TEMA 5.- VERIFICACION FORMAL EN COQ

5.1.- Verificación de programas sencillos.

- Recordemos que validar consiste en garantizar que el software implementa una función específica.
- Para ello construímos una especificación del comportamiento deseado y chequeamos su conincidencia con el comportamiento real.
- Ejemplo: Suma.

```
Coq < Print plus.
plus =
fix plus (n m : nat) {struct n} : nat :=
  match n with
  | O => m
  | S p => S (plus p m)
  end
     : nat -> nat -> nat

Argument scopes are [nat_scope nat_scope]
```

- Contruímos una especificación, esto es, un predicado que nos devuelva `True` cuando un número sea la suma de otros dos. Lógicamente se tratará de un predicado inductivo.

```
Coq < Inductive suma:nat->nat->nat->Prop:=
Coq <   suma0: forall m:nat, suma O m m
Coq < | sumaS: forall m n p:nat, suma m n p -> suma (S m) n (S p).
suma is defined
suma_ind is defined
```

- Verificar nuestro programa de la suma (o, mejor dicho, el de Coq) es tan fácil como chequear con suma que para cualquier par de naturales, `plus` obtiene el resultado deseado.

```
Coq < Theorem testSuma: forall m n:nat, suma m n (plus m n).
1 subgoal

  ============================
   forall m n : nat, suma m n (m + n)
```

- Aquí tenemos una demostración como las que ya hemos realizado en el capítulo anterior para algunas propiedades de la suma. Aplicamos `Intros+Elim` o `Induction` para obtener dos nuevos objetivos: uno para el caso base de los naturales, el 0, y el otro para el paso inductivo de m a su sucesor $S(m)$.

```
testSuma < induction m.
2 subgoals

  ============================
   forall n : nat, suma 0 n (0 + n)

subgoal 2 is:
 forall n : nat, suma (S m) n (S m + n)
```

- Fijaos que `(0 + n) = (plus (0) n)` en el primer objetivo es uno de los casos definidos en `plus`. Podemos simplificar la expresión con `Simpl`.

```
testSuma < simpl.
2 subgoals

  ============================
   forall n : nat, suma 0 n n

subgoal 2 is:
 forall n : nat, suma (S m) n (S m + n)
```

- De ese modo obtenemos en el primer objetivo el caso del predicado suma definido por el constructor `suma0` que nos dice que $0 + m = m$. Podemos resolver el objetivo mediante la táctica `apply`.

```
testSuma < apply suma0.
1 subgoal

  m : nat
  IHm : forall n : nat, suma m n (m + n)
  ============================
   forall n : nat, suma (S m) n (S m + n)
```

- Nos queda por resolver el segundo objetivo, correspondiente al paso inductivo en la definición de los naturales. Primero eliminamos el producto con `intro`.

```
testSuma < intro.
1 subgoal

  m : nat
  IHm : forall n : nat, suma m n (m + n)
  n : nat
  ============================
   suma (S m) n (S m + n)
```

- Podemos simplificar `S m + n = (plus (S m) n)` mediante la definición de `plus`. Aplicamos `simpl`.

```
testSuma2 < simpl.
1 subgoal

  m : nat
  IHm : forall n : nat, suma m n (m + n)
  n : nat
  ============================
   suma (S m) n (S (m + n))
```

- Tenemos como hipótesis que `m+n=(plus m n)` y pretendemos demostrar que $S(m)+n$`=(S (plus n n0))`. Es decir, estamos demostrando el paso inductivo de la propiedad: si ésta es cierta para n, debe serlo para $S(n)$. Este paso inductivo está definido por el constructor `sumaS` de `suma`. Por lo tanto podemos usar `apply`.

```
testSuma < apply sumaS.
1 subgoal

  m : nat
  IHm : forall n : nat, suma m n (m + n)
  n : nat
  ============================
   suma m n (m + n)
```

- Nos queda un objetivo igual a una de la hipótesis, o, por lo menos, al consecuente del producto que es el tipo de la hipótesis. Volvemos a usar `apply`.

```
testSuma < apply IHm.
Proof completed.
```

- Como curiosidad, podemos definir nuestras funciones sobre naturales a través de `nat_rec`.

```
Coq < Print nat_rec.
nat_rec =
fun P : nat -> Set => nat_rect P
     : forall P : nat -> Set,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

- `prim_rec` es el operador de la recusión primitiva. Podemos simplificar su tipo con `Eval cbv beta in` que realiza la β-reducción sobre la expresión que se le pasa como argumento. El resultado es una función con 3 argumentos: un `nat` (el resultado para el 0), una función `nat ->nat ->nat` (el paso recursivo) y un segundo `nat` (el argumento sobre el que se realizará la definición por casos).

```
Coq < Definition prim_rec := nat_rec (fun i:nat => nat).
prim_rec is defined

Coq < Check prim_rec.
prim_rec
     : (fun _ : nat => nat) 0 ->
       (forall n : nat, (fun _ : nat => nat) n -> (fun _ : nat => nat) (S n)) ->
       forall n : nat, (fun _ : nat => nat) n

Coq < Eval cbv beta in
Coq < (fun _ : nat => nat) 0 ->
Coq < (forall n : nat, (fun _ : nat => nat) n -> (fun _ : nat => nat) (S n)) ->
Coq < forall n : nat, (fun _ : nat => nat) n.
     = nat -> (nat -> nat -> nat) -> nat -> nat
     : Set
```

- Por tanto, podríamos definir la función suma como:

```
Coq < Definition Suma (n m:nat) := prim_rec m (fun p rec:nat => S rec) n.
Suma is defined
```

- `(Suma n m)` sería, de este modo, una definición por casos sobre n. Cuando $n = 0$, obtenemos m. Si $n = S(p)$ obtenemos `S rec`, donde `rec` es el resultado del cálculo recursivo `(Suma p m)`.
- Utilizando el mismo predicado suma de la demostración anterior podríamos verificar el comportamiento de esta nueva función `Suma`, de forma muy similar.

```
Coq < Theorem testSuma: forall m n:nat, suma m n (Suma m n).
1 subgoal

  ============================
   forall m n : nat, suma m n (Suma m n)

testSuma2 < intros; elim m; [(simpl; apply suma0) |
testSuma2 <                  (intros; simpl; apply sumaS; assumption)].
Proof completed.
```

- Ejemplo II: El producto de naturales.

```
Coq < Print mult.
mult =
fix mult (n m : nat) {struct n} : nat :=
  match n with
  | O => O
  | S p => m + mult p m
  end
     : nat -> nat -> nat

Argument scopes are [nat_scope nat_scope]
```

- De nuevo, necesitamos una especificación para verificar su cumplimiento por parte de la función `mult`.

```
Coq < Inductive producto:nat->nat->nat->Prop :=
Coq <     producto0: forall n:nat, producto O n O
Coq <   | productoS: forall m n p:nat, producto m n p ->
Coq <                                 producto (S m) n (plus n p).
producto is defined
producto_ind is defined
```

- Verificamos la función como la demostración del siguiente teorema.

```
Coq < Theorem testProducto : forall m n:nat, producto m n (mult m n).
1 subgoal

  ============================
   forall m n : nat, producto m n (m * n)

testProducto < intros;elim m.
2 subgoals

  m : nat
  n : nat
  ============================
   producto 0 n (0 * n)

subgoal 2 is:
 forall n0 : nat, producto n0 n (n0 * n) -> producto (S n0) n (S n0 * n)
```

- Acabamos con el primer objetivo...

```
testProducto < simpl.
2 subgoals

  m : nat
  n : nat
  ============================
   producto 0 n 0

subgoal 2 is:
 forall n0 : nat, producto n0 n (n0 * n) -> producto (S n0) n (S n0 * n)

testProducto < apply producto0.
1 subgoal

  m : nat
  n : nat
  ============================
   forall n0 : nat, producto n0 n (n0 * n) -> producto (S n0) n (S n0 * n)
```

- ...y con el segundo. Aplicamos `simpl` y `apply`. La táctica `set` nor permite sustituír un término bien formado, en este caso `(mult n0 n)`, por un identificador, tanto en el objetivo como en las hipótesis, añadiendo la igualdad *identificador* `:=` *termino* como nueva hipótesis.

```
testProducto < intros.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : producto n0 n (n0 * n)
  ============================
   producto (S n0) n (S n0 * n)

testProducto < simpl.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : producto n0 n (n0 * n)
  ============================
   producto (S n0) n (n + n0 * n)

testProducto < set (p := mult n0 n) in *.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  p := n0 * n : nat
  H : producto n0 n p
  ============================
   producto (S n0) n (n + p)
```

```
testProducto < apply productoS.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  p := n0 * n : nat
  H : producto n0 n p
  ============================
   producto n0 n p

testProducto < assumption.
Proof completed.
```

- De nuevo podemos definir el producto utilizando `prim_rec`.

```
Definition Producto (n m:nat) := prim_rec 0 (fun p rec:nat => plus m rec) n.

Coq < Theorem testProducto2 : forall m n:nat, producto m n (Producto m n).
1 subgoal

  ============================
   forall m n : nat, producto m n (Producto m n)

testProducto2 < intros; elim m; [(simpl; apply producto0) |
testProducto2 <                   (intros; simpl; apply productoS; assumption)].
Proof completed.
```

- Ejemplo III.- la potencia de números naturales.

```
Coq < Fixpoint pot (m n:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | 0 => S 0
Coq <   | S p => mult m (pot m p)
Coq <   end.

Coq < Eval compute in (pot (S (S (S 0))) 0).
     = 1
     : nat
```

```
Coq < Eval compute in (pot (S (S (S O))) (S (S O))).
     = 9
     : nat
```

- Definimos la especificación que se tiene que cumplir.

```
Coq < Inductive potencia : nat->nat->nat->Prop :=
Coq <     potencia0: forall m:nat, potencia m O (S O)
Coq <   | potenciaS: forall m n p:nat, potencia m n p
Coq <                                 -> potencia m (S n) (mult m p).
potencia is defined
potencia_ind is defined
```

- Y demostramos que la función la cumple.

```
Coq < Theorem testPotencia : forall m n:nat, potencia m n (pot m n).
1 subgoal

  ============================
   forall m n : nat, potencia m n (pot m n)
```

- Aplicamos `intros` y `elim`.

```
testPotencia < intros.
1 subgoal

  m : nat
  n : nat
  ============================
   potencia m n (pot m n)

testPotencia < elim n.
2 subgoals

  m : nat
  n : nat
  ============================
   potencia m 0 (pot m 0)

subgoal 2 is:
 forall n0 : nat,
 potencia m n0 (pot m n0) -> potencia m (S n0) (pot m (S n0))
```

- Aplicamos `Simpl` y `potencia0` para demostrar el primer objetivo...

```
testPotencia < simpl.
2 subgoals

  m : nat
  n : nat
  ============================
   potencia m 0 1

subgoal 2 is:
 forall n0 : nat,
 potencia m n0 (pot m n0) -> potencia m (S n0) (pot m (S n0))

testPotencia < apply potencia0.
1 subgoal

  m : nat
  n : nat
  ============================
   forall n0 : nat,
   potencia m n0 (pot m n0) -> potencia m (S n0) (pot m (S n0))
```

- ... y demostramos el segundo con el mismo procedimiento.

```
testPotencia < intros.
1 subgoal
  m : nat
  n : nat
  n0 : nat
  H : potencia m n0 (pot m n0)
  ============================
   potencia m (S n0) (pot m (S n0))

testPotencia < simpl.
1 subgoal
  m : nat
  n : nat
  n0 : nat
  H : potencia m n0 (pot m n0)
  ============================
   potencia m (S n0) (m * pot m n0)
```

```
testPotencia < apply potenciaS.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : potencia m n0 (pot m n0)
  ============================
   potencia m n0 (pot m n0)

testPotencia < assumption.
Proof completed.
```

5.2.- Generación de programas mediante pruebas.

- También se puede generar código a partir de una prueba. Para ello, el primer paso es generar una especificación, tal y como se ha visto en el apartado anterior.

```
Coq < Inductive producto:nat->nat->nat->Prop :=
Coq <     producto0: forall n:nat, producto 0 n 0
Coq <   | productoS: forall m n p:nat, producto m n p ->
Coq <                                 producto (S m) n (plus n p).
producto is defined
producto_ind is defined
```

- La contrucción del código se produce a partir de la demostración del siguiente teorema:

```
Coq < Theorem Prod : forall n m:nat, {p:nat | producto n m p}.
```

- Las llaves son el equivalente del ∃ para el tipo `Set`.

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

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

- Dado que se trata de una demostración sobre los naturales, comenzamos con el habitual `intros + elim`.

```
Prod < intros;elim n.
2 subgoals

  n : nat
  m : nat
  ============================
   {p : nat | producto 0 m p}

subgoal 2 is:
 forall n0 : nat,
 {p : nat | producto n0 m p} -> {p : nat | producto (S n0) m p}
```

- Para el sacar el $\exists$ del objetivo podemos utilizar el constructor del tipo `sig`, `exist`. Hacemos `apply exist with 0`. El `with 0` es necesario porque de no usarlo el intérprete no sería capaz de instanciar `p`.

```
Prod < apply exist with 0.
2 subgoals

  n : nat
  m : nat
  ============================
   producto 0 m 0

subgoal 2 is:
 forall n0 : nat,
 {p : nat | producto n0 m p} -> {p : nat | producto (S n0) m p}
```

- También se puede usar la táctica `exist` o `Constructor 1`, que hacen exactamente lo mismo.

```
Prod < exists 0.
2 subgoals

  n : nat
  m : nat
  ============================
   producto 0 m 0

subgoal 2 is:
 forall n0 : nat,
 {p : nat | producto n0 m p} -> {p : nat | producto (S n0) m p}
```

- Dado que nos queda una expresión concordante con uno de los constructores del predicado producto, podemos utilizar `apply` o `constructor` para demostrar el primer objetivo.

```
Prod < apply producto0.
1 subgoal

  n : nat
  m : nat
  ============================
   forall n0 : nat,
   {p : nat | producto n0 m p} -> {p : nat | producto (S n0) m p}
```

- Lo que queremos obtener ahora es un objetivo sobre al que podamos aplicar el segundo constructor del predicado `producto`. Tenemos de nuevo una expresión fundada en la definición inductiva del ∃. Pero no podemos aplicar `elim exist` o `exists`, dado que no sabríamos a qué instanciar `p`. Nuestra unica alternativa es un elim usando H.

```
Prod < intros.
1 subgoal
  n : nat
  m : nat
  n0 : nat
  H : {p : nat | producto n0 m p}
  ============================
   {p : nat | producto (S n0) m p}
```

```
Prod < elim H.
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : {p : nat | producto n0 m p}
  ============================
   forall x : nat, producto n0 m x -> {p0 : nat | producto (S n0) m p0}
```

- Eliminamos los productos con `intros`.

```
Prod < intros.
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : {p : nat | producto n0 m p}
  x : nat
  p : producto n0 m x
  ============================
   {p0 : nat | producto (S n0) m p0}
```

- Ahora sí que sabemos a qué instanciar `p0`. Tenemos que $n0 * m = x$ es cierto, por lo que, siguiendo la definición del constructor `productoS`, $S(n0)*m = m+x$ es cierto, lo que expresaríamos como `(producto (S n0) m (plus m x))`, con lo que podemos aplicar `exists` dando a `p0` el valor `(plus m x)`.

```
Prod < exists (plus m x).
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : {p : nat | producto n0 m p}
  x : nat
  p : producto n0 m x
  ============================
   producto (S n0) m (m + x)
```

- Por fin hemos obtenido algo sobre lo que podemos aplicar el segundo constructor de `producto`.

```
Prod < apply productoS.
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : {p : nat | producto n0 m p}
  x : nat
  p : producto n0 m x
  ============================
   producto n0 m x

Prod < assumption.
Proof completed.
```

- ¿Por qué es necesario utilizar la versión del ∃ para `Set`. Porque queremos generar código, es decir, una función sobre el dominio de `Set`. Podemos ver como lo obtenido es el resultado de una aplicación de `nat_rec`.

```
Coq < Print Prod.
Prod =
fun n m : nat =>
nat_rec (fun n0 : nat => {p : nat | producto n0 m p})
  (exist (fun p : nat => producto 0 m p) 0 (producto0 m))
  (fun (n0 : nat) (H : {p : nat | producto n0 m p}) =>
   sig_rec
     (fun _ : {p : nat | producto n0 m p} => {p : nat | producto (S n0) m p})
     (fun (x : nat) (p : producto n0 m x) =>
      exist (fun p0 : nat => producto (S n0) m p0)
        (m + x) (productoS n0 m x p)) H) n
     : forall n m : nat, {p : nat | producto n m p}

Argument scopes are [nat_scope nat_scope]
```

- Si hubiśemos usado el ∃ convencional sobre `Prop`, hubiésemos obtenido algo sobre el dominio de `Prop`, bajo `nat_ind`.

```
Coq < Print Prod2.
Prod2 =
fun n m : nat =>
nat_ind (fun n0 : nat => exists p : nat, producto n0 m p)
  (ex_intro (fun p : nat => producto 0 m p) 0 (producto0 m))
  (fun (n0 : nat) (H : exists p : nat, producto n0 m p) =>
   ex_ind
     (fun (x : nat) (H0 : producto n0 m x) =>
      ex_intro (fun p : nat => producto (S n0) m p)
        (m + x) (productoS n0 m x H0)) H) n
     : forall n m : nat, exists p : nat, producto n m p

Argument scopes are [nat_scope nat_scope]
```

- Lo primero nos permite generar código. Lo segundo no.

```
Coq < Extraction Prod.
(** val prod : nat -> nat -> nat sig0 **)

let rec prod n m =
  match n with
    | O -> O
    | S n0 -> plus m (prod n0 m)

Coq < Extraction Prod2.
(** val prod2 : __ **)

let prod2 =
  __
```

- Otro ejemplo de generacion de codigo es el de la division entera.

```
Coq < Lemma div_euclid: forall a b:nat,
Coq <     {q:nat*nat | a=(plus (snd q) (mult (fst q) (S b))) /\
Coq <                    (lt (snd q) (S b))}.
1 subgoal

  ============================
   forall a b : nat, {q : nat * nat | a = snd q + fst q * S b /\ snd q < S b}
```

- Varias aclaraciones:

  `nat*nat` es el producto cartesiano de los naturales por los naturales. Por lo tanto, `q:nat*nat` es un par de dos numeros naturales (`nat`,`nat`).

  `(fst q)` y `(snd q)` son, respectivamente, el primer y segundo elemento del par `q`. `b` no representa el divisor, sino el predecesor del divisor. se previene así el caso de la division por cero.

- Por lo tanto, el lema dice que

$$\forall a, b \in nat, \exists q = (t, r) \in nat \times nat / (a = t * S(b) + r) \wedge (r < S(b))$$

  donde a es el dividendo, $S(b)$ es el divisor y en el par q tenemos como primer elemento el cociente t y como segundo el resto r.

- Aplicamos `Intros`.

```
div_euclid < intros.
1 subgoal

  a : nat
  b : nat
  ============================
   {q : nat * nat | a = snd q + fst q * S b /\ snd q < S b}
```

- Nuestro objetivo aparece basado en la definición inductiva de los naturales. Obtenemos una demostracion por casos con `elim`.

```
div_euclid < elim a.
2 subgoals

  a : nat
  b : nat
  ============================
   {q : nat * nat | 0 = snd q + fst q * S b /\ snd q < S b}

subgoal 2 is:
 forall n : nat,
 {q : nat * nat | n = snd q + fst q * S b /\ snd q < S b} ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Aplicamos el constructor del $\exists$ con `exists`, equivalente a `apply exist`. Instanciamos `q` a `(0,0)`.

```
div_euclid < exists (0,0).
2 subgoals

  a : nat
  b : nat
  ============================
   0 = snd (0, 0) + fst (0, 0) * S b /\ snd (0, 0) < S b

subgoal 2 is:
 forall n : nat,
 {q : nat * nat | n = snd q + fst q * S b /\ snd q < S b} ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Simplificamos.

```
div_euclid < simpl.
2 subgoals

  a : nat
  b : nat
  ============================
   0 = 0 /\ 0 < S b

subgoal 2 is:
 forall n : nat,
 {q : nat * nat | n = snd q + fst q * S b /\ snd q < S b} ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Resolvemos el primer objetivo con `auto with arith`.

```
div_euclid < Require Arith.
div_euclid < info auto with arith.
 == apply conj.
    apply refl_equal.

    change 1 <= S b; apply Gt.gt_le_S; change 0 < S b;
    apply Lt.lt_0_Sn.
```

```
1 subgoal

  a : nat
  b : nat
  ============================
   forall n : nat,
   {q : nat * nat | n = snd q + fst q * S b /\ snd q < S b} ->
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- `auto` hace uso de dos teoremas qu deben cargarse con la librería `Arith`.

```
Lemma gt_le_S : forall n m, m > n -> S n <= m.

Theorem lt_0_Sn : forall n, 0 < S n.
```

- Intros.

```
div_euclid < intros.
1 subgoal

  a : nat
  b : nat
  n : nat
  H : {q : nat * nat | n = snd q + fst q * S b /\ snd q < S b}
  ============================
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Aplicamos `elim` para poder resolver el paso inductivo de los naturales.

```
div_euclid < elim H.
1 subgoal

  a : nat
  b : nat
  n : nat
  H : {q : nat * nat | n = snd q + fst q * S b /\ snd q < S b}
  ============================
   forall x : nat * nat,
   n = snd x + fst x * S b /\ snd x < S b ->
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- `elim` nos introduce nuevos productos. Los pasamos a las hipotesis con `intros`.

```
div_euclid < intros.
1 subgoal
  a : nat
  b : nat
  n : nat
  H : {q : nat * nat | n = snd q + fst q * S b /\ snd q < S b}
  x : (nat * nat)%type
  p : n = snd x + fst x * S b /\ snd x < S b
  ============================
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Todavía podemos aplicar `elim` otra vez con la nueva hipótesis generada.

```
div_euclid < elim p.
1 subgoal
  a : nat
  b : nat
  n : nat
  H : {q : nat * nat | n = snd q + fst q * S b /\ snd q < S b}
  x : (nat * nat)%type
  p : n = snd x + fst x * S b /\ snd x < S b
  ============================
   n = snd x + fst x * S b ->
   snd x < S b -> {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Nuevos productos, nuevos `intros`.

```
div_euclid < intros.
1 subgoal
  a : nat
  b : nat
  n : nat
  H : {q : nat * nat | n = snd q + fst q * S b /\ snd q < S b}
  x : (nat * nat)%type
  p : n = snd x + fst x * S b /\ snd x < S b
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  ============================
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Simplificamos un poco el problema para hacerlo mas legible.

```
div_euclid < clear p H.
1 subgoal

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  ============================
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Y ahora un truco. Hacemos un `elim` con uno de los teoremas ya demostrados de `Arith`, `lt_eq_lt_dec`.

```
div_euclid < Require Compare_dec.

div_euclid < Check Compare_dec.lt_eq_lt_dec.
Compare_dec.lt_eq_lt_dec
     : forall n m : nat, {n < m} + {n = m} + {m < n}
```

- Este teorema dice que $\forall n, m \in nat (n < m) \vee (n = m) \vee (m < n)$ Lo aplicamos.

```
div_euclid < elim Compare_dec.lt_eq_lt_dec with (S (snd x)) (S b).
2 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  ============================
   {S (snd x) < S b} + {S (snd x) = S b} ->
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}

subgoal 2 is:
 S b < S (snd x) ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Y eliminamos el producto del primer objetivo.

```
div_euclid < intro.
2 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  a0 : {S (snd x) < S b} + {S (snd x) = S b}
  ============================
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}

subgoal 2 is:
 S b < S (snd x) ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- ¿Parecía complicado? Pues un poco más.

```
div_euclid < elim a0.
3 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  a0 : {S (snd x) < S b} + {S (snd x) = S b}
  ============================
   S (snd x) < S b ->
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}

subgoal 2 is:
 S (snd x) = S b ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
subgoal 3 is:
 S b < S (snd x) ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Estos dos últimos pasos nos han permitido, generar tres objetivos en los que se intenta cubrir los tres casos del teorema `lt_eq_lt_dec`. Como en el primer objetivo tenemos un producto, hacemos `intro`.

```
div_euclid < intro.
3 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  a0 : {S (snd x) < S b} + {S (snd x) = S b}
  a1 : S (snd x) < S b
  ============================
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}

subgoal 2 is: ...
subgoal 3 is: ...
```

- Eliminamos `a0` con un `clear` y hacemos un `split`. Se da valor a `q` buscando la coincidencia con `H0`.

```
div_euclid < split with ((fst x), (S (snd x))).
3 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  a0 : {S (snd x) < S b} + {S (snd x) = S b}
  a1 : S (snd x) < S b
  ============================
   S n = snd (fst x, S (snd x)) + fst (fst x, S (snd x)) * S b /\
   snd (fst x, S (snd x)) < S b

subgoal 2 is: ...
subgoal 3 is: ...
```

- Aplicamos `simpl` para eliminar los `fst` y `snd` referenciando a pares.

```
div_euclid < simpl.
3 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  a0 : {S (snd x) < S b} + {S (snd x) = S b}
  a1 : S (snd x) < S b
  ============================
   S n = S (snd x + fst x * S b) /\ S (snd x) < S b

subgoal 2 is: ...
```

- Hacemos `split` para eliminar el $\wedge$ generando dos nuevos subobjetivos, y `auto` para resolver el primero usando `H0`.

```
div_euclid < split; info auto.
 == apply (f_equal (A:=nat)); exact H0.

 == exact a1.

2 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  a0 : {S (snd x) < S b} + {S (snd x) = S b}
  ============================
   S (snd x) = S b ->
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}

subgoal 2 is:
 S b < S (snd x) ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Hemos, pues, demostrado el primero de los objetivos que representaban los casos particulares del teorema `lt_eq_lt_dec`. Fijaos que al demostrar este primer objetivo, la hipótesis `a0` vuelve a aparecer. La eliminamos otra vez con `clear` y hacemos `intros` para eliminar el producto del primer objetivo.

```
div_euclid < clear a0;intros.
2 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  b0 : S (snd x) = S b
  ============================
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}

subgoal 2 is:
 S b < S (snd x) ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Aplicamos `split` y `simpl` como en el primer caso particular de `lt_eq_lt_dec`.

```
div_euclid < split with ((S (fst x)), 0); simpl.
2 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  b0 : S (snd x) = S b
  ============================
   S n = S (b + fst x * S b) /\ 0 < S b

subgoal 2 is:
 S b < S (snd x) ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Con un nuevo `split` eliminamos el $\wedge$.

```
div_euclid < split.
3 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  b0 : S (snd x) = S b
  ============================
   S n = S (b + fst x * S b)

subgoal 2 is:
 0 < S b
subgoal 3 is:
 S b < S (snd x) ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Usamos la táctica `Omega` para resolver el primer objetivo. `Omega` resuelve ecuaciones e inecuaciones involucrando conectivas lógicas mediante aritmética de Pressburger.

```
div_euclid < omega.
2 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  b0 : S (snd x) = S b
  ============================
   0 < S b

subgoal 2 is:
 S b < S (snd x) ->
 {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Demostramos `0 <S b` mediante `auto`.

```
div_euclid < info auto with arith.
 == change 1 <= S b; apply Gt.gt_le_S; change 0 < S b;
    change 1 <= S b; apply Lt.lt_le_S; apply Lt.lt_0_Sn.

1 subgoal

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  ============================
   S b < S (snd x) ->
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- De hecho, incluso podríamos habernos ahorrado el `split` y `simpl` anterior. Omega podría haber resuelto el objetivo.

```
S n = S (b + fst x * S b) /\ 0 < S b
```

- Pero volvamos a donde estabamos en la demostración. Debemos aplicar `intros` para deshacer el producto del objetivo.

```
div_euclid < intros.
1 subgoal

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  b0 : S b < S (snd x)
  ============================
   {q : nat * nat | S n = snd q + fst q * S b /\ snd q < S b}
```

- Es interesante observar que `H1` y `b0` son contradictorias. Aplicamos `absurd`.

```
div_euclid < absurd (lt b (snd x)).
2 subgoals

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  b0 : S b < S (snd x)
  ============================
   ~ b < snd x

subgoal 2 is:
 b < snd x
```

- Nos libramos del primer objetivo con `Omega`.

```
div_euclid < omega.
1 subgoal

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  b0 : S b < S (snd x)
  ============================
   b < snd x
```

- Y terminamos con el último objetivo usando el teorema de la librería `Arith`.

```
div_euclid < Check Lt.lt_S_n.
Lt.lt_S_n
     : forall n m : nat, S n < S m -> n < m
```

```
div_euclid < apply Lt.lt_S_n.
1 subgoal

  a : nat
  b : nat
  n : nat
  x : (nat * nat)%type
  H0 : n = snd x + fst x * S b
  H1 : snd x < S b
  b0 : S b < S (snd x)
  ============================
   S b < S (snd x)

div_euclid < assumption.
Proof completed.
```

- Y extraemos el código de la función.

```
Coq < Extraction div_euclid.
(** val div_euclid : nat -> nat -> (nat, nat) prod sig0 **)

let rec div_euclid a b =
  match a with
    | O -> Pair (O, O)
    | S n ->
        let h = div_euclid n b in
        (match lt_eq_lt_dec (S (snd h)) (S b) with
           | Inleft x ->
               (match x with
                  | Left -> Pair ((fst h), (S (snd h)))
                  | Right -> Pair ((S (fst h)), O))
           | Inright -> assert false (* absurd case *))
```

- Coq también tiene la capacidad de exportar el código generado a Haskell y Caml. Para el ejemplo del producto:

```
Coq < Extraction Language Haskell.
Coq < Extraction "producto" Prod.
Coq < Extraction Language Ocaml.
Coq < Extraction "producto" Prod.
```

- El resultado en Haskell sería un fichero "producto.hs"

```
module Producto where

import qualified Prelude

__ = Prelude.error "Logical or arity value used"

data Nat = O
         | S Nat

nat_rect f f0 n =
  case n of
     O -> f
     S n0 -> f0 n0 (nat_rect f f0 n0)

nat_rec f f0 n =
  nat_rect f f0 n

type Sig a = a
  -- singleton inductive, whose constructor was exist

sig_rect f s =
  f s __

sig_rec f s =
  sig_rect f s

plus n m =
  case n of
    O -> m
    S p -> S (plus p m)

prod n m =
  nat_rec O (\n0 h -> sig_rec (\x _ -> plus m x) h) n
```

- En OCaml, obtendríamos dos ficheros: uno de interfaces "producto.mli"...

```
type nat =
  |O
  | S of nat

type 'a sig0 = 'a
  (* singleton inductive, whose constructor was exist *)

val plus : nat -> nat -> nat

val prod : nat -> nat -> nat sig0
```

- ... Y otro con el código "producto.ml".

```
type nat =
    |O
    | S of nat

type 'a sig0 = 'a
    (* singleton inductive, whose constructor was exist *)

(** val plus : nat -> nat -> nat **)

let rec plus n m =
    match n with
      |O -> m
      |S p -> S (plus p m)

(** val prod : nat -> nat -> nat sig0 **)

let rec prod n m =
    match n with
      |O -> O
      |S n0 -> plus m (prod n0 m)
```

- En el curso de demostraciones implicando la generación o verificación de código, es muchas veces necesario demostrar la decidibilidad de algunos operadores. Vamos a ver como se hace en el caso de la igualdad.

```
Coq < Lemma iseq: forall x y : nat, {x=y}+{~x=y}.
1 subgoal

  ============================
   forall x y : nat, {x = y} + {x <> y}
```

- La notacion + hace referencia al *or* entre objetos del tipo `Set`, en oposicion a $\vee$, que es un *or* entre objetos del tipo `Prop`.

```
iseq < Print sum.
Inductive sum (A : Set) (B : Set) : Set :=
    inl : A -> A + B | inr : B -> A + B
For inl: Argument A is implicit
For inr: Argument B is implicit
For sum: Argument scopes are [type_scope type_scope]
For inl: Argument scopes are [type_scope type_scope _]
For inr: Argument scopes are [type_scope type_scope _]
```

- Al tratarse de una definición sobre `nat`, aplicamos `Induction`.

```
iseq < induction x.
2 subgoals

  ============================
   forall y : nat, {0 = y} + {0 <> y}

subgoal 2 is:
 forall y : nat, {S x = y} + {S x <> y}

iseq < intro.
2 subgoals

  y : nat
  ============================
   {0 = y} + {0 <> y}

subgoal 2 is:
 forall y : nat, {S x = y} + {S x <> y}
```

- Usamos `case` para reescribir `y`, que es un natural, en atención a las dos posibles construcciones que puede adoptar: `O o (S n)`. `case` realiza la reescritura de términos con tipo inductivo, siendo casi equivalente a `elim`.

```
iseq < case y.
3 subgoals

  y : nat
  ============================
   {0 = 0} + {0 <> 0}

subgoal 2 is:
 forall n : nat, {0 = S n} + {0 <> S n}
subgoal 3 is:
 forall y : nat, {S x = y} + {S x <> y}
```

- Dado que el objetivo es un *or* sólo es necesario demostrar uno de los dos términos del *or*. Usamos `left` para quedarnos con el de la izquierda.

```
iseq < left.
3 subgoals

  y : nat
  ============================
   0 = 0

subgoal 2 is:
 forall n : nat, {0 = S n} + {0 <> S n}
subgoal 3 is:
 forall y : nat, {S x = y} + {S x <> y}

iseq < trivial.
2 subgoals

  y : nat
  ============================
   forall n : nat, {0 = S n} + {0 <> S n}

subgoal 2 is:
 forall y : nat, {S x = y} + {S x <> y}
```

- Usamos `intro` para eliminar el producto y `right` para quedarnos con un único término del *or* en el objetivo. Con `trivial` demostramos `(0)=(S n)`, gracias a los teoremas cargados con las librerías iniciales.

```
iseq < intro.
2 subgoals

  y : nat
  n : nat
  ============================
   {0 = S n} + {0 <> S n}

subgoal 2 is:
 forall y : nat, {S x = y} + {S x <> y}

iseq < right.
2 subgoals

  y : nat
  n : nat
  ============================
   0 <> S n

subgoal 2 is:
 forall y : nat, {S x = y} + {S x <> y}

iseq < trivial.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  ============================
   forall y : nat, {S x = y} + {S x <> y}
```

- Eliminamos los productos con `intros` y `case` se usa sobre el objetivo {`S x = y`} + {`S x <>y`} para sustituir `y` por los dos casos posibles en la construcción de un natural.

```
iseq < intros.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  ============================
   {S x = y} + {S x <> y}

iseq < case y.
2 subgoals

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  ============================
   {S x = 0} + {S x <> 0}

subgoal 2 is:
 forall n : nat, {S x = S n} + {S x <> S n}
```

- Con `Right` nos quedamos con el término derecho del *or*, que es el que podemos demostrar.

```
iseq < right.
2 subgoals

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  ============================
   S x <> 0

subgoal 2 is:
 forall n : nat, {S x = S n} + {S x <> S n}
```

- Terminamos con el primer subobjetivo con `auto`.

```
iseq < auto.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  ============================
   forall n : nat, {S x = S n} + {S x <> S n}
```

- Eliminamos el producto del segundo subobjetivo con `intro`.

```
iseq < intro.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  ============================
   {S x = S n} + {S x <> S n}
```

- Aplicamos `case` sobre {S x = S n}+{S x <>S n}. En este punto, buscamos nuevas hipótesis que reflejen esas mismas igualdades o términos similares. Para ello, el `case` se realiza sobre la hipótesis `IHx` sustituyendo `y` por `n`. De este modo obtenemos dos nuevos objetivos, que son productos cuyo consecuente es el objetivo anterior con `y` sustituido por `n`, y cuyos antecedentes son los términos del *or* de la hipótesis `IHx`.

```
iseq < case (IHx n).
2 subgoals

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  ============================
   x = n -> {S x = S n} + {S x <> S n}

subgoal 2 is:
 x <> n -> {S x = S n} + {S x <> S n}
```

- Eliminamos con *intro* el producto del primer objetivo. Obtenemos como nuevo subobjetivo un *or*. Con la hipótesis *e* diciendo que `x=n`, el término del *or* con el que debemos quedarnos es el izquierdo. Aplicamos `left`.

```
iseq < intro.
2 subgoals

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  e : x = n
  ============================
   {S x = S n} + {S x <> S n}

subgoal 2 is:
 x <> n -> {S x = S n} + {S x <> S n}

iseq < left.
2 subgoals

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  e : x = n
  ============================
   S x = S n

subgoal 2 is:
 x <> n -> {S x = S n} + {S x <> S n}
```

- Demostramos el primer objetivo con `auto`.

```
iseq < auto.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  ============================
   x <> n -> {S x = S n} + {S x <> S n}
```

- El segundo subobjetivo podrá ser demostrado del mismo modo que el primero. Pero como sería incluso mas aburrido que lo habitual, utilizamos la definición de `not`.

```
iseq < unfold not.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  ============================
   (x = n -> False) -> {S x = S n} + {S x = S n -> False}

iseq < intro.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  n0 : x = n -> False
  ============================
   {S x = S n} + {S x = S n -> False}
```

- Usamos `right` para quedarnos con el segundo término del *or* y `apply` para obtener un objetivo `x=n`, que demostramos con `auto` gracias a la hipótesis `H`.

```
iseq < right.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  n0 : x = n -> False
  ============================
   S x = S n -> False
```

```
iseq < intro.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  n0 : x = n -> False
  H : S x = S n
  ============================
   False

iseq < apply n0.
1 subgoal

  x : nat
  IHx : forall y : nat, {x = y} + {x <> y}
  y : nat
  n : nat
  n0 : x = n -> False
  H : S x = S n
  ============================
   x = n

iseq < info auto.
 == apply eq_add_S; exact H.

Proof completed.
```