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

Índice general

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.

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 conjuncion 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 (nanb 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 :=
  | O : 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
    | O    => O
    | 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 O)))).
   ------------------------------------------------------------------ *)

Check (S (S (S (S O)))).
(* ===> 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
    | O        => O
    | S O      => O
    | 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
  | O        => true
  | S O      => 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
    | O    => 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
    | O    => O
    | S n' => suma m (producto n' m)
  end.

Example producto1: (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
  | (O   , _)    => O
  | (S _ , O)    => 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
    | O   => S O
    | S m => mult x (potencia x m)
  end.

Compute (potencia 2 3).
(* ===> 8 : nat *)

(* ---------------------------------------------------------------------
   Ejercicio 1.6.1. Definir la función
      factorial : nat -> nat1
   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
  | O    => 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 me son iguales.
   ------------------------------------------------------------------ *)

Fixpoint iguales_nat (n m : nat) : bool :=
  match n with
  | O => match m with
         | O    => true
         | S m’ => false
         end
  | S n’ => match m with
            | O    => 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
  | O    => true
  | S n' => match m with
            | O    => 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_O_n : forall n : nat, 0 + n = n.
Proof.
  intros n.    (* n : nat
                  ============================
                   0 + n = n *)
  simpl.       (*  n = n *)
  reflexivity.
Qed.

(* 2ª demostración *)
Theorem suma_O_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_O_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 -> 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    => O
  | 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.                     (* n’ * 0 = 0 *)
    rewrite IHn’.              (* 0 = 0 *)
    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.                   (* m = m + 0 *)
    rewrite <- suma_n_0.     (* m = m *)
    reflexivity.
  +                          (* n', m : nat
                                IHn' : n' + m = m + n'
                                ============================
                                S n' + m = m + S n' *)
    simpl.                   (* S (n' + m) = m + S n' *)
    rewrite IHn'.            (* S (m + n') = m + S n' *)
    rewrite <- suma_n_Sm.    (* S (m + n') = S (m + n') *)
    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
                                ============================
```

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

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

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

Fixpoint doble (n:nat) :=
  match n with
  | O    => O
  | 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) *)
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) *)
    rewrite -> H.              (* m + n + (p + q) = m + n + (p + q) *)
    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 indución: 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 indución: 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:nat, true = iguales_nat n n.
```

```
   ---------------------------------------------------------------- *)

Theorem iguales_n_n: forall n : nat, true = iguales_nat n n.
Proof.
  intros n.                  (* n : nat
                                ============================
                                true = iguales_nat n n *)
  induction n as [|n' IHn'].
  -                          (*
                                ============================
                                true = iguales_nat 0 0 *)
    reflexivity.
  -                          (* n' : nat
                                IHn' : true = iguales_nat n' n'
                                ============================
                                true = iguales_nat (S n') (S n') *)
    simpl.                   (* true = iguales_nat n' n' *)
    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 indución: 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.                    (* S (n' * 1) = S n' *)
    rewrite IHn'.             (* S n' = S n' *)
    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.                    (* S (n' + 1) = S (S n') *)
    rewrite HIn'.             (* S (S n') = S (S n') *)
    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'].
  -                            (* m : nat
                                  ============================
                                  0 * (m + 1) = 0 * m + 0 *)
    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.                  (* menor_o_igual n m = true *)
    rewrite H.              (* true = true *)
    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.                  (* menor_o_igual (p' + n) (p' + m) = true *)
    rewrite HIp'.           (* true = true *)
    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.         (* 0 = 0 *)
    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 subojetivo.

   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
 *)
```