TEMA 1.- INTRODUCCIÓN

- Objetivo: contruír programas libres de errores.
- Partiendo de un conjunto de especificaciones, queremos obtener un programa *correcto* y *completo*.
  - Correcto: todo lo que hace está contenido en las especificaciones.
  - Completo: sólo hace lo permitido por las especificaciones.
- ¿Cómo averiguar si un programa es correcto y completo? Dos maneras:
  - Prueba de programas
  - Verificación formal
- Prueba de programas: se construye un conjunto de casos de prueba y se comprueba si el funcionamiento del programa es el esperado.
  - Orientado a la búsqueda de errores.
  - No puede garantizar la no existencia de errores.
- Verificación formal:
  - La especificación del programa se expresa de forma precisa en notación matemática (lógica).
  - De este modo, el programa tiene un significado matemático definido.
  - Se puede demostrar matemáticamente que el programa satisface las especificaciones.

  Problemas:
  - No siempre se puede garantizar la equivalencia exacta de un programa que se ejecuta en una máquina con un modelo matemático abstracto.
  - La especificación matemática debe chequearse cuidadosamente para gqrantizar que se corresponde con la real.
- En el ámbito de las matemáticas sobre ordenador existen dos comunidades:
  - Cálculo y computación
    - Herramientas *CAS* (Computer Algebra Systems).
    - Grandas cantidades de cálculos complejos.
    - Maple, Mathematica, Matlab...
  - Herramientas de prueba automática
    - Harramientas simbólicas de demostración.

    - Todavía muy dependientes de la intervención humana.
    - Nqhtm, Coq, Nuprl, PVS...

- En esta asignatura estudiaremos el contexto llamado *Coq*, que consiste en una implementación del *Cálculo de Construcciones Inductivas* (CCI).
    - Desarrollado en el INRIA.
    - Demostrador automático de teoremas conducido por tácticas.
    - Tiene su propio lenguaje de especificación: *Gallina*.
    - Aunque es una herramienta de demostración, también tiene cierta flexibilidad en el ámbito del cálculo y computación.

TEMA 2.- CÁLCULO DE CONSTRUCCCIONES INDUCTIVAS

2.1.- INTRODUCCION

- El lenguaje formal utilizado por Coq es el Cálculo de Construcciones Inductivas (CIC o CCI). En CCI todos los objetos tienen un *tipo*. Las proposiciones o fórmulas a probar se representan como tipos, y las pruebas son términos que tienen ese tipo.
- Esta forma de ver las cosas es consecuencia del isomorfismo de Curry-Howard:

$$\frac{pruebas}{proposiciones\ o\ formulas} = \frac{objetos}{tipos}$$

- Es decir, para demostrar proposiciones debemos buscar unas pruebas. En CCI debemos encontrar objetos de nuestro lenguaje de tipo la proposición.

  Ej.-

$$\frac{p : A \longrightarrow B\ ,\ q : A}{(pq) : B}$$

  se puede leer

  - Si p es una prueba $A \longrightarrow B$ y q es una prueba de A, entoces p aplicado a q es de tipo B.
  - Para encontrar un objeto de tipo B, basta con encontrar un objeto de tipo $A \longrightarrow B$ y otro de tipo A.

  En CCI, todo, absolutamente todo tiene tipo. Hay tipos para funciones, tipos atómicos, tipos para las pruebas y tipos para los tipos.

2.2.- TÉRMINOS

- En la mayoría de las teorías de tipos, se realiza una distinción sintáctica entre *tipos* y *términos*. En CIC no.
- Tipos y términos se definen usando la misma estructura sintáctica.
- Por ejemplo, el tipo de las funciones puede tener varios significados. Supongamos que `nat` es el tipo de los números naturales. Entonces,

$$\texttt{nat} \longrightarrow \texttt{nat}$$

  es el tipo de las funciones que llevan de un natural a un natural.

$$\texttt{nat} \longrightarrow \texttt{Prop}$$

es el tipo de los predicados unarios sobre los números naturales. Por ejemplo,

$$[x : \texttt{nat}](x = x)$$

representa un predicado P que en matemáticas suele escribirse $P(x) \equiv x = x$. Si P es de tipo $\texttt{nat} \longrightarrow \texttt{Prop}$, $(P\ x)$ es una proposición y

$$(x : \texttt{nat})(P\ x)$$

representa el tipo de funciones que asocian a cada número natural n un objeto del tipo $(P\ n)$ y consecuentemente representan pruebas de la fórmula $\forall x.P(x)$.

- Como todos los términos del CCI tiene tipo, habrá "tipos de tipos", o clases, o, como se denominan en Coq, *sorts*.

2.2.1.- Sorts

- Los tipos son términos del lenguaje que, por tanto, deberán pertenecer a otros tipos. Los tipos de tipos son siempre una constante del lenguaje llamados sorts.
- Tres sorts:

  1. `Prop`: es el tipo de las proposiciones lógicas. Si M es una proposición lógica, entonces M denota una clase: la clase de los términos que representan pruebas de M. Un objeto m que pertenece a M demuestra que M es cierto.
  2. `Set`: es el tipo de las especificaciones. Incluye programas, y los conjuntos habituales (booleanas, naturales...).
  3. `Type`: tipo abstracto. Por ejemplo, `Set` y `Prop` son de tipo `Type`.

- El conjunto de sorts se denota:

$$\mathcal{S} \equiv \{\texttt{Prop}, \texttt{Set}, \texttt{Type}(i) \mid i \in \mathbb{N}\}$$

2.2.2.- Constantes

- Además de sorts, el lenguaje contiene constantes que denotan objetos del entorno.

2.2.3.- Descripción formal del lenguaje

- Tipos.

  Los tipos pueden dividirse en:

  - Atómicos: los tipos atómicos del CCI son sorts (`Set`, `Prop` y `Type`), variables de tipo o tipos inductivos.
  - Compuestos: un tipo compuesto es un producto $(x : T)U$ con T y U tipos.

- Términos.

  El lenguaje del CCI se construye de acuerdo a las siguientes reglas:

  1. Los sorts `Set`, `Prop` y `Type` son términos.
  2. Las constantes del entorno son términos.
  3. Las variables son términos.
  4. Si x es una variable y T y U son términos, $(x : T)U$ es un término. Tendremos dos casos:
     - Si x aparece en U, hablaremos de *producto dependiente* y $(x : T)U$ se leerá *para todo x de tipo T,U.*
     - Si x no aparece en U, hablaremos de *producto independiente* y $(x : T)U$ se leerá *si T entonces U* y se escribirá $T \longrightarrow U$.
  5. Si x es una variable y T y U son términos, $[x : T]U$ es un término. Esta es la notación para una abstracción en λ-cálculo: el término $[x : T]U$ es una función que mapea elementos de T en U.
  6. Si T y U son términos, (TU) es un término. (TU) se lee como *T aplicado a U.*

- Notación.

  La aplicación es asociativa por la izquierda:

  $$(tt_1t_2) = ((tt_1)t_2)$$

  y el producto y las flechas por la derecha:

  $$(x : A)B \longrightarrow C \longrightarrow D = (x : A)(B \longrightarrow (C \longrightarrow D))$$

  $(x, y : A)B$ o $[x, y : A]B$ denota el producto o la abstracción de variables del mismo tipo, es decir, $(x : A)(y : A)B$ o $[x : A][y : A]B$.

- Sustituciones.

  El término $U\{x/T\}$ significa sustituir las apariciones de la variable libre x por el término T en el término U.

2.3.- TÉRMINOS TIPADOS

- Contextos: cuando hay que darle tipo a una expresión con variables libres, debemos conocer el tipo de las variables para poder darle tipo a la expresión. Un contexto Γ se escribe $[x_1 : T_1; \ldots; x_n : t_n]$, es decir una lista de pares (variable:tipo). Si Γ contiene un $x : T$, se escribe $(x : T) \in \Gamma$ o $x \in \Gamma$. La notación $\Gamma :: (y : T)$ denota el contexto $[x_1 : T_1; \ldots; x_n : t_n; y : T]$. [] es el contexto vacío.

- Entornos: $E[\Gamma]$ es el entorno definido por el contexto Γ más las constantes que hayamos introducido en el entorno.

- La notación $E[\Gamma] \vdash t : T$ significa que el término t está bien tipado con tipo T en el entorno E y contexto Γ.

- $\mathcal{WF}(E)[\Gamma]$ significa que el entorno E está bien formado y Γ es un contexto válido en dicho entorno.

- Un término t está bien tipado en un entorno E si y sólo si existe un contexto Γ y un término T tal que $E[\Gamma] \vdash t : T$ pueda ser derivado de las siguientes reglas:

- W-E:
$$\mathcal{W}F([])[[]]$$

- W-s:
$$\frac{E[\Gamma] \vdash T : s \quad s \in \mathcal{S} \quad x \notin \Gamma \cup E}{\mathcal{WF}(E)[\Gamma :: (x : T)]}$$
Esto significa que si no tengo declarada x en mi contexto y T es válido, entonces puedo añadir $(x : T)$ a éste contexto.

- Regla de tipado Var:
$$\frac{\mathcal{WF}(E)[\Gamma] \quad (x : T) \in \Gamma}{E[\Gamma] \vdash x : T}$$

- Regla de tipado Const:
$$\frac{\mathcal{W}F(E)[\Gamma] \quad (c : T) \in E}{E[\Gamma] \vdash c : T}$$

- Prod:
$$\frac{E[\Gamma] \vdash T : s_1 \quad E[\Gamma :: (x : T)] \vdash U : s_2 \quad s_1, s_2 \in \{\texttt{Prop}, \texttt{Set}\}}{E[\Gamma] \vdash (x : T)U : s_2}$$
Esto simplemente nos permite definir el producto en el entorno.

- Lam:
$$\frac{E[\Gamma] \vdash (x:T)U : s \quad E[\Gamma :: (x:T)] \vdash t : U}{E[\Gamma] \vdash [x:T]t : (x:T)U}$$
Esto simplemente nos permite definir las λ-expresiones en el entorno.

- App:
$$\frac{E[\Gamma] \vdash t : (x:U)T \quad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{x/u\}}$$
Esto define la aplicación en el entorno.

2.4.- CCI EN COQ

- Inicialización del intérprete COQ:

```
[victor@surprise victor] coqtop
Welcome to Coq V6.3 (July 1999)
No .coqrc or .coqrc.6.3 found. Skipping rcfile loading.

Coq <
```

- Las órdenes a COQ acaban en punto.

```
Coq < Quit.
```

Cierra el intérprete COQ.

- COQ es sensible a mayúsculas/minúsculas.

2.4.1.- Declaraciones

- Asocian un nombre con una especificación.
- Tres tipos de especificaciones: proposiciones lógicas, conjuntos matemáticos y tipos abstractos, que se van a corresponder con los tres tipos de sorts ya vistos:
  1. `Prop`: es el tipo de las proposiciones lógicas.
  2. `Set`: es el tipo de las especificaciones.
  3. `Type`: tipo abstracto.

- El comando `Check` se utiliza para ver el tipo de una expresión válida.

```
Coq < Check O.

O
     : nat
```

- `Set` es uno de los tres tipos de sorts básicos del lenguaje, mientras que `O` y `nat` están predefinidos en el conjunto de librerías que carga el intérprete cuando se inicializa.

  Nótese que el cero en la definición de los naturales es la letra "O".

- También hay algunas funciones predefinidas.

```
Coq < Check gt.
gt
     : nat->nat->Prop
```

  `gt` es una función que espera dos argumentos de tipo `nat` para proporcionar un resultado proposición lógica (verdero o falso).

- Lo anterior puede interpretarse de foma similar a como se hace en el paradigma de programación funcional. Se compone un tipo `nat` $\longrightarrow$ `Prop` con uno `nat` para obtener `nat` $\longrightarrow$ `nat` $\longrightarrow$ `Prop`, que en realidad es una abreviatura de `nat` $\longrightarrow$ (`nat` $\longrightarrow$ `Prop`).

```
Coq < Check nat->Prop.
nat->Prop
     : Type

Coq < Check (gt n O).
(gt n O)
     : Prop
```

2.4.2.- Definiciones

- Inicialmente se cargan unas pocas definiciones aritméticas:
  - `nat` de tipo `Set`
  - `O` de tipo `nat`
  - `S` de tipo `nat` $\longrightarrow$ `nat`
  - `plus` de tipo `nat` $\longrightarrow$ `nat` $\longrightarrow$ `nat`

- Se pueden introducir nuevas definiciones, como por ejemplo la constante `uno` como el sucesor de `O`:

```
Coq < Definition one := (S O).
one is defined

Coq < Check one.
one
     : nat
```

- Hay varias sintaxis posibles:

```
Coq < Definition one := (S O).
one is defined

Coq < Definition two :nat := (S O).
two is defined

Coq < Definition three := (S O) : nat.
three is defined
```

- Definición de una función que dobla el valor de una variable:

```
Coq < Definition double := [m:nat] (plus m m).
double is defined

Coq < Check double.
double
     : nat->nat
```

  Esta función espera un argumento de tipo `nat` y construye su resultado como `plus m m`. Los corchetes expresan abstracción en una λ-expresión.

- En el caso anterior, `m` era una variable ligada. También podemos mezclas variables libres y ligadas.

```
Coq < Variable n:nat.
n is assumed

Coq < Definition add_n := [m:nat] (plus m n).
add_n is defined

Coq < Check add_n.
add_n
     : nat->nat
```

- Los paréntesis tienen un significado distinto. Por ejemplo:

```
Coq < Check (m:nat)(gt m 0).
(m:nat)(gt m 0)
     : Prop
```

  viene a significar $\forall m \in \mathbb{N}.m > 0$, ya que `m` aparece en el segundo término. Esto se denomina producto dependiente y, en realidad, equivale al cuantificador universal.

- Si la variable no aparece en el segundo término, tenemos el caso del producto independiente:

```
Coq < Check (m:nat)(Prop).
nat->Prop
     : Type

Coq < Check nat->Prop.
nat->Prop
     : Type
```

  esto es, el tipo de las funciones entre estos dos tipos.

- Resumen: en Coq las construcciones permitidas son

  `x | (M N) | [x:T]M | (x:T)P.`

  donde

  - `x` denota constantes o variables.
  - `(M N)` denota la aplicación.
  - `[x:T]M` representa la λ-expresión de parámetro `x` y cuerpo `M`.
  - `(x:T)P` representa el producto. Si es dependiente, expresa la cuantificación universal. Si es independiente, representa el tipo de las funciones entre ambos tipos.

2.5.- EJEMPLOS DEL CÁLCULO DE PROPOSICIONES

2.5.1.- Ejemplo I

- Definición de variables.

```
Coq < Variable A:Prop.
A is assumed

Coq < Lemma var: A -> A.
1 subgoal

  ============================
   A->A
```

  `Variable` va a definir una variable del contexto de un tipo determinado.

  `Lemma` advierte al sistema que lo que viene a continuación es algo que se quiere demostrar.

  La línea separa el contexto local de los objetivos que se pretenden demostrar.

- La flecha -¿en este caso está sobrecargada: antes era un constructor de tipos de funciones, mientras que en este caso `A ->B` conecta proposiciones y debe ser leído como "A implica B".

- Táctica Intro.

```
var < Intro.
1 subgoal

  H : A
  ============================
   A
```

- `Intro` hace lo siguiente:

$$\frac{}{T \longrightarrow U} \rightsquigarrow \frac{T}{U}$$

  En realidad, esto es aplicar la regla "Lam" para dividir los productos, tanto dependiente como independiente, dejando la primera parte del producto en el contexto local y la segunda parte como objetivo.

- Assumption.

```
var < Assumption.
Subtree proved!
```

  `Assumption` aplica la regla "Var": mira en el contexto local si hay alguna hipótesis del mismo tipo que el objetivo. En caso de que lo haya, el objetivo se prueba. En caso contrario, la regla falla.

- Qed.

```
var < Qed.
Intro.
Assumption.

var is defined
```

  `Qed.` sale del bucle (significa "como queríamos demostrar").

```
Coq < Check var.
var
     : A->A
```

2.5.2.- Ejemplo II

- Antes definíamos `A` como una variable determinada. Ahora lo hacemos para cualquier `A`. Es producto dependiente.

```
Coq < Lemma trivial: (p:Prop) p->p.
1 subgoal

  ============================
   (p:Prop)p->p
```

- `Intros` ejecuta `Intro` todas las veces que pueda.

```
trivial < Intros.
1 subgoal

  p : Prop
  H : p
  ============================
   p
```

- `Undo` da un paso atrás.

```
trivial < Undo.
1 subgoal

  ============================
   (p:Prop)p->p
```

- Con `Intros` podemos dar nombres con que especificar las hipótesis.

```
trivial < Intros q H.
1 subgoal

  q : Prop
  H : q
  ============================
   q
```

- `Exact` examina si alguna de las hipótesis del contexto es convertible en el objetivo. Es un poco más sofisticada que `Assumption`, que solamente mira si una hipótesis es igual al objetivo.

```
trivial < Exact H.
Subtree proved!

trivial < Qed.
Intros q H.
Exact H.

trivial is defined
```

- `Check` simplemente chequea el tipo de lo que se le dice que examine. En cambio, `Print` muestra el conjunto de hipótesis, lo que se ha aplicado y el resultado final.

```
Coq < Print trivial.
trivial = [q:Prop; H:q]H
     : (p:Prop)p->p
```

2.5.3.- Ejemplo III

- Se quiere probar la tautología $((A \longrightarrow (B \longrightarrow C)) \longrightarrow (A \longrightarrow B) \longrightarrow (A \longrightarrow C))$. Recordad que las flechas son asociativas por la derecha.
- se inicializan las variables con `Variable` y la fórmula a demostrar con `Goal`.

```
Coq < Variable A,B,C:Prop.
A is assumed
B is assumed
C is assumed

Coq < Goal (A->(B->C)) -> (A->B) -> (A->C).
1 subgoal

  ============================
   (A->B->C)->(A->B)->A->C
```

Como no hemos dado nombre al objetivo, aparece `Unnamed_thm`.

- Como las flechas son asociativas por la derecha, `Intros` desglosa el teorema a probar de la manera que se muestra.

```
Unnamed_thm < Intros.
1 subgoal

  H : A->B->C
  H0 : A->B
  H1 : A
  ============================
   C
```

- Por composición de las implicaciones se puede ver que al final se obtiene C.

```
Unnamed_thm < Exact ((H H1) (H0 H1)).
Subtree proved!

Unnamed_thm < Qed.
Intros.
Exact (H H1 (H0 H1)).

Unnamed_thm is defined
```

- Chequeamos.

```
Coq < Check Unnamed_thm.
Unnamed_thm
     : (A->B->C)->(A->B)->A->C

Coq < Print Unnamed_thm.
Unnamed_thm =
[H:(A->B->C); H0:(A->B); H1:A](H H1 (H0 H1))
     : (A->B->C)->(A->B)->A->C
```

- La táctica `Auto` implementa un procedimiento de resolución similar al del Prolog para resolver el objetivo. Primero prueba con `Assumption`, luego ejecuta `Intros` y luego va probando una serie de tácticas asociadas con los tipos de las cabezas de los objetivos de forma recursiva a los subobjetivos generados. Las tácticas utilizadas son del conjunto de tácticas llamado `Core`.

```
Coq < Goal (A->(B->C)) -> (A->B) -> (A->C).
1 subgoal

  ============================
   (A->B->C)->(A->B)->A->C

Unnamed_thm < Auto.
Subtree proved!

Unnamed_thm < Save prueba.
Auto.

Overriding name Unnamed_thm and using prueba
prueba is defined
```

- Como se puede ver, se puede redefinir el nombre de la táctica probada. En cualquier momento de la resolución, `Abort` sale de la demostración del teorema.

2.6.- EJEMPLOS DE CÁLCULO DE PREDICADOS

- En Coq, todo lo que se declare en el entorno global queda definido en el mismo. Esto puede dar problemas, que se pueden evitar usando el comando `Section`. Dicho comando permite realizar declaraciones en una sección limitada del entorno global. Para acabar una sección, se usa el comando `End`. Por ejemplo:

```
Coq < Section prueba.

Coq < Variable A:Prop.
A is assumed

Coq < Check A.
A
     : Prop

Coq < End prueba.
```

```
Coq < Check A.
Toplevel input, characters 6-7
> Check A.
>       ^
Error: During the relocation of global references,
 The variable A was not found in the current
 environment
 Perhaps the input is malformed

Coq <
```

- Las secciones permiten anidamientos.

2.6.1.- Ejemplo IV

- Vamos a demostrar que si una relación `R` es simétrica y transitiva sobre su dominio, entonces será reflexiva sobre cualquier punto `x` que tenga un sucesor.

```
Coq < Section CalculoPredicados.

Coq < Variable D:Set.
D is assumed

Coq < Variable R:D->D->Prop.
R is assumed

Coq < Section RSimTrans.

Coq < Hypothesis RSim: (x,y:D) (R x y) -> (R y x).
RSim is assumed

Coq < Hypothesis RTrans: (x,y,z:D) (R x y) -> (R y z) -> (R x z).
RTrans is assumed

Coq <
```

- Ahora introducimos el lema a probar:

```
Coq < Lemma RReflex: (x:D)(EX y | (R x y)) -> (R x x).
1 subgoal

  D : Set
  R : D->D->Prop
  RSim : (x,y:D)(R x y)->(R y x)
  RTrans : (x,y,z:D)(R x y)->(R y z)->(R x z)
  ============================
   (x:D)(EX y:D | (R x y))->(R x x)

RReflex <
```

- Nótese que `(x:D)` significa $\forall x \in D$, mientras que

```
(EX x:D | (P x))
```

  no es más que una sintaxis de `(ex D [x:D] (P x))` (comprobar mediante `Check ex.`) y significa $\exists x \in D, \ldots$

- Como primer paso eliminamos los productos.

```
RReflex < Intros.

1 subgoal
  D : Set
  R : D->D->Prop
  RSim : (x,y:D)(R x y)->(R y x)
  RTrans : (x,y,z:D)(R x y)->(R y z)->(R x z)
  x : D
  H : (EX y:D | (R x y))
  ============================
   (R x x)
```

- Podríamos haber obtenido lo mismo con:

```
RReflex < Undo.
RReflex < Intro y.
1 subgoal

  D : Set
  R : D->D->Prop
  RSim : (x,y:D)(R x y)->(R y x)
  RTrans : (x,y,z:D)(R x y)->(R y z)->(R x z)
  y : D
  ============================
   (EX y0:D | (R y y0))->(R y y)
```

  Nótese el renombramiento de las variables. Volvemos al caso anterior:

```
RReflex < Undo.
RReflex < Intros.

1 subgoal
  D : Set
  R : D->D->Prop
  RSim : (x,y:D)(R x y)->(R y x)
  RTrans : (x,y,z:D)(R x y)->(R y z)->(R x z)
  x : D
  H : (EX y:D | (R x y))
  ============================
   (R x x)
```

- Ahora aplicamos la táctica `Elim` para introducir el cuantificador existencial como universal en el objetivo:

```
RReflex < Elim H.
1 subgoal

  D : Set
  R : D->D->Prop
  RSim : (x,y:D)(R x y)->(R y x)
  RTrans : (x,y,z:D)(R x y)->(R y z)->(R x z)
  x : D
  H : (EX y:D | (R x y))
  ============================
   (x0:D)(R x x0)->(R x x)
```

```
RReflex < Intros y Rxy.
1 subgoal

  D : Set
  R : D->D->Prop
  RSim : (x,y:D)(R x y)->(R y x)
  RTrans : (x,y,z:D)(R x y)->(R y z)->(R x z)
  x : D
  H : (EX y:D | (R x y))
  y : D
  Rxy : (R x y)
  ============================
   (R x x)
```

- Ahora queremos aplicar la hipótesis `RTrans` mediante el comando `Apply`. El comando sabe instanciar `x` con `x` y `z` con `x`, pero no sabe qué hacer con `y`.

```
RReflex < Apply RTrans with y.
2 subgoals

  ...

  ============================
   (R x y)

subgoal 2 is:
 (R y x)
```

- Se finaliza la primera rama de la demostración

```
RReflex < Assumption.
1 subgoal

  D : Set
  R : D->D->Prop
  RSim : (x,y:D)(R x y)->(R y x)
  RTrans : (x,y,z:D)(R x y)->(R y z)->(R x z)
  x : D
  H : (EX y:D | (R x y))
  y : D
  Rxy : (R x y)
  ============================
   (R y x)
```

- Y la segunda.

```
RReflex < Apply RSim.
1 subgoal

  D : Set
  R : D->D->Prop
  RSim : (x,y:D)(R x y)->(R y x)
  RTrans : (x,y,z:D)(R x y)->(R y z)->(R x z)
  x : D
  H : (EX y:D | (R x y))
  y : D
  Rxy : (R x y)
  ============================
   (R x y)

RReflex < Assumption.
Subtree proved!
```

2.6.2.- Ejemplo VI

- Vamos a demostrar que un predicado universal es no-vacío, es decir, la cuantificación existencial puede ser deducida de la cuantificación universal: $\forall x \in D, P \Rightarrow \exists a \in D, P$

```
Coq < Variable D:Set.
D is assumed

Coq < Variable P:D->Prop.
P is assumed

Coq < Variable d:D.
d is assumed

Coq < Lemma cuantif: ((x:D) (P x)) -> (EX a | (P a)).
1 subgoal
  D : Set
  P : D->Prop
  d : D
  ============================
   ((x:D)(P x))->(EX a:D | (P a))
```

```
cuantif < Intros.
1 subgoal
  D : Set
  P : D->Prop
  d : D
  H : (x:D)(P x)
  ============================
   (EX a:D | (P a))

cuantif < Exists d.
1 subgoal
  D : Set
  P : D->Prop
  d : D
  H : (x:D)(P x)
  ============================
   (P d)

cuantif < Trivial.
Subtree proved!
```

2.7.- COMANDOS (Caps 5, 6 del Manual)

- `Print` *ident*. Proporciona información sobre el objeto *ident*.
- `Print All`. Proporciona información sobre el estado del entorno.
- `Search` *ident*. Proporciona el nombre y tipo de todos los teoremas del contexto actual cuyo consecuente tiene la forma `(ident t1 ... tn)`.
- `Load` *ident*. Carga el fichero de texto *ident.v* que no tiene que ir entre comillas. En vez de *ident* se puede especificar la ruta absoluta o relativa del fichero en una cadena (entre comillas).
- `Read Module` *ident*. Carga el módulo almacenado en el fichero de nombre *ident* pero no lo abre: sus contenidos son invisibles para el usuario.
- `Import` *ident*. Abre el módulo previamente cargado con `Read Module`. Sus contenidos resultan visibles para el usuario.
- `Require` *ident*. Carga y abre un mdulo. Si el módulo y ha sido cargado con `Read Module`, sólo lo abre.

  Variante: `Require` *ident ruta*, donde se define entre comillas la ruta al fichero.
- `Declare ML Module` $ruta_1$ ... $ruta_n$. Carga los ficheros compilados con definiciones de *Objective Caml*.

- `Pwd`. Muestra el directorio actual.
- `Cd` *ruta*.
- `AddPath` *ruta*. Añade una nueva ruta a la lista de directorios del sistema de ficheros en los que Coq busca módulos y librerías al invocar comandos como `Read Module` o `Require`.
- `DelPath` *ruta*. Lo contrario a lo anterior.
- `Print LoadPath`. Muestra la lista de directorios del sistema.
- `Locate File` *nombre*. Devuelve la ruta completa del fichero si está en algún directorio de la lista de directorios del sistema.
- `Reset` *ident*. Elimina todos los objetos que han sido introducidos en el entorno actual desde *ident*, él mismo incluído.
- `Save State` *ident*. Guarda el estado actual (los objetos definidos) de modo que se pueda volver a él desde más adelante si es necesario.
- `Print States`. Muestra la lista de estados definidos.
- `Restore State` *ident*. Vuelve al estado *ident*.

  `Restore State Initial` o `Reset Initial`, vuelve al estado inicial después de la inicialización de Coq.
- `Remove State` *ident*. Borra el estado *ident*.
- `Goal` *term*

  `Theorem` *ident* : *term*

  `Lemma` *ident* : *term*

  `Remark` *ident* : *term*

  `Fact` *ident* : *term*

  Permiten entrar en la prueba de un teorema. Con `Remark` el teorema será invisible al abandonar la sección actual. Con `Fact` el teorema será visible sólo en la sección que comprende a la actual. Con el resto de las definiciones, el teorema será conocido fuera de la sección actual.
- `Qed`

  `Save`

  `Save Theorem`

  Guardan la prueba actual actual una vez se ha finalizado con éxito.

- `Proof` *term*. Cuando se está realizando una prueba, es equivalente a `Exact` *Term* `; Save`.
- `Abort`. Cancela la prueba actual volviendo al entorno anterior.
- `Suspend`. Vuelve al entorno anterior a la prueba actual, pero sin cancelarla.
- `Resume`. Vuelve a la última prueba suspendida.
- `Undo`. Cancela los efectos de la última táctica aplicada.
- `Restart`. Reinicia el proceso de prueba.
- `Focus`. Focaliza la prueba en el primer subobjetivo, haciendo invisibles los demás.
- `Unfocus`. Deshace los efectos del comando anterior.
- `Show`. Muestra los objetivos actuales.

TEMA 3.- Tipos de datos inductivos

3.1.- Definición de tipos inductivos

- Veamos dos ejemplos: booleanos, naturales y listas.
- Definición del conjunto de los booleanos:

```
Coq < Inductive bool : Set := true:bool | false:bool.
bool_ind is defined
bool_rec is defined
bool_rect is defined
bool is defined
```

- Con esta definición se realizan varias operaciones.
  - Se define un nuevo `Set` de nombre `bool`.
  - Se definen dos *constructores* de este `Set`, llamados `false` y `true`.
  - Se definen tres reglas de eliminación que permiten razonar sobre los posibles casos de valores booleanos. Una para cada caso: `Prop`, `Set` y `Type`.

  Por ejemplo:

```
Coq < Check bool_ind.
bool_ind
     : (P:(bool->Prop))(P true)->(P false)->(b:bool)(P b)
```

  Viene a significar que para toda propiedad `P` definida sobre los booleanos que se cumple para el valor `true`, esto implica que si se cumple para el valor `false`, entonces se cumple para cualquier booleano. Lo mismo (más o menos) significan `bool_rec` y `bool_rect`.

- Probamos que todo booleano es cierto o falso:

```
Coq < Lemma dualidad: (b:bool)(b=true \/ b=false).
1 subgoal

  ============================
   (b:bool)b=true\/b=false
```

```
dualidad < Intro b.
1 subgoal

  b : bool
  ============================
   b=true\/b=false
```

- Si ahora utilizamos la táctica `Elim`, se va a utilizar `bool_ind` para dividir esto en los dos casos posibles.

```
dualidad < Elim b.
2 subgoals

  b : bool
  ============================
   true=true\/true=false

subgoal 2 is:
 false=true\/false=false

dualidad <
```

- A partir de ahí aplicamos las reglas de eliminación del or. Primero `Left`.

```
dualidad < Left.
2 subgoals

  b : bool
  ============================
   true=true

subgoal 2 is:
 false=true\/false=false

dualidad < Trivial.
1 subgoal

  b : bool
  ============================
   false=true\/false=false
```

- Y después `Right`

```
dualidad < Right.
1 subgoal

  b : bool
  ============================
   false=false

dualidad < Trivial.
Subtree proved!
```

- En cuanto a los números naturales:

```
Coq < Inductive nat:Set := O:nat | S:nat->nat.
nat_ind is defined
nat_rec is defined
nat_rect is defined
nat is defined

Coq < Check nat.
nat
     : Set

Coq < Check O.
O
     : nat

Coq < Check S.
S
     : nat->nat

Coq < Check nat_ind.
nat_ind
     : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

- Podemos hacer la misma lectura que en el caso de los booleanos. Para toda propiedad sobre los naturales que 1) se cumpla en el caso del cero, y 2) si para cualquier natural que la cumpla, la cumplirá su sucesor, tenemos que se cumple para cualquier número natural

- Por tanto, cuando queramos hacer una demostración sobre los naturales podemos utilizar la táctica `Elim` para realizar una demostración por casos.

```
Coq < Variable predicado:nat->Prop.
predicado is assumed

Coq < Lemma tonto: (n:nat)(predicado n).
1 subgoal

  ============================
   (n:nat)(predicado n)

tonto < Intro.
1 subgoal

  n : nat
  ============================
   (predicado n)

tonto < Elim n.
2 subgoals

  n : nat
  ============================
   (predicado O)

subgoal 2 is:
 (n0:nat)(predicado n0)->(predicado (S n0))
```

- En cuanto a las listas:

```
Coq < Variable A:Set.
A is assumed

Coq < Inductive Lista:Set := nula:Lista | constr:A->Lista->Lista.
Lista_ind is defined
Lista_rec is defined
Lista_rect is defined
Lista is defined
```

```
Coq < Check Lista.
Lista
     : Set

Coq < Check Lista_ind.
Lista_ind
     : (P:(Lista->Prop))
        (P nula)
        ->((a:A; l:Lista)(P l)->(P (constr a l)))
        ->(l:Lista)(P l)
```

- También podemos hacer demostraciones en atención a los casos del tipo definido.

```
Coq < Variable propiedad_lista:Lista->Prop.
propiedad_lista is assumed

Coq < Lemma tonto: (L:Lista)(propiedad_lista L).
1 subgoal

  ============================
   (L:Lista)(propiedad_lista L)

tonto < Intro L.
1 subgoal

  L : Lista
  ============================
   (propiedad_lista L)

tonto < Elim L.
2 subgoals

  L : Lista
  ============================
   (propiedad_lista nula)

subgoal 2 is:
 (a:A; l:Lista)(propiedad_lista l)->(propiedad_lista (constr a l))
```

3.2.- Demostraciones con tipos inductivos.

- Ejemplo I.- $n = n + 0$ cuando n es un número natural.
- `Elim` nos devuelve dos objetivos: uno para el cero y otro para el paso inductivo de un número y su sucesor.

```
Coq < Lemma suma_n_O: (n:nat)n=(plus n O).
1 subgoal
  ============================
   (n:nat)n=(plus n O)

suma_n_O < Intro n.
1 subgoal
  n : nat
  ============================
   n=(plus n O)

suma_n_O < Elim n.
2 subgoals
  n : nat
  ============================
   O=(plus O O)

subgoal 2 is:
 (n0:nat)n0=(plus n0 O)->(S n0)=(plus (S n0) O)
```

- La táctica `Simpl` reescribe el objetivo.

```
suma_n_O < Simpl.
2 subgoals
  n : nat
  ============================
   (0)=(0)

subgoal 2 is:
 (n0:nat)n0=(plus n0 (0))->(S n0)=(plus (S n0) (0))

suma_n_O < Trivial.
1 subgoal
  n : nat
  ============================
   (n0:nat)n0=(plus n0 O)->(S n0)=(plus (S n0) O)
```

- Usamos `Intros` y `Simpl`, para poder utilizar una táctica nueva: `Rewrite`

```
suma_n_O < Intros.
1 subgoal
  n : nat
  n0 : nat
  H : n0=(plus n0 (O))
  ============================
   (S n0)=(plus (S n0) (O))

suma_n_O < Simpl.
1 subgoal
  n : nat
  n0 : nat
  H : n0=(plus n0 O)
  ============================
   (S n0)=(S (plus n0 O))
```

- `Rewrite` se aplica al objetivo. El argumento de `Rewrite` debe ser una hipótesis que contenga una igualdad que podamos sustituir en el objetivo por unificación. La flecha significa que se aplica de derecha a izquierda.

```
suma_n_O < Rewrite <- H.
1 subgoal
  n : nat
  n0 : nat
  H : n0=(plus n0 O)
  ============================
   (S n0)=(S n0)

suma_n_O < Trivial.
Subtree proved!
```

- En este caso se que podría haber usado la táctica `Auto`. Dicha táctica hace uso del lema `eq_S`, que merece la pena observar:

```
Coq < Check eq_S.
eq_S
     : (x,y:nat)x=y->(S x)=(S y)
Coq <
```

- De hecho, se puede introducir el lema que acabamos de demostrar en el conjunto de tácticas que va a utilizar coq cuando llamemos a `Auto`:

```
Coq < Hints Resolve suma_n_O.
```

- Ejemplo II.- $\forall n, m : naturales$, tenemos que $S(n + m) = n + S(m)$.

```
Coq < Lemma suma_n_S: (n,m:nat)(S (plus n m))=(plus n (S m)).
1 subgoal
  ============================
   (n,m:nat)(S (plus n m))=(plus n (S m))

suma_n_S < Intros.
1 subgoal
  n : nat
  m : nat
  ============================
   (S (plus n m))=(plus n (S m))

suma_n_S < Elim n.
2 subgoals
  n : nat
  m : nat
  ============================
   (S (plus O m))=(plus O (S m))
subgoal 2 is:
 (n0:nat)
  (S (plus n0 m))=(plus n0 (S m))
  ->(S (plus (S n0) m))=(plus (S n0) (S m))
```

- O también, con `Induction`, que aplica los Intros necesarios antes de aplicar `Elim`.

```
suma_n_S < Restart.
1 subgoal
  ============================
   (n,m:nat)(S (plus n m))=(plus n (S m))

suma_n_S < Induction n.
2 subgoals
  n : nat
  ============================
   (m:nat)(S (plus (0) m))=(plus (0) (S m))

subgoal 2 is:
 (n0:nat)
  ((m:nat)(S (plus n0 m))=(plus n0 (S m)))
  ->(m:nat)(S (plus (S n0) m))=(plus (S n0) (S m))
```

- Usamos `Simpl` para obtener algo que se pueda resolver fácilmente en el primer subobjetivo.

```
suma_n_S < Simpl.
2 subgoals
  n : nat
  ============================
   (m:nat)(S m)=(S m)
subgoal 2 is:
 (n0:nat)
  ((m:nat)(S (plus n0 m))=(plus n0 (S m)))
  ->(m:nat)(S (plus (S n0) m))=(plus (S n0) (S m))

suma_n_S < Trivial.
1 subgoal
  n : nat
  ============================
   (n0:nat)
    ((m:nat)(S (plus n0 m))=(plus n0 (S m)))
    ->(m:nat)(S (plus (S n0) m))=(plus (S n0) (S m))
```

- Aunque con Auto podemos terminar aquí, seguiremos manualmente:

```
suma_n_S < Intros.
1 subgoal
  n : nat
  n0 : nat
  H : (m:nat)(S (plus n0 m))=(plus n0 (S m))
  m : nat
  ============================
   (S (plus (S n0) m))=(plus (S n0) (S m))
```

- Aplicamos `Simpl`

```
suma_n_S < Simpl.
1 subgoal
  n : nat
  n0 : nat
  H : (m:nat)(S (plus n0 m))=(plus n0 (S m))
  m : nat
  ============================
   (S (S (plus n0 m)))=(S (plus n0 (S m)))
```

- Terminamos con `Apply` y `Trivial`.

```
suma_n_S < Apply eq_S.
1 subgoal
  n : nat
  n0 : nat
  H : (m:nat)(S (plus n0 m))=(plus n0 (S m))
  m : nat
  ============================
   (S (plus n0 m))=(plus n0 (S m))

suma_n_S < Trivial.
Subtree proved!

suma_n_S < Save.

Coq < Hints Resolve suma_n_S.
```

- Ejemplo III.- Demostraremos la conmutatividad de la suma.

```
Coq < Lemma suma_com: (n,m:nat)(plus n m)=(plus m n).
1 subgoal
  ============================
   (n,m:nat)(plus n m)=(plus m n)

suma_com < Induction m.
2 subgoals
  n : nat
  m : nat
  ============================
   (plus n O)=(plus O n)
subgoal 2 is:
 (n0:nat)(plus n n0)=(plus n0 n)->(plus n (S n0))=(plus (S n0) n)

suma_com < Simpl.
2 subgoals
  n : nat
  m : nat
  ============================
   (plus n O)=n
subgoal 2 is:
 (n0:nat)(plus n n0)=(plus n0 n)->(plus n (S n0))=(plus (S n0) n)

suma_com < Auto.
1 subgoal
  n : nat
  m : nat
  ============================
   (n0:nat)(plus n n0)=(plus n0 n)->(plus n (S n0))=(plus (S n0) n)
```

- `Auto` funciona porque tenemos `suma_n_O` metido en `Hints`. Manualmente:

```
suma_com < Undo.
2 subgoals
  n : nat
  m : nat
  ============================
   (plus n O)=n
subgoal 2 is:
 (n0:nat)(plus n n0)=(plus n0 n)->(plus n (S n0))=(plus (S n0) n)
```

```
suma_com < Symmetry.
2 subgoals
  n : nat
  m : nat
  ============================
   n=(plus n O)
subgoal 2 is:
 (n0:nat)(plus n n0)=(plus n0 n)->(plus n (S n0))=(plus (S n0) n)

suma_com < Apply suma_n_O.
1 subgoal
  n : nat
  m : nat
  ============================
   (n0:nat)(plus n n0)=(plus n0 n)->(plus n (S n0))=(plus (S n0) n
```

- Vamos a por el segundo subobjetivo.

```
suma_com < Intros.
1 subgoal
  n : nat
  m : nat
  n0 : nat
  H : (plus n n0)=(plus n0 n)
  ============================
   (plus n (S n0))=(plus (S n0) n)

suma_com < Simpl.
1 subgoal
  n : nat
  m : nat
  n0 : nat
  H : (plus n n0)=(plus n0 n)
  ============================
   (plus n (S n0))=(S (plus n0 n))
```

```
suma_com < Rewrite <- H.
1 subgoal
  n : nat
  m : nat
  n0 : nat
  H : (plus n n0)=(plus n0 n)
  ============================
   (plus n (S n0))=(S (plus n n0))

suma_com < Auto.
Subtree proved!
```

3.3.- Funciones y predicados recursivos e inductivos.

- Sintaxis de la definición por casos:
  `[anotacion] Cases` *term0* `of` *term1* => *patrón1* `|` `...` `|` *termN* => *patrónN*
- Sintaxis de las funciones recursivas:
  - `Fixpoint` *ident* `[`*ident1* `:` *tipoN* `]` `:` *tipo0* `:=` *term0*
  - `Fixpoint` *ident1* `[` *ligadura1* `]` `:` *tipo1* `:=` *term1*
    `with ...`
    `with identN [` *ligaduraN* `]` `:` *tipoN* `:=` *termN*
    Ejemplo:

    ```
    Coq < Fixpoint plus [n:nat] : nat->nat
    Coq <       := [m:nat]Cases n of O => m | (S p) => (S (plus p m)) end
    plus is recursively defined
    ```
- Definición de Predicados recursivos.

  ```
  Coq < Definition Is_S
  Coq <        := [n:nat]Cases n of O => False | (S p) => True end.
  Is_S is defined
  ```
- Demostremos que `Is_S (S n)` es cierto para todo n.

  ```
  Coq < Lemma S_Is_S : (n:nat)(Is_S (S n)).
  1 subgoal
    ============================
     (n:nat)(Is_S (S n))
  ```

```
S_Is_S < Simpl.
1 subgoal
  ============================
   nat->True

S_Is_S < Trivial.
Subtree proved!
```

- Podemos utilizar esta definición para transformar un `False` en `(Is_S 0)`, como ocurre en esta demostración de un o de los axiomas de Peano.

```
Coq < Lemma no_confuso : (n:nat)~(0=(S n)).
1 subgoal
  ============================
   (n:nat)~(0)=(S n)
```

- Con la táctica `Red`, se reemplaza la negación por su definición.

```
no_confuso < Red.
1 subgoal
  ============================
   (n:nat)(0)=(S n)->False

no_confuso < Intros.
1 subgoal
  n : nat
  H : (0)=(S n)
  ============================
   False
```

- Usamos la táctica `Change` realizar la tranformación de `False`.

```
no_confuso < Change (Is_S 0).
1 subgoal
  n : nat
  H : (0)=(S n)
  ============================
   (Is_S (0))
```

- Usamos `Rewrite` y `Simpl` para obtener `True`.

```
no_confuso < Rewrite H.
1 subgoal
  n : nat
  H : (0)=(S n)
  ============================
    (Is_S (S n))

no_confuso < Simpl.
1 subgoal
  n : nat
  H : (0)=(S n)
  ============================
    True

no_confuso < Trivial.
Subtree proved!
```

- Existe una táctica llamada `Discriminate` para este tipo de demostraciones+ en las que se pretende ver el absurdo de considerar iguales dos términos del mismo tipo estructuralmente distintos.

```
no_confuso < Restart.
1 subgoal
  ============================
    (n:nat)~(0)=(S n)

no_confuso < Intros.
1 subgoal
  n : nat
  ============================
    ~(0)=(S n)

no_confuso < Discriminate.
Subtree proved!
```

- También podemos definir predicados inductivos.

```
Coq < Inductive le [n:nat] : nat -> Prop
Coq <     := le_n : (le n n)
Coq <      | le_S : (m:nat)(le n m) -> (le n (S m)).
le is defined
le_ind is defined

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

Coq < Check le_ind.
le_ind
     : (n:nat; P:(nat->Prop))
         (P n)
         ->((m:nat)(le n m)->(P m)->(P (S m)))
           ->(n0:nat)(le n n0)->(P n0)
```

- Con ello introducimos un predicado `le: nat->nat->Prop`, con dos constructores `le_n y le_S`, y una regla de eliminación `le_ind`.
- Ejemplo: Demostrar que $\forall n, m : n \leq m \Rightarrow n + 1 \leq m + 1$.

```
Coq < Lemma le_n_S : (n,m:nat)(le n m)->(le (S n) (S m)).
1 subgoal
  ============================
   (n,m:nat)(le n m)->(le (S n) (S m))

le_n_S < Intros.
1 subgoal
  n : nat
  m : nat
  H : (le n m)
  ============================
   (le (S n) (S m))
```

- Con `Elim`, pasamos a una demostración por casos.

```
le_n_S < Elim H.
2 subgoals
  n : nat
  m : nat
  H : (le n m)
  ============================
   (le (S n) (S n))

subgoal 2 is:
 (m0:nat)(le n m0)->(le (S n) (S m0))->(le (S n) (S (S m0)))
```

- Con un `Apply` sobre el constructor del predicado para el caso en que los dos naturales son iguales, acabamos con el primer objetivo.

```
le_n_S < Apply le_n.
1 subgoal
  n : nat
  m : nat
  H : (le n m)
  ============================
   (m0:nat)(le n m0)->(le (S n) (S m0))->(le (S n) (S (S m0)))
```

- Con un `Apply` sobre el constructor del predicado para el caso en que los dos naturales son `n` y `(S m)`, acabamos con el segundo objetivo.

```
le_n_S < Intros.
1 subgoal
  n : nat
  m : nat
  H : (le n m)
  m0 : nat
  H0 : (le n m0)
  H1 : (le (S n) (S m0))
  ============================
   (le (S n) (S (S m0)))
```

```
le_n_S < Apply le_S.
1 subgoal
  n : nat
  m : nat
  H : (le n m)
  m0 : nat
  H0 : (le n m0)
  H1 : (le (S n) (S m0))
  ============================
   (le (S n) (S m0))
le_n_S < Assumption.
Subtree proved!
```

- Otra forma de hacerlo es con `Induction` y `Auto`. Para ello introducimos nuevos axiomas en esta táctica.

```
le_n_S < Restart.
1 subgoal
  ============================
   (n,m:nat)(le n m)->(le (S n) (S m))

le_n_S < Hints Resolve le_n le_S.

le_n_S < Induction 1; Auto.
Subtree proved!
```

- Con `Induction 1` decimos: aplica una inducción (`Intros`+`Elim`) sobre la primer hipótesis sin nombre, es decir, `(le n m)`.

TEMA 4.- MÁS COQ

4.1.- Librerías de Coq (ver capítulo 3 del manual)

- Tres librerías de Coq
  - Inicial: Contiene las conectivas y relaciones lógicas más habituales y tipos de datos. Se carga al arrancar el sistema.
  - Estándar: Contiene axiomas y relaciones sobre conjuntos listas, aritmética de enteros... Accesible a través de Require.
  - Contribuciones de usuarios: No se distribuyen con el sistema. Accesibles a través de FTP.

4.2.- Tácticas (ver capítulo 8 del manual)

- Una táctica se aplica dentro de una prueba como un comando ordinario. Si no se pretende aplicar sobre el primer subobjetivo, puede precederse del número de subobjetivo sobre el que se pretende aplicar, con la sintaxia num : táctica.
- Existen algunos comandos que se pueden aplicar sobre tácticas
  - `Do` *num táctica*. Aplica *táctica num* veces.
  - `táctica1 Orelse` *táctica2*. Intenta aplicar *táctica1* y, en caso de fallar esta, *táctica2*.
  - *táctica1* `:` *táctica2*. Aplica la *táctica1* al objetivo y luego *táctica2* a todos los subobjetivos generados por *táctica1*.
  - *táctica0* `; [` *táctica1* `| ... |` *tácticaN*]. aplica *táctica0* al objetivo y luego *táctica1* al primer subobjetivo nuevo, ... , *tácticaN* al nuevo subobjetivo n-ésimo.
  - `Try` *táctica*. Intenta aplicar *táctica*. En caso de que esta falle, devuelve el fallo producido.
  - `Info` *táctica*. Aplica *táctica* y, en caso de tácticas complejas `Auto`, muestra las operaciones que se realiza.
  - `First [` *táctica1* `| ... |` *tácticaN* `]`. Aplica en orden *táctica1*, ..., *tácticaN* hasta que alguna funciona. Falla si todas as tácticas fallan.
  - `Solve [` *táctica1* `| ... |` *tácticaN* `]`. Intenta resolver el subojetivo actual aplicando por orden *táctica1*, ..., *tácticaN*. Falla si ninguna de ellas consigue resolver el objetivo.
- `Assumption`. Implementa la regla "Var" del CCI. Si existe alguna hipótesis de tipo igual al objetivo a demostrar, resuelve dicho objetivo.

- `Clear` *ident*. Borra la hipótesis *ident*.
- `Intro`. Se aplica a productos. Implementa la regla "Lam" del CCI. Mueve el antecedente del producto a la hipótesis, dejando el consecuente como nuevo objetivo.
- `Intros`. Aplica Intro cuantas veces pueda.
- `Apply` *term*. Se aplica a cualquier objetivo a demostrar. Trata de hacer concordar el objetivo con el consecuente del tipo de alguna de las hipótesis. Si tiene éxito devuelve como nuevos objetivos los antecedentes del tipo de la hipótesis.
- `EApply` *term*. Lo mismo que `Apply`, pero cuando alguna de las variables de *term* no puede ser instanciada, en lugar de fallar, la transforma en una "variable existencial" que deberá ser instanciada más adelante en la prueba.

```
Coq < Variable R:nat->nat->Prop.
R is assumed

Coq < Hypothesis RTrans : (x,y,z:nat) (R x y) -> (R y z) -> (R x z).
RTrans is assumed

Coq < Variables n,m,p:nat.
n is assumed
m is assumed
p is assumed

Coq < Hypothesis Rnm:(R n m).
Rnm is assumed

Coq < Hypothesis Rmp:(R m p).
Rmp is assumed
Coq < Goal (R n p).
1 subgoal
  ============================
   (R n p)
```

```
Unnamed_thm < EApply RTrans.
2 subgoals
  ============================
   (R n ?2)

subgoal 2 is:
 (R ?2 p)

Unnamed_thm < Apply Rnm.
1 subgoal
  ============================
   (R m p)
```

- `Cut` *term*. Aplica la regla "App" del CCI. `Cut U` obtiene del objetivo `T` dos nuevos subobjetivos `U->T` y `U`.
- `Exact`. Si existe alguna hipótesis cuyo tipo es convertible el objetivo actual, demuestra dicho objetivo y proporciona el término de prueba.
- `Generalize`. Se aplica a cualquier objetivo. Si G es el objetivo y t es un subtérmino de tipo T, entonces `Generalize` t reemplaza G por $(x : T)G'$ donde G' es obtenido a partir de G reemplazando todas las ocurrencias de t por x.

```
Coq < Show.
1 subgoal
  x : nat
  y : nat
  ============================
   (le O (plus (plus x y) y))

Coq < Generalize (plus (plus x y) y).
1 subgoal
  x : nat
  y : nat
  ============================
   (n:nat)(le O n)
```

- `Change` *term*. Se aplica a cualquier objetivo. `Change U` reemplaza el objetivo actual `T` por `U` si `U` está bien formado y `T` y `U` son convertibles mediante β-reducción o reglas de reescritura.

- `Contradiction`. Applicable a cualquier objetivo. Trata de encontrar en el contexto actual (después de aplicados todos los Intros) alguna hipótesis equivalente a `False`.
- `Red`. Se aplica a cualquier objetivo de forma `(x:T1) ... (Xk:Tk) (c x1 ... xn)` donde `c` es una constante. Se sustituye `c` por su definición (por ejemplo `t` y se aplica `(t x1 ... xn)` mediante β-reducción o la aplicación de una definición inductiva.
- `Simpl`. Aplicable a cualquier objetivo. Primero aplica β-reducción o una definición inductiva. Después intenta sustituír las constantes por su definición y aplicar de nuevo β-reducción o la definición inductiva.
- `Unfold` *ident*. Aplicable a cualquier objetivo. ident debe ser el nombre de una constante transparente. Sustituye *ident* por su definición en el objetivo y aplica β-reducción y definiciones inductivas hasta lograr una forma normal.
- `Fold` *term*. Aplicable a cualquier objetivo. Se aplica la táctica `Red` a cada ocurrencia de *term* en el objetivo.
- `Constructor` *n*. Aplicable a un objetivo cuya conclusión sea una constante inductiva. Constructor Aplica `Intros` y `Apply` sobre el constructor n-ésimo de dicha conclusión.
- `Elim` *term*. Aplicable a cualquier objetivo. Escoge el destructor apropiado y lo aplica sobre el objetivo.
- `Elim` *term* `with` *term1* `...` *termN*. Lo mismo que lo anterior pero permitiendo dar valores a las premisas dependientes en el esquema de eliminación.
- `Case` term. Equivalente a `Elim` *term* `with` *term1* `...` *termN*.
- `Decompose` *term*. Esta táctica descompone recursivamente una proposición compleja para obtener proposiciones atómicas.

```
Coq < Lemma decomp: (A,B,C:Prop) (A/\B/\C\/B/\C\/C/\A)->C.
1 subgoal
  ============================
    (A,B,C:Prop)A/\B/\C\/B/\C\/C/\A->C
```

```
decomp < Intros A B C H.
1 subgoal
  A : Prop
  B : Prop
  C : Prop
  H : A/\B/\C\/B/\C\/C/\A
  ============================
   C

decomp < Decompose [and or] H.
3 subgoals
  A : Prop
  B : Prop
  C : Prop
  H : A/\B/\C\/B/\C\/C/\A
  H1 : A
  H0 : B
  H3 : C
  ============================
   C

subgoal 2 is:
 C

subgoal 3 is:
 C
```

- `Rewrite` *term*. El tipo de *term* debe ser `(x1:T1) ... (xN:TN)` *term1* `=` *term2*. Se reemplazan todas las ocurrencias de *term1* en el objetivo por *term2*.
- `Rewrite <-` *term*. Lo mismo que lo anterior, pero es *term2* quien es sustituído por *term1*.
- `Replace` *term1* `with` *term2*. Se aplica a cualquier objetivo. Reemplaza las ocurrencias libres de *term1* por *term2* y añade la igualdad *term1*=*term2* como nuevo subobjetivo.
- `Reflexivity`. Aplicable a cualquier objetivo con la forma $t = u$. Chequea si t y u son convertibles y resuelve el objetivo.
- `Symmetry`. Convierte un objetivo con la forma $t = u$ a $u = t$.
- `Compare` *term1 term2*. *term1* y *term2* tienen que ser del mismo tipo inductivo. Si G es el subobjetivo actual, es sustituído por los subobjetivos *term1*=*term2* `->` G y $\sim$*term1*=*term2* `->G`.

- `Discriminate` *ident*. Esta táctica prueba cualquier objetivo constituído por una hipótesis absurda que afirma que dos términos estructuralmente diferentes de un mismo tipo son iguales.
- `Injection` *ident*. Si ident es una hipótesis de tipo *term1* `=` *term2*, `Injection` intenta derivar la igualdad de los subtérminos colocados en la misma posición en ambos. Por ejemplo, de `(S (S n))=(S (S (S m)))` se puede derivar la igualdad `n=(S m)`.
- `Simplify_eq` *ident*. Sea *ident* una hipótesis de la forma *term1 term2*. Si *term1* y *term2* son estructuralmente diferentes, se aplica `Discriminate`, mientras que si no lo son se aplica `Injection`.
- `Auto`. Implementa un procedimiento de resolución al estilo Prolog para demostrar el objetivo actual.
- `Auto` *num*. Fuerza la profundidad de búsqueda a *num*. El máximo por defecto es 5.
- `Auto with` *ident1* `...` *identN*. Hace uso de las bases de datos de teoremas ya demostrados *ident1* ... *identN* además de la habitual.
- `Trivial`. Es una restricción de `Auto` que no es recursiva y usa sólo teoremas con coste 0. Se utiliza para resolver igualdades triviales del tipo `X = X`.
- `EAuto`. Generalización de `Auto` que hace uso de un procedimiento de unificación en lugar de emparejamiento de patrones entre el objetivo y los teoremas de la base de datos. En otras palabras, usa `EApply` en lugar de `Apply`.

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
   {plus [n:nat] : nat->nat :=
      [m:nat]Cases n of
                  (0) => m
                | ((S p)) => (S (plus p m))
                end}
      : nat->nat->nat
```

- 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: (m:nat)(suma 0 m m)
Coq <       | sumaS: (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:(m,n:nat)(suma m n (plus m n)).
1 subgoal
   ============================
    (m,n:nat)(suma m n (plus 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
  m : nat
  ============================
    (n:nat)(suma (0) n (plus (0) n))

subgoal 2 is:
 (n:nat)
  ((n0:nat)(suma n n0 (plus n n0)))
  ->(n0:nat)(suma (S n) n0 (plus (S n) n0))
```

- Fijaos que `(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
  m : nat
  ============================
    (n:nat)(suma (0) n n)

subgoal 2 is:
 (n:nat)
  ((n0:nat)(suma n n0 (plus n n0)))
  ->(n0:nat)(suma (S n) n0 (plus (S n) n0))
```

- 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
  ============================
    (n:nat)
     ((n0:nat)(suma n n0 (plus n n0)))
     ->(n0:nat)(suma (S n) n0 (plus (S n) n0))
```

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

```
testSuma < Intros.
1 subgoal
  m : nat
  n : nat
  H : (n0:nat)(suma n n0 (plus n n0))
  n0 : nat
  ============================
  (suma (S n) n0 (plus (S n) n0))
```

- Podemos simplificar `(plus (S n) n0)` mediante la definición de `plus`. Aplicamos `Simpl`.

```
testSuma2 < Simpl.
1 subgoal
  m : nat
  n : nat
  H : (n0:nat)(suma n n0 (plus n n0))
  n0 : nat
  ============================
    (suma (S n) n0 (S (plus n n0)))
```

- Tenemos como hipótesis que `n+n0=(plus n n0)` y pretendemos demostrar que $S(n)+n0$`=(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
  n : nat
  H : (n0:nat)(suma n n0 (plus n n0))
  n0 : nat
  ============================
    (suma n n0 (plus n n0))
```

- 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 H.
Subtree proved!
```

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

```
Coq < Print nat_rec.
nat_rec =
[P:(nat->Set)](nat_rect P)
       : (P:(nat->Set))(P (0))->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)
```

- `nat_rec` recibe tres argumentos: `P`, `(P 0)` y un argumento para `((n:nat)(P n)->(P (S n)))->(n:nat)(P n)`.
- Por tanto, podríamos definir la función suma como:

```
Definition Suma:=
   (nat_rec
   ([_:nat]nat->nat)
   ([n:nat]n)
   ([_:nat][f:nat->nat][x:nat](S(f x)))).
```

- Utilizando el mismo predicado suma de la demostración anterior podríamos verificar el comportamiento de esta nueva función `Suma`, de forma muy similar.

```
Theorem testSuma:(m,n:nat)(suma m n (Suma m n)).
1 subgoal
  ============================
   (m,n:nat)(suma m n (Suma m n))
testSuma < Intros; Elim m; [(Simpl; Apply suma0) |
                                    (Intros; Simpl; Apply sumaS; Assumption)].
Subtree proved!
```

- Ejemplo II: El producto de naturales.

```
Coq < Print mult.
mult =
Fix mult
  {mult [n:nat] : nat->nat :=
      [m:nat]Cases n of
                 (0) => (0)
               | ((S p)) => (plus m (mult p m))
               end}
      : nat->nat->nat
```

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

```
Inductive producto:nat->nat->nat->Prop :=
   producto0: (n:nat)(producto O n O)
 | productoS: (m,n,p:nat)((producto m n p)->
                          (producto (S m) n (plus m p))
```

- Verificamos la función como la demostración del siguiente teorema.

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

testProducto < Intros; Elim m.
2 subgoals
  m : nat
  n : nat
  ============================
   (producto (0) n (mult (0) n))

subgoal 2 is:
 (n0:nat)
  (producto n0 n (mult n0 n))->(producto (S n0) n (mult (S n0) n))
```

- Acabamos con el primer objetivo...

```
testProducto < Simpl.
2 subgoals
   m : nat
   n : nat
   ============================
    (producto (0) n (0))

subgoal 2 is:
 (n0:nat)
   (producto n0 n (mult n0 n))->(producto (S n0) n (mult (S n0) n))

testProducto < Apply producto0.
1 subgoal
   m : nat
   n : nat
   ============================
    (n0:nat)
     (producto n0 n (mult n0 n))->(producto (S n0) n (mult (S n0) n))
```

- ...y con el segundo. Aplicamos `Simpl` y `Apply`.

```
testProducto < Intros.
1 subgoal
   m : nat
   n : nat
   n0 : nat
   H : (producto n0 n (mult n0 n))
   ============================
    (producto (S n0) n (mult (S n0) n))

testProducto < Simpl.
1 subgoal
  m : nat
  n : nat
  n0 : nat
  H : (producto n0 n (mult n0 n))
  ============================
   (producto (S n0) n (plus n (mult n0 n)))
```

```
testProducto < Apply productoS.
1 subgoal
  m : nat
  n : nat
  n0 : nat
  H : (producto n0 n (mult n0 n))
  ============================
   (producto n0 n (mult n0 n))

testProducto < Assumption.
Subtree proved!
```

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

```
Definition Producto :=
  (nat_rec
    ([_:nat]nat->nat)
    ([n:nat]0)
    ([_:nat][f:nat->nat][y:nat](plus y (f y)))).

Coq < Theorem testProducto : (m,n:nat)(producto m n (Producto m n))
1 subgoal
  ============================
   (m,n:nat)(producto m n (Producto m n))

testProducto < Intros; Elim m; [(Simpl; Apply producto0) |
testProducto <     (Intros; Simpl; Apply productoS; Assumption)].
Subtree proved!
```

- Ejemplo III.- la potencia de números naturales.

```
Fixpoint pot [m,n:nat] : nat :=
   Cases n of 0 => (S 0)
        | (S p) => (mult m (pot m p))
   end.
```

- Definimos la especificación que se tiene que cumplir.

```
Inductive potencia : nat->nat->nat->Prop :=
    potencia0: (m:nat) (potencia m 0 (S 0))
  | potenciaS: (m,n,p:nat) (potencia m n p) -> (potencia m (S n) (mult m p)).
```

- Y demostramos que la función la cumple.

```
Coq < Theorem testPotencia : (m,n:nat)(potencia m n (pot m n)).
1 subgoal
  ============================
   (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:
 (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:
 (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
   ============================
    (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) (mult 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.
Subtree proved!
```

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.

```
Inductive producto:nat->nat->nat->Prop :=
    producto0: (n:nat)(producto 0 n 0)
  | productoS: (m,n,p:nat)((producto m n p)->
                          (producto (S m) n (plus m p))
```

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

```
Coq < Theorem Prod : (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 : Set; P : A->Prop] : Prop :=
       ex_intro : (x:A)(P x)->(ex A P)

Prod < Print sig.
Inductive sig [A : Set; P : A->Prop] : Set :=
       exist : (x:A)(P x)->(sig A P)
```

- 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:
 (n0:nat){p:nat | producto n0 m p}->{p:nat | producto (S n0) m p}
```

- Para el sacar el ∃ 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:
 (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:
 (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
  ============================
   (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}
  ============================
   (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 (plus 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.
Subtree proved!
```

- ¿Por qué es necesario utilizar la versión del $\exists$ 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 =
[n,m:nat]
 (nat_rec [n0:nat]{p:nat | producto n0 m p}
   (exist nat [p:nat](producto (0) m p) (0) (producto0 m))
   [n0:nat; H:({p:nat | producto n0 m p})]
    (sig_rec nat [p:nat](producto n0 m p)
       [_:({p:nat | producto n0 m p})]{p:nat | producto (S n0) m p}
       [x:nat; p:(producto n0 m x)]
         (exist nat [p0:nat](producto (S n0) m p0) (plus m x)
           (productoS n0 m x p)) H) n)
     : (n,m:nat){p:nat | producto n m p}
```

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

```
Coq < Print Prod2.
Prod2 =
[n,m:nat]
 (nat_ind [n0:nat](EX p:nat|(producto n0 m p))
   (ex_intro nat [p:nat](producto (0) m p) (0) (producto0 m))
   [n0:nat; H:(EX p:nat|(producto n0 m p))]
    (ex_ind nat [p:nat](producto n0 m p)
       (EX p:nat|(producto (S n0) m p))
       [x:nat; H0:(producto n0 m x)]
         (ex_intro nat [p:nat](producto (S n0) m p) (plus m x)
           (productoS n0 m x H0)) H) n)
     : (n,m:nat)(EX p:nat|(producto n m p))
```

- 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: (a,b:nat)
Coq <     {q:nat*nat | a=(plus (Snd q) (mult (Fst q) (S b))) /\
Coq <                          (lt (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.

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

- Aplicamos `Intros`.

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

subgoal 2 is:
 (n:nat)
  {q:nat*nat |
     n=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
        /\(lt (snd nat nat q) (S b))}
  ->{q:nat*nat |
        (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
          /\(lt (snd nat nat 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)
      =(plus (snd nat nat ((0),(0)))
          (mult (fst nat nat ((0),(0))) (S b)))
      /\(lt (snd nat nat ((0),(0))) (S b))

subgoal 2 is:
 (n:nat)
  {q:nat*nat |
     n=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
        /\(lt (snd nat nat q) (S b))}
  ->{q:nat*nat |
        (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
          /\(lt (snd nat nat q) (S b))}
```

- Simplificamos.

```
div_euclid < Simpl.
2 subgoals
  a : nat
  b : nat
  ============================
   (0)=(0)/\(lt (0) (S b))

subgoal 2 is:
 (n:nat)
  {q:nat*nat |
    n=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
        /\(lt (snd nat nat q) (S b))}
  ->{q:nat*nat |
        (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
          /\(lt (snd nat nat q) (S b))}
```

- Resolvemos el primer objetivo con `Auto`. Para ello, debe de haberse introducido como `Hint` el `lema lt_O_Sn: (n:nat) (lt O (S n))` de la librería `Arith`, lo que puede hacerse con:

```
div_euclid < Require Arith.
div_euclid < Hints Resolve lt_O_Sn.
div_euclid < Auto.
1 subgoal
  a : nat
  b : nat
  ============================
   (n:nat)
    {q:nat*nat |
        n=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
          /\(lt (snd nat nat q) (S b))}
    ->{q:nat*nat |
          (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
            /\(lt (snd nat nat q) (S b))}
```

- Intros.

```
div_euclid < Intros.
1 subgoal
  a  :  nat
  b  :  nat
  n  :  nat
  H  :  {q:nat*nat |
          n=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
            /\(lt (snd nat nat q) (S b))}
  ============================
    {q:nat*nat |
       (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
         /\(lt (snd nat nat 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=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
            /\(lt (snd nat nat q) (S b))}
  ============================
    (x:(nat*nat))
     n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
        /\(lt (snd nat nat x) (S b))
     ->{q:nat*nat |
          (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
            /\(lt (snd nat nat q) (S b))}
```

- `Elim` nos introduce nuevos productos. Los pasamos al campo de las hipotesis con `Intros`.

```
div_euclid < Intros.
1 subgoal
  a  :   nat
  b  :   nat
  n  :   nat
  H  :   {q:nat*nat |
            n=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
               /\(lt (snd nat nat q) (S b))}
  x : nat*nat
  p : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
           /\(lt (snd nat nat x) (S b))
  ============================
    {q:nat*nat |
       (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
          /\(lt (snd nat nat 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=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
               /\(lt (snd nat nat q) (S b))}
  x : nat*nat
  p : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
           /\(lt (snd nat nat x) (S b))
  ============================
    n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
    ->(lt (snd nat nat x) (S b))
       ->{q:nat*nat |
             (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
               /\(lt (snd nat nat q) (S b))}
```

- Nuevos productos, nuevos `Intros`.

```
div_euclid < Intros.
1 subgoal
  a  :  nat
  b  :  nat
  n  :  nat
  H  :  {q:nat*nat |
          n=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
            /\(lt (snd nat nat q) (S b))}
  x : nat*nat
  p : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
          /\(lt (snd nat nat x) (S b))
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  ============================
    {q:nat*nat |
       (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
         /\(lt (snd nat nat 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
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  ============================
    {q:nat*nat |
       (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
         /\(lt (snd nat nat 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 < Check lt_eq_lt_dec.
lt_eq_lt_dec
       : (n,m:nat){lt n m}+{n=m}+{lt m n}
```

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

```
div_euclid < Elim lt_eq_lt_dec with (S (Snd x)) (S b).
2 subgoals
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  ============================
   {lt (S (snd nat nat x)) (S b)}+{(S (snd nat nat x))=(S b)}
   ->{q:nat*nat |
        (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
          /\(lt (snd nat nat q) (S b))}

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

- Y eliminamos el producto del primer objetivo.

```
div_euclid < Intro.
2 subgoals
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  a0 : {lt (S (snd nat nat x)) (S b)}+{(S (snd nat nat x))=(S b)}
  ============================
   {q:nat*nat |
      (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
        /\(lt (snd nat nat q) (S b))}

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

- ¿Parecía complicado? Pues un poco más.

```
div_euclid < Elim a0.
3 subgoals
   a : nat
   b : nat
   n : nat
   x : nat*nat
   H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
   H1 : (lt (snd nat nat x) (S b))
   a0 : {lt (S (snd nat nat x)) (S b)}+{(S (snd nat nat x))=(S b)}
   ============================
    (lt (S (snd nat nat x)) (S b))
    ->{q:nat*nat |
          (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
            /\(lt (snd nat nat q) (S b))}
subgoal 2 is:
 (S (snd nat nat x))=(S b)
 ... (el mismo consecuente que en el primer subobjetivo)
subgoal 3 is:
 (lt (S b) (S (snd nat nat x)))
 ... (el mismo consecuente que en el primer subobjetivo)
```

- 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
   H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
   H1 : (lt (snd nat nat x) (S b))
   a0 : {lt (S (snd nat nat x)) (S b)}+{(S (snd nat nat x))=(S b)}
   a1 : (lt (S (snd nat nat x)) (S b))
   ============================
    {q:nat*nat |
       (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
         /\(lt (snd nat nat 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
   H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
   H1 : (lt (snd nat nat x) (S b))
   a1 : (lt (S (snd nat nat x)) (S b))
   ============================
    (S n)
       =(plus (snd nat nat (fst nat nat x,(S (snd nat nat x))))
          (mult (fst nat nat (fst nat nat x,(S (snd nat nat x)))) (S b)))
       /\(lt (snd nat nat (fst nat nat x,(S (snd nat nat x)))) (S b))
subgoal 2 is:
 (S (snd nat nat x))=(S b)
 ->{q:nat*nat |
       (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
         /\(lt (snd nat nat q) (S b))}
subgoal 3 is:
 (lt (S b) (S (snd nat nat x)))
 ->{q:nat*nat |
       (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
         /\(lt (snd nat nat q) (S b))}
```

- 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
   H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
   H1 : (lt (snd nat nat x) (S b))
   a1 : (lt (S (snd nat nat x)) (S b))
   ============================
    (S n)=(S (plus (snd nat nat x) (mult (fst nat nat x) (S b))))
       /\(lt (S (snd nat nat 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; Auto.
3 subgoals
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  a1 : (lt (S (snd nat nat x)) (S b))
  ============================   (lt (S (snd nat nat x)) (S b))

subgoal 2 is:
 (S (snd nat nat x))=(S b)
 ->{q:nat*nat |
      (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
        /\(lt (snd nat nat q) (S b))}

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

- Y basta `Assumption` para el segundo objetivo.

```
div_euclid < Assumption.
2 subgoals
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  a0 : {lt (S (snd nat nat x)) (S b)}+{(S (snd nat nat x))=(S b)}
  ============================
```

```
   (S (snd nat nat x))=(S b)
   ->{q:nat*nat |
        (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
          /\(lt (snd nat nat q) (S b))}

subgoal 2 is:
 (lt (S b) (S (snd nat nat x)))
 ->{q:nat*nat |
      (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
        /\(lt (snd nat nat 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 < Intros.
2 subgoals
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  b0 : (S (snd nat nat x))=(S b)
  ============================
   {q:nat*nat |
      (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
        /\(lt (snd nat nat q) (S b))}

subgoal 2 is:
 (lt (S b) (S (snd nat nat x)))
 ->{q:nat*nat |
      (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
        /\(lt (snd nat nat 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
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  b0 : (S (snd nat nat x))=(S b)
  ============================
   (S n)=(S (plus b (mult (fst nat nat x) (S b))))/\(lt (0) (S b))

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

- Con un nuevo Split eliminamos el $\wedge$.

```
div_euclid < Split.
3 subgoals
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  b0 : (S (snd nat nat x))=(S b)
  ============================
   (S n)=(S (plus b (mult (fst nat nat x) (S b))))

subgoal 2 is:
 (lt (0) (S b))

subgoal 3 is:
 (lt (S b) (S (snd nat nat x)))
 ->{q:nat*nat |
     (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
        /\(lt (snd nat nat 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 < Require Omega.

div_euclid < Omega.
2 subgoals
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  b0 : (S (snd nat nat x))=(S b)
  ============================
   (lt (0) (S b))

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

- Demostramos `(lt (0) (S b))` mediante `Trivial`.

```
div_euclid < Trivial.
1 subgoal
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  ============================
   (lt (S b) (S (snd nat nat x)))
   ->{q:nat*nat |
        (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
          /\(lt (snd nat nat q) (S b))}
```

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

```
(S n)=(S (plus b (mult (fst nat nat x) (S b)))) /\ (lt (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
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  b0 : (lt (S b) (S (snd nat nat x)))
  ============================
    {q:nat*nat |
      (S n)=(plus (snd nat nat q) (mult (fst nat nat q) (S b)))
         /\(lt (snd nat nat 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
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  b0 : (lt (S b) (S (snd nat nat x)))
  ============================
    ~(lt b (snd nat nat x))

subgoal 2 is:
 (lt b (snd nat nat x))
```

- Nos libramos del primer objetivo con `Omega`.

```
div_euclid < Omega.
1 subgoal
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  b0 : (lt (S b) (S (snd nat nat x)))
  ============================
   (lt b (snd nat nat x))
```

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

```
div_euclid < Check lt_S_n.
lt_S_n
      : (n,m:nat)(lt (S n) (S m))->(lt n m)

div_euclid < Apply lt_S_n.
1 subgoal
  a : nat
  b : nat
  n : nat
  x : nat*nat
  H0 : n=(plus (snd nat nat x) (mult (fst nat nat x) (S b)))
  H1 : (lt (snd nat nat x) (S b))
  b0 : (lt (S b) (S (snd nat nat x)))
  ============================
   (lt (S b) (S (snd nat nat x)))

div_euclid < Assumption.
Subtree proved!
```

- 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.hk 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: (x,y:nat) {x=y}+{~x=y}.
1 subgoal

   ============================
    (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`.

```
Coq < Print sum.
Inductive sum [A : Set; B : Set] : Set :=
     inl : A->A+B
   | inr : B->A+B
```

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

```
iseq < Induction x.
2 subgoals
   x : nat
   ============================
    (y:nat){(0)=y}+{~(0)=y}

subgoal 2 is:
 (n:nat)((y:nat){n=y}+{~n=y})->(y:nat){(S n)=y}+{~(S n)=y}

iseq < Intro.
2 subgoals
   x : nat
   y : nat
   ============================
    {(0)=y}+{~(0)=y}

subgoal 2 is:
 (n:nat)((y:nat){n=y}+{~n=y})->(y:nat){(S n)=y}+{~(S n)=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
   x : nat
   y : nat
   ============================
    {(0)=(0)}+{~(0)=(0)}

subgoal 2 is:
 (n:nat){(0)=(S n)}+{~(0)=(S n)}

subgoal 3 is:
 (n:nat)((y:nat){n=y}+{~n=y})->(y:nat){(S n)=y}+{~(S n)=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
   x : nat
   y : nat
   ============================
    (0)=(0)

subgoal 2 is:
 (n:nat){(0)=(S n)}+{~(0)=(S n)}

subgoal 3 is:
 (n:nat)((y:nat){n=y}+{~n=y})->(y:nat){(S n)=y}+{~(S n)=y}

iseq < Trivial.
2 subgoals
   x : nat
   y : nat
   ============================
    (n:nat){(0)=(S n)}+{~(0)=(S n)}

subgoal 2 is:
 (n:nat)((y:nat){n=y}+{~n=y})->(y:nat){(S n)=y}+{~(S n)=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)` a través de `Auto`, gracias a los teoremas cargados con las librerías iniciales.

```
iseq < Intro.
2 subgoals
   x : nat
   y : nat
   n : nat
   ============================
    {(0)=(S n)}+{~(0)=(S n)}

subgoal 2 is:
 (n:nat)((y:nat){n=y}+{~n=y})->(y:nat){(S n)=y}+{~(S n)=y}
```

```
iseq < Right.
2 subgoals
   x : nat
   y : nat
   n : nat
   ============================
    ~(0)=(S n)

subgoal 2 is:
 (n:nat)((y:nat){n=y}+{~n=y})->(y:nat){(S n)=y}+{~(S n)=y}

iseq < Trivial.
1 subgoal
   x : nat
   ============================
    (n:nat)((y:nat){n=y}+{~n=y})->(y:nat){(S n)=y}+{~(S n)=y}
```

- Eliminamos los productos con `Intros`.

```
iseq < Intros.
1 subgoal
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  ============================
    {(S n)=y}+{~(S n)=y}
```

- Usamos de nuevo `Case` sobre el objetivo `{(S n)=y}+{~ (S n)=y}` para sustituir `y` por los dos casos posibles en la construcción de un natural.

```
iseq < Case y.
2 subgoals
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  ============================
    {(S n)=(0)}+{~(S n)=(0)}

subgoal 2 is:
 (n0:nat){(S n)=(S n0)}+{~(S n)=(S n0)}
```

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

```
iseq < Right.
2 subgoals
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  ============================
    ~(S n)=(0)

subgoal 2 is:
 (n0:nat){(S n)=(S n0)}+{~(S n)=(S n0)}
```

- Demostramos el primer sub objetivo con `Auto` y elmininamos el producto del segundo con `Intro`.

```
iseq < Auto.
1 subgoal
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  ============================
    (n0:nat){(S n)=(S n0)}+{~(S n)=(S n0)}

iseq < Intro.
1 subgoal
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  n0 : nat
  ============================
    {(S n)=(S n0)}+{~(S n)=(S n0)}
```

- Aplicamos `Case` sobre `{(S n)=(S n0)}+{~ (S n)=(S n0)}`. En este punto, buscamos nuevas hipótesis que reflejen esas mismas igualdades o términos similares. Para ello, el `Case` se realiza sobre la hipótesis `H` sustituyendo `y` por `n0`. De este modo obtenemos dos nuevos objetivos, que son productos cuyo consecuente es el objetivo anterior con `y` sustituido por `n0`, y cuyos antecedentes son los términos del *or* de la hipótesis `H`.

```
iseq < Case (H n0).
2 subgoals
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  n0 : nat
  ============================
    n=n0->{(S n)=(S n0)}+{~(S n)=(S n0)}

subgoal 2 is:
 ~n=n0->{(S n)=(S n0)}+{~(S n)=(S n0)}
```

- Eliminamos con *Intro* el producto del primer objetivo. Obtenemos como nuevo subobjetivo un *or*. Con la hipótesis *e* diciendo que `n=n0`, el término del *or* con el que debemos quedarnos es el izquierdo. Aplicamos `Left`.

```
iseq < Intro.
2 subgoals
   x : nat
   n : nat
   H : (y:nat){n=y}+{~n=y}
   y : nat
   n0 : nat
   e : n=n0
   ============================
    {(S n)=(S n0)}+{~(S n)=(S n0)}

subgoal 2 is:
 ~n=n0->{(S n)=(S n0)}+{~(S n)=(S n0)}
```

```
iseq < Left.
2 subgoals
   x : nat
   n : nat
   H : (y:nat){n=y}+{~n=y}
   y : nat
   n0 : nat
   e : n=n0
   ============================
    (S n)=(S n0)

subgoal 2 is:
 ~n=n0->{(S n)=(S n0)}+{~(S n)=(S n0)}
```

- Demostramos el primer objetivo con `Auto`.

```
iseq < Auto.
1 subgoal
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  n0 : nat
  ============================
    ~n=n0->{(S n)=(S n0)}+{~(S n)=(S n0)}
```

- 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
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  n0 : nat
  ============================
    (n=n0->False)->{(S n)=(S n0)}+{(S n)=(S n0)->False}
```

```
iseq < Intro.
1 subgoal
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  n0 : nat
  n1 : n=n0->False
  ============================
    {(S n)=(S n0)}+{(S n)=(S n0)->False}
```

- Usamos `Right` para quedarnos con el segundo término del *or* y `Apply` para obtener un objetivo `n=n0`, que demostramos con `Auto` gracias a la hipótesis `H0`.

```
iseq < Right.
1 subgoal
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  n0 : nat
  n1 : n=n0->False
  ============================
   (S n)=(S n0)->False

iseq < Intro.
1 subgoal
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  n0 : nat
  n1 : n=n0->False
  H0 : (S n)=(S n0)
  ============================
   False
```

```
iseq < Apply n1.
1 subgoal
  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  n0 : nat
  n1 : n=n0->False
  H0 : (S n)=(S n0)
  ============================
   n=n0

iseq < Auto.
Subtree proved!
```

- Verificaremos la corrección y terminación de un programa para el calculo del factorial de un número entero. Primero debemos establecer una definición en *gallina* y obtener una demostración del siguiente teorema:

```
Require Correctness.
Require Omega.
Require Arith.

Fixpoint fact [n:nat] : nat :=
   Cases n of
      O    => (S O)
   | (S p) => (mult n (fact p))
   end.

Lemma fact_rec : (x,y:nat)(lt O x) ->
   (mult (mult x y) (fact (pred x))) = (mult y (fact x)).
Proof.
Intros x y H.
Generalize (mult_sym x y). Intro H1. Rewrite H1.
Generalize (mult_assoc_r y x (fact (pred x))). Intro H2. Rewrite H2.
Apply (f_equal nat nat [x:nat](mult y x)).
Generalize H. Elim x; Auto with arith.
Save.
```

- Con `Correctness` introducimos el programa imperativo a demostrar.

```
Coq < Global Variable x : nat ref.
A global variable x is assumed

Coq < Global Variable y : nat ref.
A global variable y is assumed

Coq < Correctness factorial
Coq <   begin
Coq <     y := (S O);
Coq <     while (notzerop_bool !x) do
Coq <       { invariant (mult y (fact x)) = (fact x@0) as I
Coq <          variant x for lt }
Coq <       y := (mult !x !y);
Coq <       x := (pred !x)
Coq <     done
    <   end
Coq
Coq <   { y = (fact x@0) }.

3 subgoals
  x : nat
  y : nat
  phi0 : nat
  x0 : nat
  y1 : nat
  Variant1 : phi0=x0
  I : (mult y1 (fact x0))=(fact x)
  resultb : bool
  Test1 : (lt (0) x0)
  ============================
   (lt (pred x0) x0)/\(mult (mult x0 y1) (fact (pred x0)))=(fact x

subgoal 2 is:
 (mult (1) (fact x))=(fact x)

subgoal 3 is:
 y1=(fact x)
```

- Un `Split` para obtener dos nuevos subobjetivos.

```
factorial < Split.
4 subgoals
  x : nat
  y : nat
  phi0 : nat
  x0 : nat
  y1 : nat
  Variant1 : phi0=x0
  I : (mult y1 (fact x0))=(fact x)
  resultb : bool
  Test1 : (lt (0) x0)
  ============================
   (lt (pred x0) x0)

subgoal 2 is:
 (mult (mult x0 y1) (fact (pred x0)))=(fact x)

subgoal 3 is:
 (mult (1) (fact x))=(fact x)

subgoal 4 is:
 y1=(fact x)
```

- No nos compliquemos mucho la vida. Demostramos el decrecimiento de la variante con `Omega`.

```
factorial < Omega.
3 subgoals

...

  ============================
    (mult (mult x0 y1) (fact (pred x0)))=(fact x)

subgoal 2 is:
 (mult (1) (fact x))=(fact x)

subgoal 3 is:
 y1=(fact x)
```

- Nos toca la invariante. Hacemos un `Rewrite` para poder aplicar el teorema `fact_rec`, ya demostrado.

```
factorial < Rewrite <- I.
3 subgoals
  x : nat
  y : nat
  phi0 : nat
  x0 : nat
  y1 : nat
  Variant1 : phi0=x0
  I : (mult y1 (fact x0))=(fact x)
  resultb : bool
  Test1 : (lt (0) x0)
  ============================
   (mult (mult x0 y1) (fact (pred x0)))=(mult y1 (fact x0))

subgoal 2 is:
 (mult (1) (fact x))=(fact x)

subgoal 3 is:
 y1=(fact x)
```

- Aplicamos `fact_rec` mediante `Exact` y nos deshacemos del objetivo.

```
factorial < Exact (fact_rec x0 y1 Test1).
2 subgoals
  x : nat
  y : nat
  ============================
    (mult (1) (fact x))=(fact x)

subgoal 2 is:
 y1=(fact x)
```

- El siguiente subobjetivo no debería ser muy difícil. Se resuelve con `Auto`, que aplica el teorema `mult_1_n`.

```
factorial < Auto with arith.
1 subgoal
  x : nat
  y : nat
  y1 : nat
  x0 : nat
  result : unit
  I : (mult y1 (fact x0))=(fact x)/\x0=(0)
  ============================
    y1=(fact x)
```

- Nos queda demostrar que `y1=(fact x)`. Para ello, utilizamos la hipotesis `I`.

```
factorial < Elim I.
1 subgoal
  x : nat
  y : nat
  y1 : nat
  x0 : nat
  result : unit
  I : (mult y1 (fact x0))=(fact x)/\x0=(0)
  ============================
   (mult y1 (fact x0))=(fact x)->x0=(0)->y1=(fact x)

factorial < Intros H1 H2.
1 subgoal
  x : nat
  y : nat
  y1 : nat
  x0 : nat
  result : unit
  I : (mult y1 (fact x0))=(fact x)/\x0=(0)
  H1 : (mult y1 (fact x0))=(fact x)
  H2 : x0=(0)
  ============================
   y1=(fact x)
```

- Sustituímos `x0` por su valor en `H1`.

```
factorial < Rewrite H2 in H1.
1 subgoal
  x : nat
  y : nat
  y1 : nat
  x0 : nat
  result : unit
  I : (mult y1 (fact x0))=(fact x)/\x0=(0)
  H1 : (mult y1 (fact (0)))=(fact x)
  H2 : x0=(0)
  ============================
    y1=(fact x)
```

- Así podemos sustituír `(fact x)` en el objetivo.

```
factorial < Rewrite <- H1.
1 subgoal
  x : nat
  y : nat
  y1 : nat
  x0 : nat
  result : unit
  I : (mult y1 (fact x0))=(fact x)/\x0=(0)
  H1 : (mult y1 (fact (0)))=(fact x)
  H2 : x0=(0)
  ============================
    y1=(mult y1 (fact (0)))
```

- Intentamos demostrar que $y = y * 0!$. Algo que puede hacer `Auto`.

```
factorial < Auto with arith.
Subtree proved!
```

- Otro ejemplo. Vamos a verificar el programa que calcula X^n , mediante el siguiente método.

$$\begin{array}{rcl} X^{2n} & = & (X^n)^2 \\ X^{2n+1} & = & X * (X^n)^2 \end{array}$$

- Para la especificación usamos la definición tradicional de X^n .

```
Require Even.
Require Div2.
Require Correctness.
Require ArithRing.
Require ZArithRing.

Fixpoint power [x,n:nat] : nat :=
   Cases n of
     O        => (S O)
   | (S n') => (mult x (power x n'))
   end.

Definition square := [n:nat](mult n n).
```

Y debemos demostrar dos lemas para simplificar la demostración de las obligaciones de prueba.

```
(* n = 2*(n/2) => (x^(n/2))^2 = x^n *)

Lemma exp_div2_0 : (x,n:nat)
       n=(double (div2 n))
   -> (square (power x (div2 n)))=(power x n).

(* n = 2*(n/2)+1 => x*(x^(n/2))^2 = x^n *)

Lemma exp_div2_1 : (x,n:nat)
       n=(S (double (div2 n)))
   -> (mult x (square (power x (div2 n))))=(power x n).

Hints Resolve exp_div2_0 exp_div2_1.
```

- También definimos la funcion booleana que nos dice si un natural es par o impar.

```
Definition even_odd_bool := [x:nat](bool_of_sumbool (even_odd_dec x)).
```

- Y, por fin, el programa. En este caso, esta definido de forma recursiva.

```
Coq  < Correctness r_exp
Coq  <     let rec exp (x:nat) (n:nat) : nat { variant n for lt} =
Coq  <        (if (zerop_bool n) then
Coq  <            (S O)
Coq  <          else
Coq  <            let y = (exp x (div2 n)) in
Coq  <            if (even_odd_bool n) then
Coq  <               (mult y y)
Coq  <            else
Coq  <               (mult x (mult y y))
Coq  <        ) { result=(power x n) }
Coq  <.
4 subgoals
  x : nat
  n : nat
  rphi1 : nat
  exp : (phi:nat)
             (lt phi rphi1)
             ->(x0,n0:nat)phi=n0->{result:nat | result=(power x0 n0)}
  x0 : nat
  n0 : nat
  Variant1 : rphi1=n0
  resultb : bool
  Test2 : n0=(0)
  ============================
    (1)=(power x0 n0)

subgoal 2 is:
 (lt (div2 n0) n0)

subgoal 3 is:
 (mult y y)=(power x0 n0)

subgoal 4 is:
 (mult x0 (mult y y))=(power x0 n0)
```

- Reemplazamos `n0` por su valor según la hipotesis `Test2`.

```
r_exp < Rewrite Test2.
4 subgoals
  x : nat
  n : nat
  rphi1 : nat
  exp : (phi:nat)
          (lt phi rphi1)
          ->(x0,n0:nat)phi=n0->{result:nat | result=(power x0 n0)}
  x0 : nat
  n0 : nat
  Variant1 : rphi1=n0
  resultb : bool
  Test2 : n0=(0)
  ============================
    (1)=(power x0 (0))

subgoal 2 is:
 (lt (div2 n0) n0)

subgoal 3 is:
 (mult y y)=(power x0 n0)

subgoal 4 is:
 (mult x0 (mult y y))=(power x0 n0)
```

- Demostramos que $1 = x^0$ con Auto.

```
r_exp < Auto.
3 subgoals

...

  Test2 : (lt (0) n0)
  ============================
    (lt (div2 n0) n0)

subgoal 2 is:
 (mult y y)=(power x0 n0)

subgoal 3 is:
 (mult x0 (mult y y))=(power x0 n0)
```

- Para demostrar el segundo subobjetivo, utilizamos el teorema `lt_div2`.

```
r_exp < Check lt_div2.
lt_div2
     : (n:nat)(lt (0) n)->(lt (div2 n) n)
```

- En concreto, utilizamos `Exact`.

```
r_exp < Exact (lt_div2 n0 Test2).
2 subgoals
  x : nat
  n : nat
  rphi1 : nat
  exp : (phi:nat)
          (lt phi rphi1)
          ->(x0,n0:nat)phi=n0->{result:nat | result=(power x0 n0)}
  x0 : nat
  n0 : nat
  Variant1 : rphi1=n0
  resultb : bool
  Test2 : (lt (0) n0)
  y : nat
  Post7 : y=(power x0 (div2 n0))
  resultb0 : bool
  Test1 : (even n0)
  ============================
   (mult y y)=(power x0 n0)

subgoal 2 is:
 (mult x0 (mult y y))=(power x0 n0)
```

- Para demostrar el tercer sub objetivo, primero cambiamos `(mult y y)` por `(square y)`.

```
r_exp < Change (square y)=(power x0 n0).
2 subgoals

... (* Las mismas hipotesis que antes *)
  ============================
   (square y)=(power x0 n0)

subgoal 2 is:
 (mult x0 (mult y y))=(power x0 n0)
```

- Sustituímos y por su valor segun `Post7`.

```
r_exp < Rewrite Post7.
2 subgoals
  x : nat
  n : nat
  rphi1 : nat
  exp : (phi:nat)
          (lt phi rphi1)
          ->(x0,n0:nat)phi=n0->{result:nat | result=(power x0 n0)}
  x0 : nat
  n0 : nat
  Variant1 : rphi1=n0
  resultb : bool
  Test2 : (lt (0) n0)
  y : nat
  Post7 : y=(power x0 (div2 n0))
  resultb0 : bool
  Test1 : (even n0)
  ============================
   (square (power x0 (div2 n0)))=(power x0 n0)

subgoal 2 is:
 (mult x0 (mult y y))=(power x0 n0)
```

- Como puede verse, el objetivo que queda puede resolverse aplicando `exp_div2_0`. Lo hacemos con `Auto`.

```
r_exp < Auto with arith.
1 subgoal

...(* Las mismas hipotesis que antes *)

  Test1 : (odd n0)
  ============================
   (mult x0 (mult y y))=(power x0 n0)
```

- Para el ultimo subobjetivo reescribimos `(mult y y)` por `(square y)`.

```
r_exp < Change (mult x0 (square y))=(power x0 n0).
1 subgoal
  x : nat
  n : nat
  rphi1 : nat
  exp : (phi:nat)
           (lt phi rphi1)
           ->(x0,n0:nat)phi=n0->{result:nat | result=(power x0 n0)}
  x0 : nat
  n0 : nat
  Variant1 : rphi1=n0
  resultb : bool
  Test2 : (lt (0) n0)
  y : nat
  Post7 : y=(power x0 (div2 n0))
  resultb0 : bool
  Test1 : (odd n0)
  ============================
   (mult x0 (square y))=(power x0 n0)
```

- Hacemos un `Rewrite` con vistas a poder usar `exp_div2_1`.

```
r_exp < Rewrite Post7.
1 subgoal
  x : nat
  n : nat
  rphi1 : nat
  exp : (phi:nat)
           (lt phi rphi1)
           ->(x0,n0:nat)phi=n0->{result:nat | result=(power x0 n0)}
  x0 : nat
  n0 : nat
  Variant1 : rphi1=n0
  resultb : bool
  Test2 : (lt (0) n0)
  y : nat
  Post7 : y=(power x0 (div2 n0))
  resultb0 : bool
  Test1 : (odd n0)
  ============================
   (mult x0 (square (power x0 (div2 n0))))=(power x0 n0)
```

- `p_div2_1` es aplicado mediante `Auto`.

```
r_exp < Auto with arith.
Subtree proved!
```