k : nat, n = doble k \*)

**destruct**

(esPar\_doble\_aux n)

**as** [k Hk].

(\* n : nat

H : esPar n = true

k : nat

Hk : n = (if esPar n then doble k else S (doble k))

=====

exists k0 : nat, n = doble k0 \*)

**rewrite** Hk.

(\* exists k0 : nat,

(if esPar n

then doble k

else S (doble k))

= doble k0 \*)

**rewrite** H.

(\* exists k0 : nat, doble k = doble k0 \*)

**exists** k.

(\* doble k = doble k \*)

**reflexivity.**

-

(\* n : nat

=====

(exists k : nat, n = doble k) -> esPar n = true \*)

```

intros [k Hk].          (* n, k : nat
                           Hk : n = doble k
                           =====
                           esPar n = true *)
rewrite Hk.              (* esPar (doble k) = true *)
apply esPar_doble.
Qed.

(* -----
   Ejemplo 5.2.3. Demostrar que
     forall n m : nat,
       iguales_nat n m = true <-> n = m.
   ----- *)

Theorem iguales_nat_bool_prop:
  forall n m : nat,
    iguales_nat n m = true <-> n = m.
Proof.
  intros n m.            (* n, m : nat
                           =====
                           iguales_nat n m = true <-> n = m *)
  split.
  -
    (* n, m : nat
       =====
       iguales_nat n m = true -> n = m *)
  apply iguales_nat_true.
  -
    (* n, m : nat
       =====
       n = m -> iguales_nat n m = true *)
  intros H.              (* n, m : nat
                           H : n = m
                           =====
                           iguales_nat n m = true *)
  rewrite H.              (* iguales_nat m m = true *)
  rewrite iguales_nat_refl. (* true = true *)
  reflexivity.
Qed.

(* -----
   Ejemplo 5.2.4. Definir la función es_primo_par tal que

```

```

    (es_primo_par n) es verifica si n es un primo par.
    ----- *)

(* 1º intento *)
Fail Definition es_primo_par n :=
  if n = 2
  then true
  else false.

(* 2º intento *)
Definition es_primo_par n :=
  if iguales_nat n 2
  then true
  else false.

(* -----
    Ejemplo 5.2.5.1. Demostrar que
    exists k : nat, 1000 = doble k.
    ----- *)

Example esPar_1000: exists k : nat, 1000 = doble k.
Proof.
  exists 500.
  reflexivity.
Qed.

(* -----
    Ejemplo 5.2.5.2. Demostrar que
    esPar 1000 = true.
    ----- *)

Example esPar_1000' : esPar 1000 = true.
Proof.
  reflexivity.
Qed.

(* -----
    Ejemplo 5.2.5.3. Demostrar que
    exists k : nat, 1000 = doble k.
    ----- *)

```

**Example** esPar\_1000'': **exists** k : **nat**, 1000 = doble k.

**Proof.**

**apply** esPar\_bool\_prop. (\* esPar 1000 = true \*)

**reflexivity.**

**Qed.**

```
(* -----
Notas.
1. En la proposicional se necesita proporcionar un testigo.
2. En la booleano se calcula sin testigo.
3. Se puede demostrar la proposicional usando la equivalencia con la
   booleana sin necesidad de testigo.
----- *)
```

```
(* -----
Ejercicio 5.2.2.1. Demostrar que
  forall x y : bool,
    x && y = true <-> x = true /\ y = true.
----- *)
```

**Lemma** conj\_verdad\_syss:

**forall** x y : **bool**,

x && y = **true** <-> x = **true** /\ y = **true**.

**Proof.**

**intros** x y.

(\* x, y : bool

=====

x && y = true <-> x = true /\ y = true \*)

**destruct** x.

-

(\* y : bool

=====

true && y = true <-> true = true /\ y = true \*)

**destruct** y.

+

(\*

=====

true && true = true <-> true = true /\ true=true \*)

**simpl.**

(\* true = true <-> true = true /\ true = true \*)

**split.**

\*

(\*

=====

```

                                true = true -> true = true /\ true = true *)
apply conj_intro. (* true = true *)
reflexivity.
*                               (*
                                =====
                                true = true /\ true = true -> true = true *)

+   apply conj_e1.
                                (*
                                =====
                                true && false = true <-> true=true /\ false=true *)
simpl.
split.
*                               (*
                                =====
                                false = true -> true = true /\ false = true *)
intros H.
                                (* H : false = true
                                =====
                                true = true /\ false = true *)

inversion H.
*                               (*
                                =====
                                true = true /\ false = true -> false = true *)

intros [H1 H2].
                                (* H1 : true = true
                                H2 : false = true
                                =====
                                false = true *)

apply H2.
-   (* y : bool
                                =====
                                false && y = true <-> false = true /\ y = true *)

split.
+   (* y : bool
                                =====
                                false && y = true -> false = true /\ y = true *)
simpl.
                                (* false = true -> false = true /\ y = true *)
intros H.
                                (* y : bool
                                H : false = true
                                =====
                                false = true /\ y = true *)

inversion H.

```



```

+
      (* y : bool
      =====
      false = true /\ y = true -> false && y = true *)
simpl.
intros [H1 H2].
      (* y : bool
      H1 : false = true
      H2 : y = true
      =====
      false = true *)

apply H1.
Qed.

(* -----
   Ejercicio 5.2.2.2. Demostrar que
   forall x y : bool,
   x || y = true <-> x = true \/ y = true.
   ----- *)

Lemma disy_verdad_syss:
  forall x y : bool,
    x || y = true <-> x = true \/ y = true.
Proof.
  intros x y.
      (* x, y : bool
      =====
      x || y = true <-> x = true \/ y = true *)

  destruct x.
  -
      (* y : bool
      =====
      true || y = true <-> true = true \/ y = true *)
      (* true = true <-> true = true \/ y = true *)

  simpl.
  split.
  +
      (* y : bool
      =====
      true = true -> true = true \/ y = true *)

      apply disy_intro.
  +
      (* y : bool
      =====
      true = true \/ y = true -> true = true *)

  intros.
      (* y : bool
      H : true = true \/ y = true

```

```

=====
true = true *)

reflexivity.
-
(* y : bool
=====
false || y = true <-> false = true \/ y = true *)
simpl.
split.
+
(* y : bool
=====
y = true -> false = true \/ y = true *)

destruct y.
*
(*
=====
true = true -> false = true \/ true = true *)
intros.
(* H : true = true
=====
false = true \/ true = true *)
right.
reflexivity.
*
(*
=====
false = true -> false = true \/ false = true *)

apply disy_intro.
+
(* y : bool
=====
false = true \/ y = true -> y = true *)

intros [H1 | H2].
*
(* y : bool
H1 : false = true
=====
y = true *)

inversion H1.
*
(* y : bool
H2 : y = true
=====
y = true *)

apply H2.

Qed.

```

```
(* -----
  Ejercicio 5.2.3. Demostrar que
    forall x y : nat,
      iguales_nat x y = false <-> x <> y.
  ----- *)
```

**Theorem** iguales\_nat\_falso\_syss:

```
forall x y : nat,
  iguales_nat x y = false <-> x <> y.
```

**Proof.**

```
intros x y.                                     (* x, y : nat
                                              =====
                                              iguales_nat x y = false <-> x <> y *)

destruct (iguales_nat x y) eqn:H.              (* x, y : nat
                                              H : iguales_nat x y = true
                                              =====
                                              true = false <-> x <> y *)

rewrite iguales_nat_bool_prop in H.           (* x, y : nat
                                              H : x = y
                                              =====
                                              true = false <-> x <> y *)

rewrite H.                                     (* true = false <-> y <> y *)

split.                                         (* x, y : nat
                                              H : x = y
                                              =====
                                              true = false -> y <> y *)

+                                              (* x, y : nat
                                              H : x = y
                                              =====
                                              true = false -> y <> y *)

  intros H1.                                  (* x, y : nat
                                              H : x = y
                                              H1 : true = false
                                              =====
                                              y <> y *)

  inversion H1.                               (* x, y : nat
                                              H : x = y
                                              =====
                                              y <> y -> true = false *)

+                                              (* x, y : nat
                                              H : x = y
                                              =====
                                              y <> y -> true = false *)

  intros H1.
```

```

exfalso.
unfold not in H1.

apply H1.
apply eq_refl.

-

split.
+

unfold not.
intros H1 H2.

rewrite H2 in H.

rewrite iguales_nat_refl in H.

inversion H.

```

```

H1 : y <> y
=====
true = false *)
(* False *)
(* x, y : nat
   H : x = y
   H1 : y = y -> False
   =====
   False *)
(* y = y *)

(* x, y : nat
   H : iguales_nat x y = false
   =====
   false = false <-> x <> y *)

(* x, y : nat
   H : iguales_nat x y = false
   =====
   false = false -> x <> y *)
(* false = false -> x = y -> False *)
(* x, y : nat
   H : iguales_nat x y = false
   H1 : false = false
   H2 : x = y
   =====
   False *)
(* x, y : nat
   H : iguales_nat y y = false
   H1 : false = false
   H2 : x = y
   =====
   False *)
(* x, y : nat
   H : true = false
   H1 : false = false
   H2 : x = y
   =====
   False *)

```

```

+
      intros.

      reflexivity.

Qed.

(* -----
Ejercicio 5.2.4.1. Definir la función
  iguales_lista {A : Type} (i : A -> A -> bool) (xs ys : list A)
tal que (iguales_lists xs ys) se verifica si los correspondientes
elementos de las listas xs e ys son iguales respecto de la relación
de igualdad i.
----- *)

Fixpoint iguales_lista {A : Type} (i : A -> A -> bool) (xs ys : list A) : bool :=
match xs, ys with
| nil, nil           => true
| x' :: xs', y' :: ys' => i x' y' && iguales_lista i xs' ys'
| _, _              => false
end.

(* -----
Ejercicio 5.2.4.2. Demostrar que
  forall A (i : A -> A -> bool),
    (forall x y, i x y = true <-> x = y) ->
    forall xs ys, iguales_lista i xs ys = true <-> xs = ys.
----- *)

Lemma iguales_lista_verdad_CN:
forall A (i : A -> A -> bool),
  (forall x y, i x y = true <-> x = y) ->
  forall xs ys, iguales_lista i xs ys = true -> xs = ys.
Proof.
  intros A i H xs.

```

```

induction xs as [|x xs' HIXs'].
-
  (* A : Type
  i : A -> A -> bool
  H : forall x y : A,
      i x y = true <-> x = y
  xs : list A
  =====
  forall ys : list A,
      iguales_lista i xs ys = true -> xs=ys *)

destruct ys as [|y ys'].
+
  intros.

  reflexivity.
+
  (* y : A
  ys' : list A
  =====
  iguales_lista i [ ] (y :: ys') = true
  -> [ ] = y :: ys' *)
  (* false = true -> [ ] = y :: ys' *)
  (* H1 : false = true
  =====
  [ ] = y :: ys' *)

  (* x : A
  xs' : list A
  HIXs' : forall ys : list A,
      iguales_lista i xs' ys = true
      -> xs' = ys
  =====
  forall ys : list A,

```

```

destruct ys as [|y ys'].
+
  simpl.
  intros H1.

  inversion H1.
+
  simpl.
  intros H1.

  apply conj_verdad_syss in H1.

  destruct H1 as [H2 H3].

  f_equal.
  *
    apply H.
    apply H2.
  *
    apply H1xs'.
    apply H3.

Qed.

Lemma iguales_lista_verdad_CS:
  forall A (i : A -> A -> bool),

```

```

    iguales_lista i (x :: xs') ys = true
    -> x :: xs' = ys *)

(* iguales_lista i (x :: xs') [ ] = true
   -> x :: xs' = [ ] *)
(* false = true -> x :: xs' = [ ] *)
(* H1 : false = true
   =====
   x :: xs' = [ ] *)

(* y : A
   ys' : list A
   =====
   iguales_lista i (x::xs') (y::ys') = true
   -> x :: xs' = y :: ys' *)
(* i x y && iguales_lista i xs' ys' = true
   -> x :: xs' = y :: ys' *)
(* H1 : i x y && iguales_lista i xs' ys' =
   true
   =====
   x :: xs' = y :: ys' *)
(* H1 : i x y = true /\
   iguales_lista i xs' ys' = true
   =====
   x :: xs' = y :: ys' *)
(* H2 : i x y = true
   H3 : iguales_lista i xs' ys' = true
   =====
   x :: xs' = y :: ys' *)

(* x = y *)
(* i x y = true *)

(* xs' = ys' *)
(* iguales_lista i xs' ys' = true *)

```

```

    iguales_lista i (x :: xs') ys = true
    -> x :: xs' = ys *)

```

```

  (forall x y, i x y = true <-> x = y) ->
  forall xs ys, xs = ys -> iguales_lista i xs ys = true.

```

**Proof.**

```

  intros A i H xs.

```

```

  (* A : Type
     i : A -> A -> bool
     H : forall x y : A, i x y = true <-> x = y
     xs : list A
     =====
     forall ys :
       list A, xs = ys ->
       iguales_lista i xs ys = true *)

```

```

  induction xs as [|x xs' HIXs'].

```

```

  -

```

```

  (* forall ys :
     list A, [ ] = ys ->
     iguales_lista i [ ] ys = true *)

```

```

  intros ys H1.

```

```

  (* ys : list A
     H1 : [ ] = ys
     =====
     iguales_lista i [ ] ys = true *)
  (* iguales_lista i [ ] [ ] = true *)
  (* true = true *)

```

```

  rewrite <- H1.
  simpl.
  reflexivity.

```

```

  -

```

```

  (* x : A
     xs' : list A
     HIXs' : forall ys :
       list A, xs' = ys ->
       iguales_lista i xs' ys = true
     =====
     forall ys :
       list A, x :: xs' = ys ->
       iguales_lista i (x :: xs') ys = true *)

```

```

  intros ys H1.

```

```

  (* ys : list A
     H1 : x :: xs' = ys
     =====
     iguales_lista i (x :: xs') ys = true *)
  (* iguales_lista i (x::xs') (x::xs') = true *)
  (* i x x && iguales_lista i xs' xs' = true *)
  (* i x x = true /\
     iguales_lista i xs' xs' = true *)

```

```

  rewrite <-H1.
  simpl.
  apply conj_verdad_syss.

  split.

```



```

+                               (* i x x = true *)
  apply H.                     (* x = x *)
  reflexivity.

+                               (* iguales_lista i xs' xs' = true *)
  apply Hixs'.                  (* xs' = xs' *)
  reflexivity.
Qed.

```

**Lemma** iguales\_lista\_verdad\_syss:

```

forall A (i : A -> A -> bool),
  (forall x y, i x y = true <-> x = y) ->
  forall xs ys, iguales_lista i xs ys = true <-> xs = ys.

```

**Proof.**

```

intros A i H xs ys.           (* A : Type
                               i : A -> A -> bool
                               H : forall x y : A, i x y = true <-> x = y
                               xs, ys : list A
                               =====
                               iguales_lista i xs ys = true <-> xs=ys *)

split.

-                               (* iguales_lista i xs ys = true -> xs = ys *)
  apply iguales_lista_verdad_CN. (* forall x y : A, i x y = true <-> x = y *)
  apply H.

-                               (* xs = ys -> iguales_lista i xs ys = true *)
  apply iguales_lista_verdad_CS. (* forall x y : A, i x y = true <-> x = y *)
  apply H.

```

**Qed.**

```

(* -----
   Ejercicio 5.2.5. Demostrar que
   forall (X : Type) (p : X -> bool) (xs : list X),
     todos p xs = true <-> Todos (fun x => p x = true) xs.
   ----- *)

```

**Theorem** todos\_verdad\_CN:

```

forall (X : Type) (p : X -> bool) (xs : list X),
  todos p xs = true -> Todos (fun x => p x = true) xs.

```

**Proof.**

```

intros X p xs.                (* X : Type
                               p : X -> bool

```

```

xs : list X
=====
todos p xs = true ->
  Todos (fun x : X => p x = true) xs *)

induction xs as [|x' xs' HI|.
-
  simpl.
  intros.

  reflexivity.
-
  (* x' : X
     xs' : list X
     HI : todos p xs' = true ->
         Todos (fun x : X => p x = true) xs'
     =====
     todos p (x' :: xs') = true ->
       Todos (fun x : X => p x = true) (x'::xs') *)
  simpl.
  (* p x' && todos p xs' = true ->
     p x' = true /\
       Todos (fun x : X => p x = true) xs' *)
  intros H.
  (* H : p x' && todos p xs' = true
     =====
     p x' = true /\
       Todos (fun x : X => p x = true) xs' *)
  apply conj_verdad_sys in H.
  (* H : p x' = true /\ todos p xs' = true
     =====
     p x' = true /\
       Todos (fun x : X => p x = true) xs' *)
  destruct H as [H1 H2].
  (* H1 : p x' = true
     H2 : todos p xs' = true
     =====
     p x' = true /\
       Todos (fun x : X => p x = true) xs' *)

  split.
  +
    apply H1.
  +
    apply HI.
  (* p x' = true *)

  (* Todos (fun x : X => p x = true) xs' *)
  (* todos p xs' = true *)

```

```

    apply H2.
Qed.

```

**Theorem** todos\_verdad\_CS:

```

forall (X : Type) (p : X -> bool) (xs : list X),
  Todos (fun x => p x = true) xs -> todos p xs = true.

```

**Proof.**

```

intros X p xs.
(* X : Type
   p : X -> bool
   xs : list X
   =====
   Todos (fun x : X => p x = true) xs ->
   todos p xs = true *)

induction xs as [|x' xs' HI].
-
  (* Todos (fun x : X => p x = true) [ ] ->
   todos p [ ] = true *)
  (* True -> true = true *)
  (* H : True
   =====
   true = true *)

  reflexivity.
-
  (* x' : X
   xs' : list X
   HI : Todos (fun x : X => p x = true) xs' ->
   todos p xs' = true
   =====
   Todos (fun x : X => p x = true) (x' :: xs')
   -> todos p (x' :: xs') = true *)

  simpl.
  (* p x' = true /\
   Todos (fun x : X => p x = true) xs' ->
   p x' && todos p xs' = true *)

  intros [H1 H2].
  (* H1 : p x' = true
   H2 : Todos (fun x : X => p x = true) xs'
   =====
   p x' && todos p xs' = true *)

  apply conj_verdad_syss.
  split.
  +
    (* p x' = true *)
    apply H1.
  +
    (* todos p xs' = true *)

```

```

    apply HI.                                (* Todos (fun x : X => p x = true) xs' *)
    apply H2.
Qed.

```

```

Theorem todos_verdad_syss:
  forall (X : Type) (p : X -> bool) (xs : list X),
    todos p xs = true <-> Todos (fun x => p x = true) xs.

```

```

Proof.
  intros X p xs.                            (* X : Type
                                             p : X -> bool
                                             xs : list X
                                             =====
                                             todos p xs = true <->
                                             Todos (fun x : X => p x = true) xs *)

  split.
  -                                           (* todos p xs = true ->
                                             Todos (fun x : X => p x = true) xs *)

    apply todos_verdad_CN.
  -                                           (* Todos (fun x : X => p x = true) xs ->
                                             todos p xs = true *)

    apply todos_verdad_CS.
Qed.

```

```

(* =====
   §§ 5.3. Lógica clásica vs. constructiva
   ===== *)

(* -----
   Ejemplo 5.3.1. Definir la proposicion
   tercio_excluso
   que afirma que (forall P : Prop, P \ / ~ P).
   ----- *)

```

```

Definition tercio_excluso : Prop := forall P : Prop,
  P \ / ~ P.

```

```

(* -----
   Nota. La proposición tercio_excluso no es demostrable en Coq.
   ----- *)

```

```
(* -----
Ejemplo 5.3.2. Demostrar que
  forall (P : Prop) (b : bool),
    (P <-> b = true) -> P \ / ~ P.
----- *)
```

**Theorem** tercio\_exluso\_restringido :

```
forall (P : Prop) (b : bool),
  (P <-> b = true) -> P \ / ~ P.
```

**Proof.**

```
intros P [] H.
-
  (* P : Prop
   H : P <-> true = true
   =====
   P \ / ~ P *)

left.
  (* P *)
rewrite H.
  (* true = true *)
reflexivity.
-
  (* P : Prop
   H : P <-> false = true
   =====
   P \ / ~ P *)

right.
  (* ~ P *)
rewrite H.
  (* false <> true *)
intros H1.
  (* H1 : false = true
   =====
   False *)

inversion H1.
```

**Qed.**

```
(* -----
Ejemplo 5.3.3. Demostrar que
  forall (n m : nat),
    n = m \ / n <> m.
----- *)
```

**Theorem** tercio\_exluso\_restringido\_eq:

```
forall (n m : nat),
  n = m \ / n <> m.
```

**Proof.**

```

intros n m.                                (* n, m : nat
                                             =====
                                             n = m \/\ n <> m *)

apply (tercio_excluso_restringido
      (n = m)
      (iguales_nat n m)).                    (* n = m <-> iguales_nat n m = true *)
symmetry.                                (* iguales_nat n m = true <-> n = m *)
apply iguales_nat_bool_prop.
Qed.

```

```

(* -----
Notas.
1. En Coq no se puede demostrar el principio del tercio exclusivo.
2. Las demostraciones de las fórmulas existenciales tienen que
   proporcionar un testigo.
2. La lógica de Coq es constructiva.
----- *)

```

```

(* -----
Ejercicio 5.3.1. Demostrar que
  forall (P : Prop),
    ~ ~ (P \/\ ~ P).
----- *)

```

**Theorem** tercio\_excluso\_irrefutable:

```

forall (P : Prop),
  ~ ~ (P \/\ ~ P).

```

**Proof.**

```

intros P.  (* P : Prop
             =====
             ~ ~ (P \/\ ~ P) *)

unfold not. (* (P \/\ (P -> False) -> False) -> False *)
intros H.   (* H : P \/\ (P -> False) -> False
             =====
             False *)

apply H.   (* P \/\ (P -> False) *)
right.    (* P -> False *)
intro H1.  (* H1 : P
             =====

```

```

      False *)
apply H.      (* P \ / (P -> False) *)
left.        (* P *)
apply H1.
Qed.

```

```

(* -----
  Nota. El teorema anterior garantiza que añadir el tercio excluso como
  axioma no provoca contradicción.
  ----- *)

```

```

(* -----
  Ejercicio 5.3.2. Demostrar que
    tercio_excluso ->
    forall (X : Type) (P : X -> Prop),
      ~ (exists x, ~ P x) -> (forall x, P x).

  Nota. La condición del tercio_excluso es necesaria.
  ----- *)

```

```

Theorem no_existe_no:
  tercio_excluso ->
  forall (X : Type) (P : X -> Prop),
    ~ (exists x, ~ P x) -> (forall x, P x).

```

**Proof.**

```

intros H1 X P H2 x.      (* H1 : tercio_excluso
                           X : Type
                           P : X -> Prop
                           H2 : ~ (exists x : X, ~ P x)
                           x : X
                           =====
                           P x *)
unfold tercio_excluso in H1. (* H1 : forall P : Prop, P \ / ~ P *)
assert (P x \ / ~ P x).
-
  (* P x \ / ~ P x *)
  apply H1.
-
  (* H : P x \ / ~ P x
     =====
     P x *)
  destruct H as [H3 | H4].

```

```

+
(* x : X
   H3 : P x
   =====
   P x *)

apply H3.
+
(* x : X
   H4 : ~ P x
   =====
   P x *)

exfalso.
apply H2.
exists x.
apply H4.
Qed.

(* -----
   Ejercicio 5.3.1. En este ejercicio se van a demostrar 4 formas
   equivalentes del principio del tercio excluso.

   Sea peirce la proposición definida por
   Definition peirce: Prop := forall P Q : Prop,
     ((P -> Q) -> P) -> P.

   Demostrar que
   tercio_excluso <-> peirce
   ----- *)

Definition peirce: Prop := forall P Q : Prop,
  ((P -> Q) -> P) -> P.

Theorem tercio_excluso_peirce_L1:
  tercio_excluso -> peirce.
Proof.
  unfold tercio_excluso. (*
    =====
    (forall P : Prop, P \ / ~ P) -> peirce *)
  unfold peirce.
  (* (forall P : Prop, P \ / ~ P) ->
     forall P Q : Prop, ((P -> Q) -> P) -> P *)
  intros H1 P Q H2.
  (* H1 : forall P : Prop, P \ / ~ P
     P, Q : Prop

```



```

                                H2 : (P -> Q) -> P
                                =====
                                P *)

assert (P \/ ~P).
-
  apply H1.
-
                                (* H : P \/ ~ P
                                =====
                                P *)

destruct H as [H3 | H4].
+
                                (* H3 : P
                                =====
                                P *)

  apply H3.
+
                                (* H4 : ~ P
                                =====
                                P *)

  apply H2.
  intros H5.
                                (* P -> Q *)
                                (* H5 : P
                                =====
                                Q *)

  exfalso.
  apply H4.
  apply H5.
  Qed.

Theorem tercio_excluso_peirce_L2:
  peirce -> tercio_excluso.
Proof.
  unfold peirce.
                                (*
                                =====
                                (forall P Q : Prop, ((P -> Q) -> P) -> P) ->
                                tercio_excluso *)
  unfold tercio_excluso.
                                (* (forall P Q : Prop, ((P -> Q) -> P) -> P) ->
                                forall P : Prop, P \/ ~ P *)
  intros H P.
                                (* H : forall P Q : Prop, ((P -> Q) -> P) -> P
                                P : Prop
                                =====
                                P \/ ~ P *)
  apply H with (Q := False). (* (P \/ ~ P -> False) -> P \/ ~ P *)

```

```

intros H1.          (* H1 : P \ / ~ P -> False
                      =====
                      P \ / ~ P *)
right.             (* ~ P *)
unfold not.         (* P -> False *)
intros H2.          (* H2 : P
                      =====
                      False *)
apply H1.           (* P \ / ~ P *)
left.              (* P *)
apply H2.
Qed.

```

**Theorem** `tercio_excluso_equiv_peirce`:

`tercio_excluso <-> peirce.`

**Proof.**

**split.**

```

- (*
  =====
  tercio_excluso -> peirce *)
apply tercio_excluso_peirce_L1.
- (*
  =====
  peirce -> tercio_excluso *)
apply tercio_excluso_peirce_L2.

```

**Qed.**

```

(* -----
   Ejercicio 5.3.2. Sea eliminacion_doble_negacion la proposición
   definida por
       Definition eliminacion_doble_negacion: Prop := forall P : Prop,
         ~~P -> P.

   Demostrar que
       tercio_excluso <-> eliminacion_doble_negacion
   ----- *)

```

**Definition** `eliminacion_doble_negacion`: **Prop** := **forall** P : **Prop**,  
`~~P -> P.`

**Theorem** `tercio_excluso_equiv Eliminacion_doble_negacion_L1:`  
`tercio_excluso -> eliminacion_doble_negacion.`

**Proof.**

```

unfold tercio_excluso.
(*
=====
(forall P : Prop, P \/ ~ P) ->
  eliminacion_doble_negacion *)
unfold eliminacion_doble_negacion. (* (forall P : Prop, P \/ ~ P) ->
  forall P : Prop, ~ ~ P -> P *)
intros H1 P H2.
(* H1 : forall P : Prop, P \/ ~ P
   P : Prop
   H2 : ~ ~ P
=====
   P *)

assert (P \/ ~P).
-
  apply H1.
-
  (* H : P \/ ~ P
   =====
   P *)

destruct H as [H3 | H4].
+
  (* H2 : ~ ~ P
   H3 : P
   =====
   P *)

  apply H3.
+
  (* H4 : ~ P
   =====
   P *)

  exfalso.
  apply H2.
  apply H4.

```

**Qed.**

**Theorem** `tercio_excluso_equiv Eliminacion_doble_negacion_L2:`  
`eliminacion_doble_negacion -> tercio_excluso.`

**Proof.**

```

unfold eliminacion_doble_negacion. (*
=====

```

```

                                (forall P : Prop, ~ ~ P -> P) ->
                                tercio_excluso *)
unfold tercio_excluso.
                                (* (forall P : Prop, ~ ~ P -> P) ->
                                forall P : Prop, P \/ ~ P *)
intros H P.
                                (* H : forall P : Prop, ~ ~ P -> P
                                P : Prop
                                =====
                                P \/ ~ P *)
apply H.
                                (* ~ ~ (P \/ ~ P) *)
apply tercio_excluso_irrefutable.
Qed.

```

**Theorem** tercio\_excluso\_equiv Eliminacion\_doble\_negacion:

tercio\_excluso <-> eliminacion\_doble\_negacion.

**Proof.**

**split.**

-

**apply** tercio\_excluso\_equiv Eliminacion\_doble\_negacion\_L1.

-

**apply** tercio\_excluso\_equiv Eliminacion\_doble\_negacion\_L2.

**Qed.**

```

(* -----
Ejercicio 5.3.3. Sea morgan_no_no la proposición
definida por
    Definition de_morgan_no_no: Prop :=
        forall P Q : Prop, ~(~P /\ ~Q) -> P \/ Q.

    Demostrar que
        tercio_excluso <-> morgan_no_no
----- *)

```

**Definition** de\_morgan\_no\_no: **Prop** :=

**forall** P Q : **Prop**, ~(~P /\ ~Q) -> P \/ Q.

**Theorem** tercio\_excluso\_equiv\_de\_morgan\_no\_no\_L1:

tercio\_excluso -> de\_morgan\_no\_no.

**Proof.**

**unfold** tercio\_excluso. (\*

```

=====
(forall P : Prop, P \ / ~ P) ->
de_morgan_no_no *)
unfold de_morgan_no_no. (* (forall P : Prop, P \ / ~ P) ->
                           forall P Q : Prop, ~ (~ P /\ ~ Q) -> P \ / Q *)
intros H1 P Q H2. (* H1 : forall P : Prop, P \ / ~ P
                     P, Q : Prop
                     H2 : ~ (~ P /\ ~ Q)
=====
                     P \ / Q *)

assert (P \ / ~P).
- (* P \ / ~ P *)
  apply H1.
- (* H : P \ / ~ P
=====
    P \ / Q *)

destruct H as [H3 | H4].
+ (* H2 : ~ (~ P /\ ~ Q)
   H3 : P
=====
   P \ / Q *)

  left.
  apply H3.
+ (* P *)

  right.
  assert (Q \ / ~ Q).
* (* H4 : ~ P
=====
   P \ / Q *)
  (* Q *)

  apply H1.
* (* Q \ / ~ Q *)

  (* H : Q \ / ~ Q
=====
    Q *)

destruct H as [H5 | H6].
-- (* H5 : Q
=====
    Q *)

  apply H5.
-- (* H6 : ~ Q
=====

```

```

                                Q *)
exfalso.                        (* False *)
apply H2.                      (* ~ P /\ ~ Q *)
split.
++                              (* ~ P *)
    apply H4.
++                              (* ~ Q *)
    apply H6.

```

**Qed.**

**Theorem** `tercio_excluso_equiv_de_morgan_no_no_L2`:  
`de_morgan_no_no -> tercio_excluso.`

**Proof.**

```

unfold de_morgan_no_no. (*
=====
                                (forall P Q : Prop, ~ (~ P /\ ~ Q) -> P \/ Q) ->
                                tercio_excluso *)
unfold tercio_excluso.  (* (forall P Q : Prop, ~ (~ P /\ ~ Q) -> P \/ Q) ->
                                forall P : Prop, P \/ ~ P *)
intros H1 P.           (* H1 : forall P Q : Prop, ~ (~ P /\ ~ Q) -> P \/ Q
                                P : Prop
=====
                                P \/ ~ P *)
apply H1.              (* ~ (~ P /\ ~ ~ P) *)
intros H2.             (* H2 : ~ P /\ ~ ~ P
=====
                                False *)
destruct H2 as (H3,H4). (* H3 : ~ P
                                H4 : ~ ~ P
=====
                                False *)
apply H4.              (* ~ P *)
apply H3.

```

**Qed.**

**Theorem** `tercio_excluso_equiv_de_morgan_no_no`:  
`tercio_excluso <-> de_morgan_no_no.`

**Proof.**

**split.**

```

-
  apply tercio_excluso_equiv_de_morgan_no_no_L1.
-
  apply tercio_excluso_equiv_de_morgan_no_no_L2.
Qed.

```

```

(* -----
   Ejercicio 5.3.4. Sea condicional_a_disyuncion la proposición
   definida por
     Definition condicional_a_disyuncion: Prop :=
       forall P Q : Prop, (P -> Q) -> (~P \ / Q).

   Demostrar que
     tercio_excluso <-> morgan_no_no
   ----- *)

```

```

Definition condicional_a_disyuncion: Prop :=
  forall P Q : Prop, (P -> Q) -> (~P \ / Q).

```

```

Lemma tercio_excluso_equiv_condicional_a_disyuncion_L1:
  tercio_excluso -> condicional_a_disyuncion.

```

**Proof.**

```

  unfold tercio_excluso.
  (*
     =====
     (forall P : Prop, P \ / ~ P) ->
     condicional_a_disyuncion *)
  unfold condicional_a_disyuncion. (* (forall P : Prop, P \ / ~ P) ->
     forall P Q : Prop, (P -> Q) -> ~ P \ / Q *)
  intros H1 P Q H2.
  (* H1 : forall P : Prop, P \ / ~ P
     P, Q : Prop
     H2 : P -> Q
     =====
     ~ P \ / Q *)
  assert (P \ / ~P).
  -
    (* P \ / ~ P *)
    apply H1.
  -
    (* H : P \ / ~ P
     =====
     ~ P \ / Q *)
    destruct H as [H3 | H4].

```

```

+
(* H3 : P
=====
~ P \ / Q *)
right.
apply H2.
apply H3.
+
(* H4 : ~ P
=====
~ P \ / Q *)
left.
apply H4.
Qed.

Lemma tercio_excluso_equiv_condicional_a_disyuncion_L2:
  condicional_a_disyuncion -> tercio_excluso.
Proof.
  unfold condicional_a_disyuncion. (*
=====
(forall P Q:Prop, (P -> Q) -> ~ P \ / Q)
-> tercio_excluso *)
  unfold tercio_excluso. (* (forall P Q:Prop, (P -> Q) -> ~ P \ / Q)
-> forall P : Prop, P \ / ~ P *)
  intros H1 P. (* H1 : forall P Q : Prop,
(P -> Q) -> ~ P \ / Q
P : Prop
=====
P \ / ~ P *)
  apply disy_conmutativa. (* ~ P \ / P *)
  apply H1. (* P -> P *)
  intros. (* H : P
=====
P *)

  apply H.
Qed.

```

**Theorem** tercio\_excluso\_equiv\_condicional\_a\_disyuncion:  
 tercio\_excluso <-> condicional\_a\_disyuncion.

**Proof.**  
**split.**



- **apply** tercio\_excluso\_equiv\_condicional\_a\_disyuncion\_L1.
- **apply** tercio\_excluso\_equiv\_condicional\_a\_disyuncion\_L2.

**Qed.**

```
(* =====  
  § Bibliografía  
  ===== *)
```

```
(*  
  + "Logic in Coq" de Peirce et als. http://bit.ly/2nv1T9Z *)
```