Matemática Discreta II

Curso: 2001/2002
Cuatrimestre: Primero
Alumna: Laura M. Castro Souto
Profesor: Jose Luis Freire Nistal

1. Introducción. Diferencia entre programas y propiedades

En `Coq` tenemos dos *mundos*: el `Set`, de lo "codificable" y el `Prop`, de propiedades, enunciados, predicciones, etc.

Hay pocas herramientas que combinen ambos aspectos, pero `Coq` lo hace. `Coq` es el compilador de un lenguaje que se llama `gallina`, lenguaje de cálculo de construcciones inductivas, que es la suma del λ-cálculo y la lógica de orden superior.

Existe la *lógica de proposiciones* (la más básica, proposiciones y operadores), el *cálculo de proposiciones* (añade los cuantificadores $\exists$ y $\forall$) y la *lógica de orden superior* (permite que los cuantificadores se puedan aplicar a proposiciones). El λ-cálculo es un núcleo de cómputo asociado a lenguajes funcionales (que no usan el estado de la máquina); en él hay variables, aplicaciones y abstracciones. El cálculo de proposiciones indcutivas abarca el λ-cálculo tipado. `Coq` permite expresar todo en λ-cálculo y escribir propiedades de la lógica de orden superior.

El λ-cálculo es un cálculo fuertemente tipado, todo tiene tipo, por ejemplo: $\forall x : nat \quad x > 0$ es de tipo `nat->Prop`. Si damos un valor $x = 5$, obtendremos $5 > 0$, que es una proposición. Pero las proposiciones son tipos a su vez. Saber si una proposición es cierta o no significa *saber si ese tipo tiene algo "dentro" o no.*

Expresaremos todo del modo *término*:*tipo* (**isomorfismo de Curry-Howard**):

$$\left.\begin{array}{l} A : Prop \\ B : Prop \end{array}\right\} \Rightarrow A \to B : Prop$$

donde A, B, y $A \to B$ son pruebas de `Prop`.

El *isomorfismo* es considerar los tipos como fórmulas y los términos como pruebas. Un término es asimilable a un programa, como un tipo (mundo `Set`).

¿Podemos usar lo mismo en el mundo `Prop`? Sí, pero cambia el punto de vista. Un tipo pasa de ser una función a ser una fórmula: si construimos algo de ese tipo veremos que es verdad. Por ejemplo, el tipo `false` es vacío, no se puede construir algo de tipo `false`. El término es una prueba de la fórmula porque es del tipo que representa esa fórmula.

En Informática no es válida la lógica de reducción al absurdo, es necesario construir una prueba para demostrar algo, es una **lógica constructiva**. Por ejemplo:

$$\forall \alpha, \beta \in \mathbb{I}, \alpha^\beta \in \mathbb{Q}$$

la demostración matemática sería:

$$\text{Sea } \sqrt{2}^{\sqrt{2}} \begin{cases} \text{si } \in \mathbb{Q} \Rightarrow \text{ demostración terminada} \\ \text{si } \in \mathbb{I}, \sqrt{2}^{\sqrt{2}^{\sqrt{2}}} = 2 \Rightarrow \text{ demostración terminada} \end{cases}$$

La demostración matemática es correcta pero no es *constructiva*, no sabemos qué números son. En Informática deberíamos construir un algoritmo que tomase dos números irracionales y nos dijese si su exponenciación es, efectivamente, racional o irracional.

Esos términos son programas y son términos de esta lógica constructivista. Lo importante es que el programa (término) cumpla el tipo, la propiedad. Tendremos que ver si la propiedad es cierta, es decir, que podemos construir un programa (un término) a partir de la prueba de la propiedad o especificación.

En resumen, se trata de *probar* en vez de *programar*.

2. Definición del lenguaje

El lenguaje de `Coq` nos sirve para hablar de objetos y de sus propiedades. Tendremos **términos** y **tipos**, como ya hemos comentado. Los *términos* son expresiones del λ-cálculo tipado, como por ejemplo:

- Variables, x.
- Aplicaciones (un término aplicado a otro), $(t \quad t')$.
- Abstracciones, $[x : A]t$[1]. Una abstracción puede representar un programa.
- Productos, $(x : A)T$ (donde A, T son tipos)[2]. El tipo T puede depender de x o no. Si T *no* depende de x, lo escribimos $A \to T$, a lo que llamamos **producto no dependiente**.

3. Los tipos y las reglas de tipado

Manejaremos dos conceptos:

1. El entorno E (asociado a constantes).
2. El entorno Γ (asociado a variables)[3].

Ambos se refieren a la suposición que hacemos sobre los tipos y lo único que les pediremos es que no sean contradictorios. Para ello, usaremos la expresión $WF(E)[\Gamma]$[4], que significa que E y Γ están bien construidos (no son contradictorios).

Las reglas para ver si E o Γ son correctos son:

1. El entorno vacío no tiene que cumplir nada:

$$\frac{}{WF([])([])}W - E$$

[1] Función que a un término de tipo A le hace corresponder un término t.

[2] Programas que toman algo de tipo A y devuelven algo de tipo T que puede depender de x (de su valor) o no.

[3] Lista de pares *variable*:*tipo*.

[4] WF significa *well-formed*.

2. En otro caso:
$$\frac{E[\Gamma] \vdash T : s, s \in S, x \notin \Gamma \cup E}{WF(E)[\Gamma :: (x : T)]} W - s$$
donde $S \equiv \{Prop, Set, Type(i)/i \in \mathbb{N}\}$ son el conjunto de los *sorts* (tipos simples). `Set` y `Prop` tienen `Type(0)`. En teoría hay infinitos tipos, pero en una máquina, claro está, serán finitos.

Un término t está **bien tipado** en un entorno `E` si allí existe un contexto Γ y un término T tal que el predicado $E[\Gamma] \vdash t : T$ puede derivarse de lo siguiente:

$$\frac{WF(E)[\Gamma]}{E[\Gamma] \vdash Prop : Type(0)} \quad \frac{WF(E)[\Gamma]}{E[\Gamma] \vdash Set : Type(0)} \quad \frac{WF(E)[\Gamma], i < j}{E[\Gamma] \vdash Type(i) : Type(j)} Ax$$

Estas reglas se llaman `Ax` (de *axiomáticas*).

Es decir, basta con coger un entorno y contexto vacíos e ir añadiendo cosas *que no estén ya* para que el resultado esté bien formado.

3.1. ¿Cómo podemos tipar bien? Reglas de tipado

a) Si en el contexto tenemos que x es de tipo T, podemos afirmar que x es de tipo T en ese $E[\Gamma]$:
$$\frac{WF(E)[\Gamma], (x : T) \in \Gamma}{E[\Gamma] \vdash x : T} Var$$
Es decir, para demostrar A, basta que en las hipótesis haya un $a : A$, una prueba (táctica `Assumption`). Por tanto, si en el contexo hay un par $x : T$ podemos afirmar que x es de tipo T.

b) Si en el entorno tenemos un par $c : T$, podemos afirmar que en ese $E[\Gamma]$, c es de tipo T:
$$\frac{WF(E)[\Gamma](c : T) \in E}{E[\Gamma] \vdash c : T} Const$$

c) Introducción de $(x : A)T$, el producto:
$$\frac{E[\Gamma] \vdash T : s_1, E[\Gamma :: (x : T)] \vdash U : s_2, s_1 \in \{Prop, Set\} or s_2 \in \{Prop, Set\}}{E[\Gamma] \vdash (x : T)U : s_2} Prod$$
En `Coq`, el comando `Check` deduce el tipo de lo que se le indique.

d) Esta regla es como la anterior, pero nos dice cuándo podemos utilizar *universos*:
$$\frac{E[\Gamma] \vdash T : Type(i), E[\Gamma :: (x : T)] \vdash U : Type(j), i \leq k, j \leq k}{E[\Gamma] \vdash (x : T)U : Type(k)} Prod$$
lo que quiere decir que en un producto no podemos poner lo que queramos.

e) Regla `Lam`:

$$\frac{E[\Gamma] \vdash (x : T)U : s, E[\Gamma :: (x : T)] \vdash f : U}{E[\Gamma] \vdash [x : T]f : (x : T)U} Lam$$

viene a ser algo como

$$\frac{t : T(x)}{[x : A]t : (x : A)T}$$

es decir, teniendo en $E[\Gamma]$ todo lo de arriba se tiene que el término $[x : T]f$ es una prueba de $(x : T)U$ en el $E[\Gamma]$. La expresión $(x : T)U$ es el *tipo producto*.

Por tanto, podemos tipar problemas más complejos que los $A \to T$, pues se puede demostrar que los programas que hagan $[x : t]f$ son de tipo $(x : T)U$ siempre que podamos demostrar que la expresión f es de tipo U. Con ello podemos tipar programas más complejos incluso de los que no podríamos escribir en `caml`.

¿Hay una regla de demostración asociada a esto? El *objetivo* (`Goal`) es demostrar $\overline{(x : T)U}$, ¿qué táctica podemos usar?

$$E[\Gamma :: (x : T)] \vdash f : U$$

Pasamos a suponer que x es de tipo T:

$$\frac{x : T}{U} \text{ si hay suerte... } \frac{x : T}{f : U}$$

y encontramos f, una prueba de U, y con eso

$$\frac{\ldots}{[x : T]f : (x : T)U}$$

donde $[x : T]f$ es una prueba de $(x : T)U$ (para todo x de T, U).

El tipo $(x : T)U$ es el *tipo de todos los programas*. Recuérdese que U puede depender de x. Si no depende, entonces es el caso:

$$\frac{}{T \to U} \text{ Intro } x \ \frac{x : T}{U}$$

es decir, construimos una función que a cada prueba de T le asocia algo de tipo U. La táctica `Intro` es la más importante.

La expresión $(x : T)U$ puede anidarse: $(x : T)((y : R)U) = (x : T)(y : R)U$, donde si hacemos `Intro` sube la x, si lo volvemos a hacer sube la y. Con `Intros`, hacemos todos los `Intro` posibles de una sola vez. Si tenemos $A \to B \to C$ y hacemos `Intros`, primero sube algo de tipo A, luego algo de tipo B.

En la regla *Lam*, si hacemos `Intro h`, en el tipo U (de $(x : T)U$) aparecerá h donde aparecería x.

f) Regla `App` (aplicación):

$$\frac{E[\Gamma] \vdash f : (x : T)U, E[\Gamma] \vdash t : T}{E[\Gamma] \vdash (ft) : U\{x/t\}} App$$

Es decir, si tenemos un *programa* o *función* de tipo producto y se le aplica a t, nos da algo de tipo U (cambiando x por la t). Por ejemplo:

$$\left.\begin{array}{l} f : (x : A)B \\ a : A \end{array}\right\} \Rightarrow (fa) : B\{x/a\}$$

Esto es parecido al **modus ponens** (si tenemos una prueba de A y una prueba de $(x : A)B$, tendremos una prueba de B). Si intentamos encontrar una prueba de B:

$$\frac{E[\Gamma]}{(fa) : B} \text{ Cut } A\frac{E[\Gamma]}{a : A} \text{ y } \frac{E[\Gamma]}{f : A \rightarrow B}$$

$(fa) : B$ es la prueba de B que queremos encontrar y $a : A$, $f : A \rightarrow B$ son dos nuevos *subobjetivos*; si se demuestran los subobjetivos, por hipótesis, tendremos una prueba de B, como pretendíamos. En `Coq` pueden numerarse los subobjetivos de `Cut` y mostrarlos con `show`+*número de objetivo*.

3.2. Introducción a los tipos inductivos

Los **tipos inductivos** son tipos que podemos definir con constructores (como se suele hacer en `caml`). En `Coq` hay algunos ya predefinidos (listas, operador existencial, comparadores, etc.). Sobre todos estos tipos se pueden aplicar *técnicas de inducción*.

3.2.1. Construcción y comportamiento de *entornos* y *contextos*

Para definir una constante y añadirla al entorno `E` haremos:

```
Definition <nombre> := <termino>
```

Mediante:

```
Variable <nombre> : <tipo>
```

añadimos la variable *nombre* al contexto Γ. Veamos un ejemplo de ejecución en el entorno:

```
Coq < Theorem mi_teorema : A -> A.
```

En lugar de `Theorem` podríamos haber escrito `Lemma` o bien `Fact`, `Definition`,... Lo que sigue a los dos puntos es el tipo que se quiere probar, como la *incógnita*. Tras pulsar la tecla ENTER, entramos en un bucle de prueba, del que podemos escapar tecleando `Abort`.

```
1 subgoal

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

mi_teorema < Intro H.
1 subgoal

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

mi_teorema < Assumption.
Subtree proved!

mi_teorema < Save.
Intro.
Assumption.

mi_teorema is defined
```

`Save` guarda la prueba que acabamos de conseguir; lo que ha hecho es construir la prueba, el término `mi_teorema`. Podemos comprobarlo:

```
Coq < Print mi_teorema.
mi_teorema = [H:A]H
: A->A
```

Es decir, es una función identidad. Hemos construido –probado– algo del tipo $A \to A$; `mi_teorema` no es ya un nombre, es un objeto, la prueba del teorema.

Veamos a continuación un ejemplo de una salida de tipo dependiente del tipo de entrada:

$$(n : nat)(\text{if } n \text{ par } \wedge n > 2 \to \exists x, y : nat \text{ primos } / n = x + y \text{ else nada})$$

Compárese con $(x : A)B$. ¿Cómo sería una prueba de este producto? Sería un programa que aplicado a $4, 6, 8, \ldots$ nos diese una prueba de que esos números se descomponen en suma de dos primos. Por tanto el tipo resultante depende del n sobre el que se aplique.

Pero no podemos encontrar una prueba de B (de todo ese teorema), es la **conjetura de Göldbach**.

El **tipo producto** da como resultado un producto cartesiano:

$$(x : A)B = \prod_{x \in A} B(x)$$

De modo que la prueba del producto es el producto cartesiano de las pruebas para cada uno de los x.

Vamos ahora a tratar de definir un tipo que diga si un número es mayor que otro (propiedad $n \leq m$):

```
(n <= m)
(le n m) : Prop
```

En un ejemplo del lado `Set` usaríamos **constructores**, como con:

```
O : nat | S : nat -> nat
nil : (list A) | cons : A -> (list A) -> (list A)
```

En el lado `Prop`, sin embargo, los *constructores* pasan a ser **axiomas**. Para la función `le`:

```
le_n : (n:nat)(le n n)
| le_s : (n,m:nat)(le n m) -> (le n (S m))
```

El primer *axioma* representa que todo número es menor o igual que sí mismo, mientras que el segundo es la definición recursiva (se aplica a otra instancia de `le`). La definición completa es:

```
Inductive le : nat->nat->Prop :=
le_n : (n:nat)(le n n)
| le_s : (n,m:nat)(le n m)->(le n (S m)).
```

Si esta es una buena definición, deberíamos ser capaces de demostrar el siguiente `Goal` a partir de lo anterior:

```
Goal (n:nat)(le O n)
```

Si consiguiésemos demostrarlo[5], tendríamos $p : (n : nat)(le\ \mathrm{O}\ n)$, es decir, un programa que $\forall n : nat$ nos diría si $0 \leq n$ (toma un n y devuelve algo de tipo $(le\ \mathrm{O}\ n)$. El tipo del programa es $(x : T)U$.

Teniendo en cuenta las reglas que vimos, *Lam* y *App*:

$$\frac{E[\Gamma] \vdash x : T, E[\Gamma : (x : T)] \vdash f : U}{E[\Gamma] \vdash [x : T]f : (x : T)U} Lam \qquad \frac{h : (x : T)U, a : T}{(ha) : U\{x/a\}} App$$

[5]Se consigue con la secuencia de acciones: `Intros. Elim n. Constructor. Intros. Constructor. Assumption.`

Podemos ver cómo va a funcionar p:

$$(p\ \mathrm{O}) : (le\ \mathrm{O}\ \mathrm{O})$$
$$(p\ 1) : (le\ \mathrm{O}\ 1)$$
etc.

¿Qué puede tener como valor $(p\ \mathrm{O})$? Pues $(le\_n\ \mathrm{O})$, que es una prueba de $(le\ \mathrm{O}\ \mathrm{O})$. ¿Y $(p\ 1)$? Pues $(le\_s\ \mathrm{O}\ \mathrm{O}(le\_n\ \mathrm{O}))$, ya que $(le\_s\ \mathrm{O}\ \mathrm{O})$ es una prueba de que si $0 \leq 0$ entonces $0 \leq (\mathrm{S}\ \mathrm{O}) \equiv 0 \leq 1$. De modo que tenemos que aplicar $(le\_s\ \mathrm{O}\ \mathrm{O})$ a una prueba de que $0 \leq 0$. Esta prueba es $(le\_n\ \mathrm{O})$. En resumen:

$$p : \prod_{n \in nat} (le\ \mathrm{O}\ n)$$

Ejemplos

1. Introducimos en el entorno `A, B, C : Prop`. Pretendemos demostrar que:

```
Goal (A->(B->C))->(A->B)->A->C.
```

que es un producto no dependiente. Para demostrarlo, intentaremos construir un programa de este tipo. Hacemos `Intros`:

```
1 subgoal

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

Como $(H1 : A)$ y $H0 : A \rightarrow B$, tenemos que $(H0\ \ H1)B$. Y, por lo tanto, $(H\ \ H1(H0\ \ H1)) : A \rightarrow (B \rightarrow C)$. Podemos introducir por lo tanto `Exact (H H1 (H0 H1))`; la táctica `Exact` proporciona la prueba exacta de algo. Al guardar (`Save q`) tenemos que q es:

```
q : (A->B->C)->(A->B)->A->C
```

es decir, es una función $q = [x : A- > B- > C; y : A- > B; z : A](x\ \ z(y\ \ z))$ donde $(x\ \ z(y\ \ z))$ es de tipo C. El objeto q es por tanto la prueba del teorema y un objeto que podemos usar.

Otra forma de haberlo demostrado habría sido siguiendo la secuencia de tácticas `Apply H. Assumption. Apply H0. Assumption.`; la regla `Apply` es una aplicación de la regla *Lam*.

2. Demostraremos ahora que $A \wedge B \rightarrow B \wedge A$. Es un producto, así que hacemos `Intros`:

```
1 subgoal

  H : A/\B
  ============================
   B/\A
```

¿Cómo le decimos al sistema que use una hipótesis? Con `Elim H`. Esto nos da una prueba de A y una de B:

```
1 subgoal

  H : A/\B
  ============================
   A->B->B/\A

goal < Intros.
1 subgoal

  H : A/\B
  H0 : A
  H1 : B
  ============================
   B/\A

goal < Split.
2 subgoals

  H : A/\B
  H0 : A
  H1 : B
  ============================
   B

subgoal 2 is:
 A
```

Que con `Assumption` quedan demostrados.

¿Y si hubiésemos tenido $A \vee B \rightarrow B \vee A$? La situación habría sido:

```
1 subgoal

  H : A\/B
  ============================
   B\/A

goal < Elim H.
2 subgoals

  H : A\/B
  ============================
   A->B\/A

subgoal 2 is:
 B->B\/A

goal < Intros.
2 subgoals

  H : A\/B
  H0 : A
  ============================
   B\/A

subgoal 2 is:
 B->B\/A

goal < Right.
2 subgoals

  H : A\/B
  H0 : A
  ============================
   A

subgoal 2 is:
 B->B\/A

goal < Assumption.
1 subgoal

  H : A\/B
  ============================
```

```
   B->B\/A

goal < Intros.
1 subgoal

  H : A\/B
  H0 : B
  ============================
   B\/A

goal < Left.
1 subgoal

  H : A\/B
  H0 : B
  ============================
   B

goal < Assumption.
Subtree proved!
```

3.2.2. Definición de tipos inductivos

Veremos a continuación algunas definiciones de tipos ya existentes (predefinidos) en Coq:

```
Inductive nat : Set :=
  O : nat
  | S : nat -> nat.

Inductive and [A,B : Prop] : Prop :=
  conj : A -> B -> (and A B).

Inductive le [n : nat]: Prop :=
  le_n : (le n n)
  | le_s : (m : nat)(le n m) -> (le n (S m)).
```

Son, como podemos ver, tipos **recursivos** (de ahí el `Inductive`).

En Coq, siempre que definimos un tipo de dato, el sistema automáticamente nos provee de medios para probar ese tipo de dato y para construir programas con ese tipo de dato. Así, tendremos los mecanismos `nat_int`, `and_ind` y `le_ind` para probar estos tipos que acabamos de mencionar (mecanismos inductivos asociados a cada uno de ellos), y ocurrirá con todo tipo de dato que definamos:

```
nat_ind : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

and_ind : (A,B,P:Prop)(A->B->P)->A/\B->P
```

`nat_ind` no es sino la *definición de la inducción* misma sobre los naturales.

Para decirle a Coq que utilice el principio de inducción usaremos la táctica ya conocida `Elim`[6].

Existe una táctica que llamamos `Auto` que consulta un árbol mantenido por el sistema con todas las cosas que están en el entorno, todo lo que hay definido, e intenta aplicarlo al objetivo. A `Auto` se le puede pasar un número que representa el nivel del árbol para que busque hasta ese nivel (por defecto es 5). Debemos tener en cuenta que `Auto` no va a hacer cosas como aplicar la inducción.

Cuando un desarrollo se prevee muy largo, puede *abrirse una sección*:

```
Coq < Section mi_seccion.

Coq < Variables A,B,C : Prop.
A is assumed
B is assumed
C is assumed

Coq < Goal A -> (B /\ C) -> ((A -> B) /\ (A -> C)).

...

Coq < Save p.

Coq < End mi_seccion.
```

Al cerrar la sección, el tipo del objeto p será $p : (A, B, C : Prop)A \to (B \wedge C) \to ((A \to B) \wedge (A \to C))$. Es decir, las variables locales se *abstraen*.

[6] Véanse demostraciones como las de `A /\ B ->B /\ A` y `(n:nat)(le O n)`, páginas 10 y 8 respectivamente.

Escribir `<tactica1>; <tactica2>` aplica a todas las subtareas que deje la primera táctica la táctica 2. También se puede hacer `T; [T1 | T2 | ... | Tn]`, y a la rama i se le aplicará la táctica T_i (claro que esto exige saber cuántas ramas vamos a obtener de la aplicación de la táctica 1[7]).

Otro tipo inductivo:

```
Inductive or [A,B : Prop] : Prop :=
  left_intro : A -> (or A B)
  | right_intro : B -> (or A B).
```

Por supuesto, existe el principio de inducción asociado:

```
or_ind : (A,B,P : Prop)(A -> P) -> (B -> P) -> A\/B -> P
```

Así, si se pide demostrar algo en lo que esté implicado el `or`, con `Elim` podremos invocar la aplación del mismo[8]. Para demostrar algo de tipo $A \vee B$ basta con tener una prueba de A o una prueba de B.

Como hemos dicho, todos los tipos que definamos tendrán un *principio de inducción* asociado. Algunos tendrán además también un **principio de recursión**.

3.2.3. La táctica `Elim`

Hemos visto que el lenguaje nos permite incorporar nuevos tipos al contexto. Todos los que vamos a manejar normalmente serán tipos inductivos, es decir, tendrán asociados mecanismos inductivos para probar propiedades sobre dichos tipos y para construir programas sobre los mismos. La prueba y construcción de programas se harán *por inducción estructural*.

El esquema general de la definición de un tipo se hace del siguiente modo:

```
Inductive <nombre> : tipo :=
  <constructores del tipo>.
```

Por ejemplo, como ya sabemos:

```
Inductive nat : Set :=
  O : nat
  | S : nat -> nat.
```

[7] Pistas sobre esto nos las pueden dar, por ejemplo, el número de constructores de un tipo,...

[8] Véase el ejemplo de la página 11, donde se usan los constructores del tipo `or`, `right` y `left`.

Esto es un ejemplo de tipo inductivo *recursivo* (para definir un *nat* se necesita otro *nat*). Otro ejemplo:

```
Inductive list [A : Set] : Set :=
  nil : (list A)
  | cons : A -> (list A) -> (list A).
```

Así, `(cons A a (nil A))` es una lista, que en `caml`, por ejemplo, se expresaría `[a]`. ¿Qué tipo tiene *nil*? `nil:(A:Set)(list A)` y del mismo modo `cons:(A:Set) A ->(list A) ->(list A)`. Esto es un tipo inductivo y *paramétrico*.

Más ejemplos:

```
Inductive and [A,B: Prop] : Prop :=
  conj : A -> B -> (and A B).

Inductive False : Prop := .
```

El tipo `and` es inductivo y no recursivo; la única manera de tener una prueba de $A \wedge B$ será tener $H : (conj \quad x \quad y)$ con $x : A$ e $y : B$. En cuanto al tipo `False`[9], es un tipo sin constructores, un *tipo vacío*, es falso (además de no recursivo y no paramétrico).

Como ya hemos comentado en más de una ocasión, el sistema siempre construye un principio de inducción asociado al tipo, que se nombra de la forma `<nombre tipo>_ind`. Son *predicados*, propiedades que se predican sobre los elementos del tipo en cuestión. Ejemplos:

```
nat_ind : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

and_ind : (A,B,P:Prop)(A->B->P) -> A/\B -> P

False_ind : (P:Prop) False -> P
```

La forma del `and_ind` nos recuerda al *curry* y el *uncurry* de `caml`. De `False` se sigue cualquier cosa. Son los mismos principios que en de los números naturales, pero sobre distintos tipos de datos.

Identifiquemos esto con la táctica `Elim`:

> Supongamos que en el entorno tenemos una hipótesis inductiva. Al hacer `Elim`, se aplicará el principio de inducción correspondiente. Si conseguimos demostrar la primera parte, el sistema podrá resolver el resto.

[9] ¡Ojo! no es lo mismo que `false`, que es del mundo `Set`, un objeto de tipo `bool`, también un tipo inductivo: `Inductive bool : Set := true : bool | false : bool`.

Sólo podemos hacer `Elim` de algo que haya en el entorno. Su *pseudónimo* es `Induction`.

La táctica `Case` hace lo mismo que `Elim` en tipos no recursivos. Su *pseudónimo* es `Destroy`.

> Con `Save State <nombre>` podemos guardar el estado actual de una prueba o demostración. Para guardarlo en un fichero, sólo tenemos que hacer `Save State <nombrefichero.coq>`. Si queremos recuperar ese punto, iniciaríamos el entorno de la siguiente manera: `coqtop inputstate nombrefichero`, o bien, en el propio entorno, con `Restore`. El comando `Print States` imprime los estados que el sistema tiene guardados.

3.2.4. Ciclo completo de un problema

Supongamos que queremos construir el programa que haya el producto de dos números y no sabemos cómo hacerlo. Lo que podemos hacer es *especificar* el programa, es decir, indicarle al sistema qué propiedades queremos que tenga. Para ello, por tanto, en nuestro caso, deberíamos definir la propiedad que especifica el producto.

Podemos suponer que se trata de una relación ternaria ($factor1$, $factor2$ y *producto*):

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

donde los constructores (como `productoO`) son axiomas, ya que son algo de un tipo determinado (en este caso, `(m:nat)(producto O m O)`).

Una vez elegidas las propiedades que debe tener, hemos dicho *qué es* el producto, pero no hemos construido ningún programa que calcule el producto de dos números. Para ello:

```
Theorem Prod : (n,m:nat){p:nat | (producto n m p)}
```

Si demostramos este teorema, entonces estamos construyendo el programa que calcula ese producto (a partir de la prueba de este teorema).

Antes de enfrentarnos con esta demostración, añadiremos un apunte sobre el **cuantificador** `Exist`. En `Coq` va a haber dos tipos para definir el *cuantificador* `Exist`: uno en el lado `Prop` y otro en el lado `Set`:

```
Inductive ex [A : Set; P : A->Prop]  : Prop :=
   ex_intro : (x:A)(P x)->(Ex P)

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

Como podemos ver, necesitamos un elemento de A y una prueba de que ese elemento de A satisface P para garantizar el existe (tener una prueba de que $(EX \quad A \quad P)$). Este `sig` es el que en `Coq` escribimos generalmente en forma de llaves.

Volviendo al teorema:

```
Coq < Theorem Prod : (n,m:nat){p:nat | (producto n m p)}
1 subgoal

  ============================
   (n,m:nat){p:nat | (producto n m p)}

Prod < Intros.
1 subgoal

  n : nat
  m : nat
  ============================
   {p:nat | (producto n m p)}

Prod < Elim n.
2 subgoals

  n : nat
  m : nat
  ============================
   {p:nat | (producto O m p)}

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

Prod < Exists O.
2 subgoals

  n : nat
  m : nat
  ============================
   (producto O m O)
```

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

Prod < Constructor 1.
1 subgoal

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

Prod < Clear n.
1 subgoal

  m : nat
  ============================
   (n:nat){p:nat | (producto n m p)}->
          {p:nat | (producto (S n) m p)}

Prod < Intros.
1 subgoal

  m : nat
  n : nat
  H : {p:nat | (producto n m p)}
  ============================
   {p:nat | (producto (S n) m p)}

Prod < Elim H.
1 subgoal

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

Prod < Intros.
1 subgoal

  m : nat
```

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

Prod < Exists (plus x m).
1 subgoal

  m : nat
  n : nat
  H : {p:nat | (producto n m p)}
  x : nat
  p : (producto n m x)
  ============================
   (producto (S n) m (plus x m))

Prod < Constructor 2.
1 subgoal

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

Prod < Assumption.
Subtree proved!
```

El hacer `Intros` después de `Elim` es útil para incorporar hipótesis al entorno. Después de hacer `Save`, tenemos ya toda la prueba hecha.

```
Coq < Print Prod.
Prod =
[n,m:nat]
 (nat_rec [n0:nat]{p:nat | (producto n0 m p)}
   (exist nat [p:nat](producto O m p) O (productoO 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 x m)
            (productoS m n0 x p)) H) n)
       : (n,m:nat){p:nat | (producto n m p)}
```

Para extraer el programa, tecleamos:

```
Coq < Extraction Prod.
Prod ==>
  [n,m:nat]
   (nat_rec
     nat
     O
     [_,H:nat]
       (sig_rec
         nat
         nat
         [x_nat](plus x m) H) n)
  : (_,_:nat) nat
```

Esto es un lenguaje intermedio. El sistema puede traducirlo a otros lenguajes como `caml` o `haskell`:

```
Coq < Write Caml File "miprod" [Prod].
type nat =
O
| S of nat

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

let prod n m =
  let rec f1 = function
    O -> O
    | S n1 -> plus (f1 n1) m
  in f1 n
```

Probar es más difícil que programar, pero el programa construido a partir de la prueba está garantizado que es totalmente correcto.

3.2.5. Definición del tipo *igualdad*

El **tipo igualdad** se define de la siguiente manera:

```
Inductive eq [A : Set; x : A]  : A->Prop :=  refl_equal : x=x
```

donde $x = x$ equivale a `(eq A x x)`. Esto es un predicado sobre A. Es un tipo *polimórfico*, pues admite como parámetro el tipo sobre el que opera. La igualdad la especificamos, como se puede ver, indicando que una cosa es igual a sí misma. Esta filosofía es la que encarna la táctica `Trivial`, que demuestra que $x = x$.

Como todo tipo inductivo, tenemos un principio de inducción asociado:

```
eq_ind =
[A:Set; x:A; P:(A->Prop); f:(P x); y:A; e:(x=y)]
  Cases e of refl_equal => f end
     : (A:Set; x:A; P:(A->Prop))(P x)->(y:A)x=y->(P y)
```

También, aunque no los habíamos visto hasta ahora, tenemos, claro está:

```
ex_ind =
[A:Set; P:(A->Prop); P0:Prop; f:((x:A)(P x)->P0); e:(Ex P)]
 Cases e of (ex_intro x x0) => (f x x0) end
     : (A:Set; P:(A->Prop); P0:Prop)((x:A)(P x)->P0)->(Ex P)->P0

sig_ind =
[A:Set;
 P:(A->Prop);
 P0:((sig A P)->Prop);
 f:((x:A; p:(P x))(P0 (exist A P x p)));
 s:((sig A P))](let (x, x0) = s in (f x x0))
     : (A:Set; P:(A->Prop); P0:((sig A P)->Prop))
        ((x:A; p:(P x))(P0 (exist A P x p)))->(s:((sig A P)))(P0 s)
```

Una fórmula $\exists x \in S|(P \quad x)$ tiene tipo $(ex \quad A \quad P)$, pues:

```
(ex_intro x h):(ex A P)
```

con $x : A$ y $h : (P \quad x)$. Toda prueba de que existe P es siempre un par de cosas, un elemento de A y una prueba de que verifica P. ¿Qué hace `ex_ind`? A partir de la prueba, construye una prueba de `(P 0)`[10].

[10] Obsérvese que `Prop` ya está probado –tiene dentro un montón de cosas–, de modo que lo que hay que probar es lo que está dentro de `Prop`.

3.2.6. Notas sobre otras tácticas

- La táctica `Induction` equivale a hacer `Intros; Elim`.
- La táctica `Simpl` simplifica funciones de acuerdo con la definición que se tenga del programa, intenta "hacer cuentas".
- La táctica `Hints <Resolve>` mete en la base de datos aquello que se le pase como argumento, de suerte que cuando utilicemos la estrategia `Auto`, se tendrá en cuenta automáticamente por el sistema en su prueba exhaustiva.
- La táctica `Replace` sustituye una cosa por otra, aplicando el principio de inducción del igual.
- En ocasiones, al utilizar la táctica `Apply` pasándole una hipótesis o axioma, debemos indicarle también algún objeto más, ya que de otro modo `Coq` no responderá adecuadamente. Ello se debe a que `Coq` no hace *unificación de segundo orden.*
- La táctica `Eval Compute` evalúa el valor de una función concreta según los parámetros que se le pasen.
- Si aplicamos la táctica `Extraction` al objeto resultante de una demostración en el lado `Prop`, no obtendremos nada. Lo que se puede hacer es extracción de los programas (objetos en `Set`).
- La táctica `Case` es lo mismo que la `Elim` si no hay hipótesis de inducción.
- La táctica `Injection` encarna el teorema:

```
(n,m:nat) (S n)=(S m) -> n=m
```

- La táctica `EAuto` hace lo mismo que `Auto` pero pone interrogantes cuando encuentra cosas que no puede demostrar.

Además de existir un *or para proposiciones*:

```
Inductive or [A : Prop; B : Prop]  : Prop :=
   or_introl : A -> A\/B
 | or_intror : B - >A\/B
```

existe también uno que maneja *conjuntos* en lugar de proposiciones:

```
{A} + {B} : Prop
```

(donde A y B son de tipo `Set`) y que nos permite construir programas que decidan sobre la *visibilidad* de las cosas:

```
Lemma iseq : (x,y:nat) {x=y}+{~x=y}.

Lemma or_eq_noq : (n,m:nat) {(lt n m)} + {n=m} + {(lt m n)}.
```

Que exista un objeto que pruebe alguna de las dos proposiciones anteriores no es evidente, y el criterio de igualdad no está claro[11]. La manera de comprobarlo es demostrando esos lemas:

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

  ============================
   (x,y:nat){x=y}+{~x=y}

iseq < Induction x.
2 subgoals

  x : nat
  ============================
   (y:nat){O=y}+{~O=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
  ============================
   {O=y}+{~O=y}

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

iseq < Case y.
3 subgoals

  x : nat
  y : nat
  ============================
   {O=O}+{~O=O}
```

[11]Para los números reales, no podemos saber si dos números son iguales, aunque sí para los naturales.

```
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 < 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}

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
  ============================
   ~O=(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}

iseq < Intros.
1 subgoal

  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  ============================
   {(S n)=y}+{~(S n)=y}

iseq < Case y.
2 subgoals

  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
  ============================
   {(S n)=O}+{~(S n)=O}

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

iseq < Right.
2 subgoals

  x : nat
  n : nat
  H : (y:nat){n=y}+{~n=y}
  y : nat
```

```
  ============================
   ~(S n)=O

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

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 < Intros.
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 < 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)}

iseq < Intros.
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)}

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)}

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 < Intros.
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)}

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!
```

La palabra clave `Fixpoint` se utiliza para definir **funciones recursivas**:

```
Fixpoint app [X:Set; l:(list X)] : (list X) -> (list X) :=
  [m:(list X)] Cases l of
    Nil => m
    | (Cons a l1) => (Cons X a (app X l1 m))
  end.
```

claro que esta misma definición podría haberse hecho utilizando el `list_rec`:

```
Definition otro_app := [X:Set]
  (list_rec
     X
     [_:(list X)](list X) -> (list X)
     [a:(list X)] a
     [y:X][l:(list X)][f:(list X)->(list X)][ll:(list X)]
          (Cons X y (f ll))).
```

Cuando creamos una definición con `Fixpoint`, el sistema sólo la acepta si es capaz de comprobar que su recursividad *termina*, es decir, que la llamada recursiva se hace sobre algo *estructuralmente menor*.

Otro ejemplo:

```
Definition factorial := (nat_rec
           [_:nat] nat
           (S O)
           [n:nat][f:nat](Product (S n) f)
).
```

Esta es una definición de la función *factorial*; otra definición posible habría sido:

```
Inductive Factorial : nat -> nat -> Prop :=
  Factorial0 : (Factorial 0 (S 0))
  | FactorialS : (n,p:nat) (Factorial n p) ->
                           (Factorial (S n) (Product (S n) p)).
```

De hecho, el primer programa podría salir de la demostración de la siguiente proposición:

```
Coq < Lemma test : (n:nat) (Factorial n (factorial n)).

test < Intros.
1 subgoal

  n : nat
  ============================
   (Factorial n (factorial n))

test < Induction n.
2 subgoals

  ============================
   (Factorial 0 (factorial 0))

subgoal 2 is:
 (Factorial (S n) (factorial (S n)))

test < Constructor.
1 subgoal

  n : nat
  Hrecn : (Factorial n (factorial n))
  ============================
   (Factorial (S n) (factorial (S n)))

test < Simpl.
1 subgoal

  n : nat
  Hrecn : (Factorial n (factorial n))
  ============================
   (Factorial (S n) (plus (factorial n) (mult n (factorial n))))
```

Si dejamos que la simplificación nos expanda todo el factorial no podremos unificar con el segundo constructor después. Para pedir a `Coq` que no desarrolle un término se hace con `Opaque mult`:

```
test < Undo.
1 subgoal

  n : nat
  Hrecn : (Factorial n (factorial n))
  ============================
   (Factorial (S n) (factorial (S n)))

test < Opaque mult.
Warning: This command turns the constants which depend on the
definition/proof of mult un-re-checkable until the next
"Transparent mult" command.

test < Simpl.
1 subgoal

  n : nat
  Hrecn : (Factorial n (factorial n))
  ============================
   (Factorial (S n) (mult (S n) (factorial n)))

test < Constructor.
1 subgoal

  n : nat
  Hrecn : (Factorial n (factorial n))
  ============================
   (Factorial n (factorial n))

test < Assumption.
Subtree proved!
```

3.2.7. Familias `OR` y `AND`

Pensemos en una familia, finita o no, $(\vee_i B_i)$, y en una familia de implicaciones $(\wedge_i(B_i \rightarrow C_i))$; ¿cómo podríamos demostrar

$$(\vee_i B_i) \rightarrow (\wedge_i(B_i \rightarrow C_i)) \rightarrow (\vee_j(B_j \wedge C_j)?$$

(Es decir, demostrar que existe alguna intersección cierta).

El problema es que no podemos usar las propiedades de los tipos como `or` o `and` porque son tipos que reciben dos elementos. La alternativa es:

```
Coq < Theorem interseccion : (B,C:nat->Prop)
Coq < (EX i:nat | (B i)) -> ((i:nat) (B i)->(C i)) ->
```

```
Coq < (EX j:nat | ((B j)/\(C j))).
```

Con el `EX` hemos expresado la familia `or`, y con $\forall i$ hemos solucionado la familia de `and`. Veamos ahora el desarrollo de la prueba:

```
1 subgoal

  ============================
   (B,C:(nat->Prop))
    (EX i:nat | (B i))
    ->((i:nat)(B i)->(C i))
    ->(EX j:nat | (B j)/\(C j))

interseccion < Intros B C e disj.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  e : (EX i:nat | (B i))
  disj : (i:nat)(B i)->(C i)
  ============================
   (EX j:nat | (B j)/\(C j))

interseccion < Case e.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  e : (EX i:nat | (B i))
  disj : (i:nat)(B i)->(C i)
  ============================
   (x:nat)(B x)->(EX j:nat | (B j)/\(C j))

interseccion < Intros.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  e : (EX i:nat | (B i))
  disj : (i:nat)(B i)->(C i)
  x : nat
  H : (B x)
  ============================
   (EX j:nat | (B j)/\(C j))
```

```
interseccion < Exists x.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  e : (EX i:nat | (B i))
  disj : (i:nat)(B i)->(C i)
  x : nat
  H : (B x)
  ============================
   (B x)/\(C x)

interseccion < Split.
2 subgoals

  B : nat->Prop
  C : nat->Prop
  e : (EX i:nat | (B i))
  disj : (i:nat)(B i)->(C i)
  x : nat
  H : (B x)
  ============================
   (B x)

subgoal 2 is:
 (C x)

interseccion < Assumption.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  e : (EX i:nat | (B i))
  disj : (i:nat)(B i)->(C i)
  x : nat
  H : (B x)
  ============================
   (C x)

interseccion < Apply (disj x).
1 subgoal

  B : nat->Prop
  C : nat->Prop
  e : (EX i:nat | (B i))
```

```
  disj : (i:nat)(B i)->(C i)
  x : nat
  H : (B x)
  ============================
   (B x)

interseccion < Assumption.
Subtree proved!
```

Veamos otro ejemplo:

$$\forall i, j/(i \neq j) \rightarrow \neg(B_i \wedge B_j)) \rightarrow (\vee_j(B_j \wedge C_j)) \rightarrow \wedge_i(B_i \rightarrow C_i)$$

```
Coq < Theorem implicacion : (B,C:nat->Prop)
Coq < ((i,j:nat)(~(i=j)->~((B i)/\(B j)))) ->
Coq < ((EX j:nat | ((B j)/\(C j))) -> ((i:nat)((B i)->(C i)))).
1 subgoal

  ============================
   (B,C:(nat->Prop))
    ((i,j:nat)~i=j->~((B i)/\(B j)))
    ->(EX j:nat | (B j)/\(C j))
    ->(i:nat)(B i)->(C i)

implicacion < Require Peano_dec.
```

Necesitaremos utilizar que, dados dos naturales, o son iguales o no lo son (para ello necesitamos cargar el módulo `Peano_dec`[12]). Esto no es algo trivial; de hecho, lo sabemos para los naturales, pero no para otros tipos de datos (la igualdad no siempre es `decidible`).

```
implicacion < Intros.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  ============================
```

[12]Este módulo pone a nuestra disposición la función `eq_nat_dec:(n,m:nat){n=m}+{ ~(n=m)}`, que puede ser demostrada por inducción

```
   (C i)

implicacion < Case H0.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  ============================
   (x:nat)(B x)/\(C x)->(C i)

implicacion < Intros.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  ============================
   (C i)

implicacion < Check (eq_nat_dec i x).
(eq_nat_dec i x)
     : {i=x}+{~i=x}

implicacion < Case (eq_nat_dec i x).
2 subgoals

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  ============================
   i=x->(C i)
```

```
subgoal 2 is:
 ~i=x->(C i)

implicacion < Intros.
2 subgoals

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  e : i=x
  ============================
   (C i)

subgoal 2 is:
 ~i=x->(C i)

implicacion < Rewrite e.
2 subgoals

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  e : i=x
  ============================
   (C x)

subgoal 2 is:
 ~i=x->(C i)

implicacion < Case H2.
2 subgoals

  B : nat->Prop
  C : nat->Prop
```

```
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  e : i=x
  ============================
   (B x)->(C x)->(C x)

subgoal 2 is:
 ~i=x->(C i)

implicacion < Intros.
2 subgoals

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  e : i=x
  H3 : (B x)
  H4 : (C x)
  ============================
   (C x)

subgoal 2 is:
 ~i=x->(C i)

implicacion < Assumption.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  ============================
```

```
   ~i=x->(C i)

implicacion < Intros.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  n : ~i=x
  ============================
   (C i)

implicacion < Case (H i x).
2 subgoals

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  n : ~i=x
  ============================
   ~i=x

subgoal 2 is:
 (B i)/\(B x)

implicacion < Assumption.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
```

```
  H2 : (B x)/\(C x)
  n : ~i=x
  ============================
   (B i)/\(B x)

implicacion < Split.
2 subgoals

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  n : ~i=x
  ============================
   (B i)

subgoal 2 is:
 (B x)

implicacion < Assumption.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  n : ~i=x
  ============================
   (B x)

implicacion < Case H2.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
```

```
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  n : ~i=x
  ============================
   (B x)->(C x)->(B x)

implicacion < Intros.
1 subgoal

  B : nat->Prop
  C : nat->Prop
  H : (i,j:nat)~i=j->~((B i)/\(B j))
  H0 : (EX j:nat | (B j)/\(C j))
  i : nat
  H1 : (B i)
  x : nat
  H2 : (B x)/\(C x)
  n : ~i=x
  H3 : (B x)
  H4 : (C x)
  ============================
   (B x)

implicacion < Assumption.
Subtree proved!
```

3.2.8. Un ejemplo con árboles

Imaginemos un árbol binario de números naturales, y el algoritmo asociado que obtuviese el producto de todas sus hojas. Dicho algoritmo, es fácil de implementar recursivamente. Claro que hay que tener en cuenta que si alguna de las hojas vale cero, podría haber alguna forma de intentar no hacer operaciones inútilmente.

Veremos a continuación que diferentes formas de demostrar un teorema nos llevan a distintos tipos de programas, de soluciones, algunos más óptimos que otros. De hecho, existe el llamado **CPS** (*estilo de paso de continuaciones*), un modo de programar que coincide con una manera de demostrar concreta.

```
Coq < Require Arith.
```

(esto nos permite escribir cosas como `(1)`, `(2)`,... en lugar de `(S O)`, `(S (S O))`,...).

```
Coq < Section Domain.
```

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

Coq < Inductive tree:Set :=
Coq < leaf : Dom -> tree
Coq < | cons : tree -> tree -> tree.
tree is defined
tree_ind is defined
tree_rec is defined
tree_rect is defined

Coq < End Domain.
tree is discharged.
tree_ind is discharged.
tree_rec is discharged.
tree_rect is discharged.
```

(al salir de la sección, `tree` espera un tipo como parámetro).

```
Coq < Check tree.
tree
     : Set->Set

Coq < Check leaf.
leaf
     : (Dom:Set)Dom->(tree Dom)

Coq < Check cons.
cons
     : (Dom:Set)(tree Dom)->(tree Dom)->(tree Dom)
```

Así, `(tree nat)` son los árboles binarios de números naturales. Para simplificar definiremos:

```
Coq < Definition nat_tree := (tree nat).
nat_tree is defined

Coq < Definition nat_cons := (cons nat) :
Coq <    nat_tree -> nat_tree -> nat_tree.
nat_cons is defined

Coq < Definition nat_leaf := (leaf nat) : nat -> (nat_tree).
nat_leaf is defined
```

```
Coq < Check nat_tree.
nat_tree
     : Set

Coq < Check nat_cons.
nat_cons
     : nat_tree->nat_tree->nat_tree

Coq < Check nat_leaf.
nat_leaf
     : nat->nat_tree
```

Una función trivial que realice lo que pretendemos podría ser:

```
Fixpoint leavemult[t:nat_tree]:nat:=
Cases  t of
   (leaf n) => n
 | (cons t1 t2) => (mult (leavemult t1) (leavemult t2))
end.
```

Función nada eficiente, debido al matiz que ya comentamos (caso de existir una hoja de valor cero).

La *especificación* de dicha función debe ser:

```
Definition SPECIF:=[t:nat_tree] {n:nat | (n=(leavemult t))}.
```

Y la siguiente demostración debería dar origen a la propia función `leavemult`:

```
Coq < Definition SPECIF := [t: nat_tree] {n:nat | (n=(leavemult t))}.
SPECIF is defined

Coq < Theorem trivialg: (t:nat_tree)(SPECIF t).
1 subgoal

  ============================
   (t:nat_tree)(SPECIF t)

trivialg < Intro.
1 subgoal

  t : nat_tree
```

```
============================
 (SPECIF t)
```

Hacer `Simpl` ahora no conseguiría nada, porque el sistema no sabe qué es `t` (si fuera `leaf t` o `cons t1 t2`, sí sabría simplificarlo). Es por ello que usamos `Unfold`:

```
trivialg < Unfold SPECIF.
1 subgoal

  t : nat_tree
  ============================
   {n:nat | n=(leavemult t)}

trivialg < Exists (leavemult t).
1 subgoal

  t : nat_tree
  ============================
   (leavemult t)=(leavemult t)

trivialg < Trivial.
Subtree proved!
```

Ahora podríamos extraer de ahí un programa (`Write Caml File "triv"[trivialg]`). En `caml`, se definirán `nat`, `suma`, `mult` y `leavemult`:

```
let rec leavemult = function
  leaf n1 -> n1
  | cons t1 t2 -> (mult (leavemult t1) (leavemult t2));;
```

Estudiemos a continuación cómo optimizar este algoritmo. Necesitamos una función que detecte un cero entre las hojas de un árbol:

```
Fixpoint has_zero[t:nat_tree]:Prop:=
Cases t of
   (leaf n) => (n=(0))
 | (cons t1 t2) => ((has_zero t1) \/ (has_zero t2))
end.
```

Debemos demostrar:

```
Coq < Lemma zero_occ : (t:nat_tree)
Coq <    (has_zero t) -> (leavemult t)=(0).
1 subgoal

  ============================
   (t:nat_tree)(has_zero t)->(leavemult t)=(0)

zero_occ < Induction t.
2 subgoals

  t : nat_tree
  ============================
   (d:nat)(has_zero (leaf nat d))->(leavemult (leaf nat d))=(0)

subgoal 2 is:
 (t0:(tree nat))
  ((has_zero t0)->(leavemult t0)=(0))
  ->(t1:(tree nat))
     ((has_zero t1)->(leavemult t1)=(0))
     ->(has_zero (cons nat t0 t1))
     ->(leavemult (cons nat t0 t1))=(0)

zero_occ < Induction d; Simpl; Auto.
1 subgoal

  t : nat_tree
  ============================
   (t0:(tree nat))
    ((has_zero t0)->(leavemult t0)=(0))
    ->(t1:(tree nat))
       ((has_zero t1)->(leavemult t1)=(0))
       ->(has_zero (cons nat t0 t1))
       ->(leavemult (cons nat t0 t1))=(0)

zero_occ < Intros t1 H1 t2 H2 H.
1 subgoal

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
  H2 : (has_zero t2)->(leavemult t2)=(0)
  H : (has_zero (cons nat t1 t2))
  ============================
   (leavemult (cons nat t1 t2))=(0)
```

```
zero_occ < Simpl.
1 subgoal

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
  H2 : (has_zero t2)->(leavemult t2)=(0)
  H : (has_zero (cons nat t1 t2))
  ============================
   (mult (leavemult t1) (leavemult t2))=(0)

zero_occ < Elim H; Intro H0.
2 subgoals

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
  H2 : (has_zero t2)->(leavemult t2)=(0)
  H : (has_zero (cons nat t1 t2))
  H0 : (has_zero t1)
  ============================
   (mult (leavemult t1) (leavemult t2))=(0)

subgoal 2 is:
 (mult (leavemult t1) (leavemult t2))=(0)

zero_occ < Cut (leavemult t1)=(0).
3 subgoals

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
  H2 : (has_zero t2)->(leavemult t2)=(0)
  H : (has_zero (cons nat t1 t2))
  H0 : (has_zero t1)
  ============================
   (leavemult t1)=(0)->(mult (leavemult t1) (leavemult t2))=(0)

subgoal 2 is:
 (leavemult t1)=(0)
subgoal 3 is:
```

```
 (mult (leavemult t1) (leavemult t2))=(0)

zero_occ < Intro H3.
3 subgoals

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
  H2 : (has_zero t2)->(leavemult t2)=(0)
  H : (has_zero (cons nat t1 t2))
  H0 : (has_zero t1)
  H3 : (leavemult t1)=(0)
  ============================
   (mult (leavemult t1) (leavemult t2))=(0)

subgoal 2 is:
 (leavemult t1)=(0)
subgoal 3 is:
 (mult (leavemult t1) (leavemult t2))=(0)

zero_occ < Rewrite H3; Simpl; Auto.
2 subgoals

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
  H2 : (has_zero t2)->(leavemult t2)=(0)
  H : (has_zero (cons nat t1 t2))
  H0 : (has_zero t1)
  ============================
   (leavemult t1)=(0)

subgoal 2 is:
 (mult (leavemult t1) (leavemult t2))=(0)

zero_occ < Auto.
1 subgoal

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
  H2 : (has_zero t2)->(leavemult t2)=(0)
```

```
  H : (has_zero (cons nat t1 t2))
  H0 : (has_zero t2)
  ============================
   (mult (leavemult t1) (leavemult t2))=(0)

zero_occ < Cut (leavemult t2)=(0).
2 subgoals

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
  H2 : (has_zero t2)->(leavemult t2)=(0)
  H : (has_zero (cons nat t1 t2))
  H0 : (has_zero t2)
  ============================
   (leavemult t2)=(0)->(mult (leavemult t1) (leavemult t2))=(0)

subgoal 2 is:
 (leavemult t2)=(0)

zero_occ < Intro H4.
2 subgoals

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
  H2 : (has_zero t2)->(leavemult t2)=(0)
  H : (has_zero (cons nat t1 t2))
  H0 : (has_zero t2)
  H4 : (leavemult t2)=(0)
  ============================
   (mult (leavemult t1) (leavemult t2))=(0)

subgoal 2 is:
 (leavemult t2)=(0)

zero_occ < Rewrite H4; Simpl; Auto.
1 subgoal

  t : nat_tree
  t1 : (tree nat)
  H1 : (has_zero t1)->(leavemult t1)=(0)
  t2 : (tree nat)
```

```
  H2 : (has_zero t2)->(leavemult t2)=(0)
  H : (has_zero (cons nat t1 t2))
  H0 : (has_zero t2)
  ============================
   (leavemult t2)=(0)

zero_occ < Auto.
Subtree proved!
```

Bien, para obtener el programa optimizado, demostraremos de nuevo:

```
Theorem cpsalg : (t:nat_tree)(SPECIF t).
```

pero de otra manera.

```
1 subgoal

  ============================
   (t:nat_tree)(SPECIF t)

cpsalg < Intro.
1 subgoal

  t : nat_tree
  ============================
   (SPECIF t)

cpsalg < Cut (has_zero t) -> (SPECIF t).
2 subgoals

  t : nat_tree
  ============================
   ((has_zero t)->(SPECIF t))->(SPECIF t)

subgoal 2 is:
 (has_zero t)->(SPECIF t)

cpsalg < Intro ESCAPE_0.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  ============================
   (SPECIF t)
```

```
subgoal 2 is:
 (has_zero t)->(SPECIF t)

cpsalg < 2: Intro.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  ============================
   (SPECIF t)

subgoal 2 is:
 (SPECIF t)

cpsalg < 2: Unfold SPECIF.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  ============================
   (SPECIF t)

subgoal 2 is:
 {n:nat | n=(leavemult t)}

cpsalg < 2 : Exists (0).
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  ============================
   (SPECIF t)

subgoal 2 is:
 (0)=(leavemult t)

cpsalg < 2 : Symmetry.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  ============================
   (SPECIF t)
```

```
subgoal 2 is:
 (leavemult t)=(0)

cpsalg < 2 : Apply zero_occ.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  ============================
   (SPECIF t)

subgoal 2 is:
 (has_zero t)

cpsalg < 2 : Auto.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  ============================
   (SPECIF t)

cpsalg < Local subtree_ersatz := [t',t:nat_tree]
cpsalg <    ((has_zero t')->(has_zero t)).
Warning: subtree_ersatz is declared as a global definition
subtree_ersatz is defined

cpsalg < Hints Unfold subtree_ersatz.

cpsalg < Local kappa := [t:nat_tree][t':nat_tree]
cpsalg <    (n':nat)(n'=(leavemult t'))->(SPECIF t).
Warning: kappa is declared as a global definition
kappa is defined

cpsalg < Hints Unfold kappa.
```

La definición `subtree_ersatz` abarca la relación de *ser subárbol*; `kappa` es algo similar, pero más genérico, representando el hecho de que si encontramos un cero no debemos continuar.

```
cpsalg < Cut (t':nat_tree)
cpsalg <    (subtree_ersatz t' t)->(kappa t t')->(SPECIF t).
2 subgoals

  t : nat_tree
```

```
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  ============================
   ((t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t))
   ->(SPECIF t)

subgoal 2 is:
 (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)

cpsalg < Hints Unfold SPECIF.

cpsalg < Intro AUX.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  AUX : (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)
  ============================
   (SPECIF t)

subgoal 2 is:
 (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)

cpsalg < Apply AUX with t.
3 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  AUX : (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)
  ============================
   (subtree_ersatz t t)

subgoal 2 is:
 (kappa t t)
subgoal 3 is:
 (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)

cpsalg < Auto.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  AUX : (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)
  ============================
   (kappa t t)
```

```
subgoal 2 is:
 (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)

cpsalg < Unfold kappa.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  AUX : (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)
  ============================
   (n':nat)n'=(leavemult t)->(SPECIF t)

subgoal 2 is:
 (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)

cpsalg < Intros n H.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  AUX : (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)
  n : nat
  H : n=(leavemult t)
  ============================
   (SPECIF t)

subgoal 2 is:
 (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)

cpsalg < Unfold SPECIF; Apply exist with n; Auto.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  ============================
   (t':nat_tree)(subtree_ersatz t' t)->(kappa t t')->(SPECIF t)

cpsalg < Induction t'.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  ============================
   (d:nat)
```

```
    (subtree_ersatz (leaf nat d) t)->(kappa t (leaf nat d))->(SPECIF t)

subgoal 2 is:
 (t0:(tree nat))
  ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
   ->(t1:(tree nat))
    ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
     ->(subtree_ersatz (cons nat t0 t1) t)
     ->(kappa t (cons nat t0 t1))
     ->(SPECIF t)

cpsalg < Induction d.
3 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  d : nat
  ============================
   (subtree_ersatz (leaf nat (0)) t)
   ->(kappa t (leaf nat (0)))
   ->(SPECIF t)

subgoal 2 is:
 (n:nat)
  ((subtree_ersatz (leaf nat n) t)->(kappa t (leaf nat n))->(SPECIF t))
  ->(subtree_ersatz (leaf nat (S n)) t)
  ->(kappa t (leaf nat (S n)))
  ->(SPECIF t)
subgoal 3 is:
 (t0:(tree nat))
  ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
  ->(t1:(tree nat))
     ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
     ->(subtree_ersatz (cons nat t0 t1) t)
     ->(kappa t (cons nat t0 t1))
     ->(SPECIF t)

cpsalg < Intros H H0.
3 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  d : nat
```

```
  H : (subtree_ersatz (leaf nat (0)) t)
  H0 : (kappa t (leaf nat (0)))
  ============================
   (SPECIF t)

subgoal 2 is:
 (n:nat)
  ((subtree_ersatz (leaf nat n) t)->(kappa t (leaf nat n))->(SPECIF t))
  ->(subtree_ersatz (leaf nat (S n)) t)
  ->(kappa t (leaf nat (S n)))
  ->(SPECIF t)
subgoal 3 is:
 (t0:(tree nat))
  ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
   ->(t1:(tree nat))
     ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
     ->(subtree_ersatz (cons nat t0 t1) t)
     ->(kappa t (cons nat t0 t1))
     ->(SPECIF t)

cpsalg < Apply ESCAPE_0.
3 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  d : nat
  H : (subtree_ersatz (leaf nat (0)) t)
  H0 : (kappa t (leaf nat (0)))
  ============================
   (has_zero t)

subgoal 2 is:
 (n:nat)
  ((subtree_ersatz (leaf nat n) t)->(kappa t (leaf nat n))->(SPECIF t))
  ->(subtree_ersatz (leaf nat (S n)) t)
  ->(kappa t (leaf nat (S n)))
  ->(SPECIF t)
subgoal 3 is:
 (t0:(tree nat))
  ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
  ->(t1:(tree nat))
     ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
     ->(subtree_ersatz (cons nat t0 t1) t)
     ->(kappa t (cons nat t0 t1))
```

```
     ->(SPECIF t)

cpsalg < Apply H.
3 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  d : nat
  H : (subtree_ersatz (leaf nat (0)) t)
  H0 : (kappa t (leaf nat (0)))
  ============================
   (has_zero (leaf nat (0)))

subgoal 2 is:
 (n:nat)
  ((subtree_ersatz (leaf nat n) t)->(kappa t (leaf nat n))->(SPECIF t))
  ->(subtree_ersatz (leaf nat (S n)) t)
  ->(kappa t (leaf nat (S n)))
  ->(SPECIF t)
subgoal 3 is:
 (t0:(tree nat))
  ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
  ->(t1:(tree nat))
     ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
     ->(subtree_ersatz (cons nat t0 t1) t)
     ->(kappa t (cons nat t0 t1))
     ->(SPECIF t)

cpsalg < Simpl; Auto.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  d : nat
  ============================
   (n:nat)
    ((subtree_ersatz (leaf nat n) t)
     ->(kappa t (leaf nat n))
     ->(SPECIF t))
    ->(subtree_ersatz (leaf nat (S n)) t)
    ->(kappa t (leaf nat (S n)))
   ->(SPECIF t)
```

```
subgoal 2 is:
 (t0:(tree nat))
  ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
  ->(t1:(tree nat))
     ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
     ->(subtree_ersatz (cons nat t0 t1) t)
     ->(kappa t (cons nat t0 t1))
     ->(SPECIF t)

cpsalg < Intros y H1 H2 H3.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  d : nat
  y : nat
  H1 : (subtree_ersatz (leaf nat y) t)
       ->(kappa t (leaf nat y))
       ->(SPECIF t)
  H2 : (subtree_ersatz (leaf nat (S y)) t)
  H3 : (kappa t (leaf nat (S y)))
  ============================
   (SPECIF t)

subgoal 2 is:
 (t0:(tree nat))
  ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
  ->(t1:(tree nat))
     ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
     ->(subtree_ersatz (cons nat t0 t1) t)
     ->(kappa t (cons nat t0 t1))
     ->(SPECIF t)

cpsalg < Unfold kappa in H3.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  d : nat
  y : nat
  H1 : (subtree_ersatz (leaf nat y) t)
       ->(kappa t (leaf nat y))
       ->(SPECIF t)
```

```
  H2 : (subtree_ersatz (leaf nat (S y)) t)
  H3 : (n':nat)n'=(leavemult (leaf nat (S y)))->(SPECIF t)
  ============================
   (SPECIF t)

subgoal 2 is:
 (t0:(tree nat))
  ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
  ->(t1:(tree nat))
     ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
     ->(subtree_ersatz (cons nat t0 t1) t)
     ->(kappa t (cons nat t0 t1))
     ->(SPECIF t)

cpsalg < Apply H3 with (S y).
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  d : nat
  y : nat
  H1 : (subtree_ersatz (leaf nat y) t)
       ->(kappa t (leaf nat y))
       ->(SPECIF t)
  H2 : (subtree_ersatz (leaf nat (S y)) t)
  H3 : (n':nat)n'=(leavemult (leaf nat (S y)))->(SPECIF t)
  ============================
   (S y)=(leavemult (leaf nat (S y)))

subgoal 2 is:
 (t0:(tree nat))
  ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
  ->(t1:(tree nat))
     ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
     ->(subtree_ersatz (cons nat t0 t1) t)
     ->(kappa t (cons nat t0 t1))
     ->(SPECIF t)

cpsalg < Auto; Trivial.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
```

```
  ============================
   (t0:(tree nat))
    ((subtree_ersatz t0 t)->(kappa t t0)->(SPECIF t))
    ->(t1:(tree nat))
       ((subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t))
       ->(subtree_ersatz (cons nat t0 t1) t)
       ->(kappa t (cons nat t0 t1))
       ->(SPECIF t)

cpsalg < Intro t1; Intro ind1; Intro t2; Intro ind2.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  ============================
   (subtree_ersatz (cons nat t1 t2) t)
   ->(kappa t (cons nat t1 t2))
   ->(SPECIF t)

cpsalg < Intros H H0.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (kappa t (cons nat t1 t2))
  ============================
   (SPECIF t)

cpsalg < Apply ind2.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
```

```
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (kappa t (cons nat t1 t2))
  ============================
   (subtree_ersatz t2 t)

subgoal 2 is:
 (kappa t t2)

cpsalg < Unfold subtree_ersatz.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (kappa t (cons nat t1 t2))
  ============================
   (has_zero t2)->(has_zero t)

subgoal 2 is:
 (kappa t t2)

cpsalg < Intro H1.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (kappa t (cons nat t1 t2))
  H1 : (has_zero t2)
  ============================
   (has_zero t)
```

```
subgoal 2 is:
 (kappa t t2)

cpsalg < Unfold subtree_ersatz in H; Apply H.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (has_zero (cons nat t1 t2))->(has_zero t)
  H0 : (kappa t (cons nat t1 t2))
  H1 : (has_zero t2)
  ============================
   (has_zero (cons nat t1 t2))

subgoal 2 is:
 (kappa t t2)

cpsalg < Simpl.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (has_zero (cons nat t1 t2))->(has_zero t)
  H0 : (kappa t (cons nat t1 t2))
  H1 : (has_zero t2)
  ============================
   (has_zero t1)\/(has_zero t2)

subgoal 2 is:
 (kappa t t2)

cpsalg < Auto.
1 subgoal
```

```
  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (kappa t (cons nat t1 t2))
  ============================
   (kappa t t2)

cpsalg < Unfold kappa.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (kappa t (cons nat t1 t2))
  ============================
   (n':nat)n'=(leavemult t2)->(SPECIF t)

cpsalg < Unfold kappa in H0.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  ============================
   (n':nat)n'=(leavemult t2)->(SPECIF t)

cpsalg < Intros n2 eg2.
1 subgoal
```

```
  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  ============================
   (SPECIF t)

cpsalg < Apply ind1.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  ============================
   (subtree_ersatz t1 t)

subgoal 2 is:
 (kappa t t1)

cpsalg < Unfold subtree_ersatz.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
```

```
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  ============================
   (has_zero t1)->(has_zero t)

subgoal 2 is:
 (kappa t t1)

cpsalg < Intro.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  H1 : (has_zero t1)
  ============================
   (has_zero t)

subgoal 2 is:
 (kappa t t1)

cpsalg < Apply H.
2 subgoals

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  H1 : (has_zero t1)
```

```
  ============================
   (has_zero (cons nat t1 t2))

subgoal 2 is:
 (kappa t t1)

cpsalg < Simpl; Auto.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  ============================
   (kappa t t1)

cpsalg < Unfold kappa.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  ============================
   (n':nat)n'=(leavemult t1)->(SPECIF t)

cpsalg < Intros n1 eg1.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
```

```
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  n1 : nat
  eg1 : n1=(leavemult t1)
  ============================
   (SPECIF t)

cpsalg < Apply (H0 (mult n1 n2)).
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  n1 : nat
  eg1 : n1=(leavemult t1)
  ============================
   (mult n1 n2)=(leavemult (cons nat t1 t2))

cpsalg < Simpl.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
```

```
  n2 : nat
  eg2 : n2=(leavemult t2)
  n1 : nat
  eg1 : n1=(leavemult t1)
  ============================
   (mult n1 n2)=(mult (leavemult t1) (leavemult t2))

cpsalg < Rewrite eg2; Rewrite eg1.
1 subgoal

  t : nat_tree
  ESCAPE_0 : (has_zero t)->(SPECIF t)
  t' : nat_tree
  t1 : (tree nat)
  ind1 : (subtree_ersatz t1 t)->(kappa t t1)->(SPECIF t)
  t2 : (tree nat)
  ind2 : (subtree_ersatz t2 t)->(kappa t t2)->(SPECIF t)
  H : (subtree_ersatz (cons nat t1 t2) t)
  H0 : (n':nat)n'=(leavemult (cons nat t1 t2))->(SPECIF t)
  n2 : nat
  eg2 : n2=(leavemult t2)
  n1 : nat
  eg1 : n1=(leavemult t1)
  ============================
   (mult (leavemult t1) (leavemult t2))
    =(mult (leavemult t1) (leavemult t2))

cpsalg < Auto.
Subtree proved!
```

De la extracción (`Recursive Extraction cpsalg.`) resulta:

```
type 'Dom tree =
    leaf of 'Dom
  | cons of 'Dom tree * 'Dom tree

let tree_rec f0 f =
  let rec f1 = function
    leaf d -> f0 d
  | cons (t1, t0) -> f t1 (f1 t1) t0 (f1 t0)
  in f1

type nat =
    O
  | S of nat
```

```
let nat_rec f0 f =
  let rec f1 = function
    O -> f0
  | S n0 -> f n0 (f1 n0)
  in f1

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

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

let cpsalg t =
  tree_rec (fun d ->
    nat_rec (fun _ h0 -> O) (fun y h1 _ h3 -> h3 (S y) prop) d)
    (fun t1 ind1 t2 ind2 _ h0 ->
    ind2 prop (fun n2 _ ->
      ind1 prop (fun n1 _ -> h0 (mult n1 n2) prop))) t prop
        (fun n _ -> n)
```

En conclusión, hemos comprobado que un mismo lema puede tener varias demostraciones y, por lo tanto, ser implementado mediante diferentes programas. La diferencia entre un informático y un matemático es que a éste le interesa más la definición que la prueba y, por tanto, para él todas las pruebas de un lema son *iguales* a efectos prácticos, mientras que para un informático no, porque se pueden implementar de distinto modo.

3.3. Consideraciones sobre la recursividad. Relaciones *Well-Founded*

Dada una *relación*

$$R : A \to A \to \text{Prop}$$

tenemos que $(R \quad a \quad b)$ es una *proposición* que será *verdadera* o *falsa* según el par (a, b) esté o no en la relación R.

Podemos pensar en una propiedad $P : A \to \text{Prop}$ definida sobre elementos de A: si dado un elemento b, si todos los elementos que se relacionan con b cumplen una propiedad

y de ello puede deducirse que también b la cumple, entonces *todos* los elementos cumplen dicha propiedad.

Esto no es sino una expresión muy general del **principio de inducción** (si todos los números anteriores a uno dado cumplen una propiedad y de ello se deduce que el siguiente la cumple, entonces todos los números cumplen dicha propiedad), aunque ya no es *lineal*. Este modo de inducción puede aplicarse, por ejemplo, a un árbol. En ese caso decimos que tenemos una estructura **well-founded**.

Formalmente, R es WF si:

$$\forall P : A \to \text{Prop}\ (\forall x : A \quad (\forall y : A \quad (R \quad y \quad x) \to (P \quad y)) \to (P \quad x)) \to \forall a : A \quad (P \quad a)$$

En `Coq`:

```
WF : (P:A->Prop) ((x:A)((y:A) (R y x)->(P y))->(P x)) -> (a:A)(P a).
```

(habiendo declarado previamente `Variable A:Set` y `Variable R:A->A->Prop`).

Podemos pensarlo de otra manera, definiendo un tipo de dato que especifique la siguiente propiedad: si una relación R es WF, no se puede encontrar una *cadena infinita* descendente de elementos relacionados, esto es, siempre ha de haber una base, un término (el cero en los naturales, las hojas en los árboles...).

El tipo inductivo que especifica esta propiedad es[13]:

```
Inductive ACC [A:Set][R:A->A->Prop]:A->Prop :=
  ACC_intro : (x:A)((y:A)(R y x)->(ACC A R y))->(ACC A R x).
```

y la nueva definición de WF:

```
WF' [A:Set][R:A->A->Prop] : (a:A)(ACC A R a)
```

es decir, R es WF si todos los elementos de A son *accesibles* mediante R.

Para comprobar que ambas definiciones son equivalentes, habría que demostrar cada una de ellas (por separado), teniendo en cada caso la otra como hipótesis.

[13]De hecho, ya existe en el sistema la función `acc`.

3.3.1. Propiedades de las relaciones *well-founded*

Una relación WF nunca es **reflexiva** (pues un elemento relacionado consigo mismo no sería accesible):

```
Lemma noreflex : (a:A) ~(R a a).
```

Esto, aunque se sobreentienda, hay que demostrarlo. Para ello necesitamos una **hipótesis de decidibilidad** (sin ella no podemos demostrar si la relación tiene bucles o no):

```
Hypothesis decR : (a,b:A) {(R a b)} + {~(R a b)}.
```

Entonces tendríamos que probar:

```
Lemma clave: (x:A) ((y:A)(R y x) -> ~(R y y)) -> ~(R x x).
```

Aplicando `decR`, o x está relacionado consigo mismo o no lo está. Si no lo está, ya quedaría demostrado y si lo está, si se da `(R x x)` es que x es uno de los y que están relacionados con x y por hipótesis $\forall y$ `(R y y)`, lo cual es una contradicción.

4. Otros resultados

4.1. Sobre la igualdad de números naturales

```
Coq < Fixpoint egal_nat [n:nat]:nat->bool :=
Coq < [m:nat]
Coq < Cases n m of
Coq < O O => true
Coq < | (S n) (S m) => (egal_nat n m)
Coq < | _ _ => false
Coq < end.
egal_nat is recursively defined

Coq < Lemma refl_egal: (x:nat)(egal_nat x x)=true.
1 subgoal

  ============================
   (x:nat)(egal_nat x x)=true

refl_egal < Induction x.
2 subgoals

  x : nat
  ============================
   (egal_nat O O)=true

subgoal 2 is:
 (n:nat)(egal_nat n n)=true->(egal_nat (S n) (S n))=true

refl_egal < Simpl.
2 subgoals

  x : nat
  ============================
   true=true

subgoal 2 is:
 (n:nat)(egal_nat n n)=true->(egal_nat (S n) (S n))=true

refl_egal < Trivial.
1 subgoal

  x : nat
  ============================
   (n:nat)(egal_nat n n)=true->(egal_nat (S n) (S n))=true
```

```
refl_egal < Intros.
1 subgoal

  x : nat
  n : nat
  H : (egal_nat n n)=true
  ============================
   (egal_nat (S n) (S n))=true

refl_egal < Elim H.
1 subgoal

  x : nat
  n : nat
  H : (egal_nat n n)=true
  ============================
   (egal_nat (S n) (S n))=(egal_nat n n)

refl_egal < Simpl.
1 subgoal

  x : nat
  n : nat
  H : (egal_nat n n)=true
  ============================
   (egal_nat n n)=(egal_nat n n)

refl_egal < Trivial.
Subtree proved!

refl_egal < Qed.
Induction x.
Simpl.
Trivial.

Intros.
Elim H.
Simpl.
Trivial.

refl_egal is defined
```

```
Coq < Lemma dec_egal_nat_d : (n,m:nat) (egal_nat n m)=true -> n=m.
1 subgoal

  ============================
   (n,m:nat)(egal_nat n m)=true->n=m

dec_egal_nat_d < Induction n; Induction m.
4 subgoals

  n : nat
  m : nat
  ============================
   (egal_nat O O)=true->O=O

subgoal 2 is:
 (n0:nat)
  ((egal_nat O n0)=true->O=n0)->(egal_nat O (S n0))=true->O=(S n0)
subgoal 3 is:
 (egal_nat (S n0) O)=true->(S n0)=O
subgoal 4 is:
 (n1:nat)
  ((egal_nat (S n0) n1)=true->(S n0)=n1)
  ->(egal_nat (S n0) (S n1))=true
  ->(S n0)=(S n1)

dec_egal_nat_d < Simpl.
4 subgoals

  n : nat
  m : nat
  ============================
   true=true->O=O

subgoal 2 is:
 (n0:nat)
  ((egal_nat O n0)=true->O=n0)->(egal_nat O (S n0))=true->O=(S n0)
subgoal 3 is:
 (egal_nat (S n0) O)=true->(S n0)=O
subgoal 4 is:
 (n1:nat)
  ((egal_nat (S n0) n1)=true->(S n0)=n1)
  ->(egal_nat (S n0) (S n1))=true
  ->(S n0)=(S n1)
```

```
dec_egal_nat_d < Intro.
4 subgoals

  n : nat
  m : nat
  H : true=true
  ============================
   0=0

subgoal 2 is:
 (n0:nat)
  ((egal_nat 0 n0)=true->0=n0)->(egal_nat 0 (S n0))=true->0=(S n0)
subgoal 3 is:
 (egal_nat (S n0) 0)=true->(S n0)=0
subgoal 4 is:
 (n1:nat)
  ((egal_nat (S n0) n1)=true->(S n0)=n1)
  ->(egal_nat (S n0) (S n1))=true
  ->(S n0)=(S n1)

dec_egal_nat_d < Trivial.
3 subgoals

  n : nat
  m : nat
  ============================
   (n0:nat)
    ((egal_nat 0 n0)=true->0=n0)->(egal_nat 0 (S n0))=true->0=(S n0)

subgoal 2 is:
 (egal_nat (S n0) 0)=true->(S n0)=0
subgoal 3 is:
 (n1:nat)
  ((egal_nat (S n0) n1)=true->(S n0)=n1)
  ->(egal_nat (S n0) (S n1))=true
  ->(S n0)=(S n1)

dec_egal_nat_d < Clear n.
3 subgoals

  m : nat
  ============================
   (n:nat)((egal_nat 0 n)=true->0=n)->(egal_nat 0 (S n))=true->0=(S n)
```

```
subgoal 2 is:
 (egal_nat (S n0) O)=true->(S n0)=O
subgoal 3 is:
 (n1:nat)
  ((egal_nat (S n0) n1)=true->(S n0)=n1)
  ->(egal_nat (S n0) (S n1))=true
  ->(S n0)=(S n1)

dec_egal_nat_d < Intros.
3 subgoals

  m : nat
  n : nat
  H : (egal_nat O n)=true->O=n
  H0 : (egal_nat O (S n))=true
  ============================
   O=(S n)

subgoal 2 is:
 (egal_nat (S n0) O)=true->(S n0)=O
subgoal 3 is:
 (n1:nat)
  ((egal_nat (S n0) n1)=true->(S n0)=n1)
  ->(egal_nat (S n0) (S n1))=true
  ->(S n0)=(S n1)

dec_egal_nat_d < Discriminate.
2 subgoals

  n : nat
  n0 : nat
  H : (m:nat)(egal_nat n0 m)=true->n0=m
  m : nat
  ============================
   (egal_nat (S n0) O)=true->(S n0)=O

subgoal 2 is:
 (n1:nat)
  ((egal_nat (S n0) n1)=true->(S n0)=n1)
  ->(egal_nat (S n0) (S n1))=true
  ->(S n0)=(S n1)

dec_egal_nat_d < Intro.
2 subgoals
```

```
  n : nat
  n0 : nat
  H : (m:nat)(egal_nat n0 m)=true->n0=m
  m : nat
  H0 : (egal_nat (S n0) 0)=true
  ============================
   (S n0)=0

subgoal 2 is:
 (n1:nat)
  ((egal_nat (S n0) n1)=true->(S n0)=n1)
  ->(egal_nat (S n0) (S n1))=true
  ->(S n0)=(S n1)

dec_egal_nat_d < Discriminate.
1 subgoal

  n : nat
  n0 : nat
  H : (m:nat)(egal_nat n0 m)=true->n0=m
  m : nat
  ============================
   (n1:nat)
    ((egal_nat (S n0) n1)=true->(S n0)=n1)
    ->(egal_nat (S n0) (S n1))=true
    ->(S n0)=(S n1)

dec_egal_nat_d < Clear n.
1 subgoal

  n0 : nat
  H : (m:nat)(egal_nat n0 m)=true->n0=m
  m : nat
  ============================
   (n:nat)
    ((egal_nat (S n0) n)=true->(S n0)=n)
    ->(egal_nat (S n0) (S n))=true
    ->(S n0)=(S n)

dec_egal_nat_d < Intro.
1 subgoal

  n0 : nat
  H : (m:nat)(egal_nat n0 m)=true->n0=m
  m : nat
```

```
  n : nat
  ============================
   ((egal_nat (S n0) n)=true->(S n0)=n)
   ->(egal_nat (S n0) (S n))=true
   ->(S n0)=(S n)

dec_egal_nat_d < Intro.
1 subgoal

  n0 : nat
  H : (m:nat)(egal_nat n0 m)=true->n0=m
  m : nat
  n : nat
  H0 : (egal_nat (S n0) n)=true->(S n0)=n
  ============================
   (egal_nat (S n0) (S n))=true->(S n0)=(S n)

dec_egal_nat_d < Simpl.
1 subgoal

  n0 : nat
  H : (m:nat)(egal_nat n0 m)=true->n0=m
  m : nat
  n : nat
  H0 : (egal_nat (S n0) n)=true->(S n0)=n
  ============================
   (egal_nat n0 n)=true->(S n0)=(S n)

dec_egal_nat_d < Intro.
1 subgoal

  n0 : nat
  H : (m:nat)(egal_nat n0 m)=true->n0=m
  m : nat
  n : nat
  H0 : (egal_nat (S n0) n)=true->(S n0)=n
  H1 : (egal_nat n0 n)=true
  ============================
   (S n0)=(S n)

dec_egal_nat_d < Elim (H n H1).
1 subgoal

  n0 : nat
  H : (m:nat)(egal_nat n0 m)=true->n0=m
```

```
  m : nat
  n : nat
  H0 : (egal_nat (S n0) n)=true->(S n0)=n
  H1 : (egal_nat n0 n)=true
  ============================
   (S n0)=(S n0)

dec_egal_nat_d < Trivial.
Subtree proved!

dec_egal_nat_d < Qed.
Induction n; Induction m.
Simpl.
Intro.
Trivial.

Clear n.
Intros.
Discriminate.

Intro.
Discriminate.

Clear n.
Intro.
Intro.
Simpl.
Intro.
Elim (H n H1).
Trivial.

dec_egal_nat_d is defined




Coq < Lemma dec_egal_nat_inv : (n,m:nat) n=m -> (egal_nat n m)=true.
1 subgoal

  ============================
   (n,m:nat)n=m->(egal_nat n m)=true

dec_egal_nat_inv < Intros.
1 subgoal
```

```
  n : nat
  m : nat
  H : n=m
  ============================
   (egal_nat n m)=true

dec_egal_nat_inv < Elim H.
1 subgoal

  n : nat
  m : nat
  H : n=m
  ============================
   (egal_nat n n)=true

dec_egal_nat_inv < Apply refl_egal.
Subtree proved!

dec_egal_nat_inv < Qed.
Intros.
Elim H.
Apply refl_egal.

dec_egal_nat_inv is defined
```

Otra forma, no tan directa:

```
Coq < Lemma dec_egal_nat_inv : (n,m:nat) n=m -> (egal_nat n m)=true.
1 subgoal

  ============================
   (n,m:nat)n=m->(egal_nat n m)=true

dec_egal_nat_inv < Intros.
1 subgoal

  n : nat
  m : nat
  H : n=m
  ============================
   (egal_nat n m)=true

dec_egal_nat_inv < Rewrite H.
1 subgoal
```

```
  n : nat
  m : nat
  H : n=m
  ============================
   (egal_nat m m)=true

dec_egal_nat_inv < Elim m.
2 subgoals

  n : nat
  m : nat
  H : n=m
  ============================
   (egal_nat O O)=true

subgoal 2 is:
 (n0:nat)(egal_nat n0 n0)=true->(egal_nat (S n0) (S n0))=true

dec_egal_nat_inv < Simpl.
2 subgoals

  n : nat
  m : nat
  H : n=m
  ============================
   true=true

subgoal 2 is:
 (n0:nat)(egal_nat n0 n0)=true->(egal_nat (S n0) (S n0))=true

dec_egal_nat_inv < Trivial.
1 subgoal

  n : nat
  m : nat
  H : n=m
  ============================
   (n0:nat)(egal_nat n0 n0)=true->(egal_nat (S n0) (S n0))=true

dec_egal_nat_inv < Intros.
1 subgoal

  n : nat
  m : nat
```

```
  H : n=m
  n0 : nat
  H0 : (egal_nat n0 n0)=true
  ============================
   (egal_nat (S n0) (S n0))=true

dec_egal_nat_inv < Simpl.
1 subgoal

  n : nat
  m : nat
  H : n=m
  n0 : nat
  H0 : (egal_nat n0 n0)=true
  ============================
   (egal_nat n0 n0)=true

dec_egal_nat_inv < Assumption.
Subtree proved!

dec_egal_nat_inv < Qed.
Intros.
Rewrite H.
Elim m.
Simpl.
Trivial.

Intros.
Simpl.
Assumption.

dec_egal_nat_inv is defined




Coq < Lemma dec_egal_nat : (n,m:nat){(egal_nat n m)=true}
Coq <                            + {(egal_nat n m)=false}.
1 subgoal

  ============================
   (n,m:nat){(egal_nat n m)=true}+{(egal_nat n m)=false}
```

```
dec_egal_nat < Induction n; Induction m.
4 subgoals

  n : nat
  m : nat
  ============================
   {(egal_nat O O)=true}+{(egal_nat O O)=false}

subgoal 2 is:
 (n0:nat)
  {(egal_nat O n0)=true}+{(egal_nat O n0)=false}
  ->{(egal_nat O (S n0))=true}+{(egal_nat O (S n0))=false}
subgoal 3 is:
 {(egal_nat (S n0) O)=true}+{(egal_nat (S n0) O)=false}
subgoal 4 is:
 (n1:nat)
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Simpl.
4 subgoals

  n : nat
  m : nat
  ============================
   {true=true}+{true=false}

subgoal 2 is:
 (n0:nat)
  {(egal_nat O n0)=true}+{(egal_nat O n0)=false}
  ->{(egal_nat O (S n0))=true}+{(egal_nat O (S n0))=false}
subgoal 3 is:
 {(egal_nat (S n0) O)=true}+{(egal_nat (S n0) O)=false}
subgoal 4 is:
 (n1:nat)
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Left.
4 subgoals

  n : nat
  m : nat
  ============================
   true=true
```

```
subgoal 2 is:
 (n0:nat)
  {(egal_nat O n0)=true}+{(egal_nat O n0)=false}
  ->{(egal_nat O (S n0))=true}+{(egal_nat O (S n0))=false}
subgoal 3 is:
 {(egal_nat (S n0) O)=true}+{(egal_nat (S n0) O)=false}
subgoal 4 is:
 (n1:nat)
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Trivial.
3 subgoals

  n : nat
  m : nat
  ============================
   (n0:nat)
    {(egal_nat O n0)=true}+{(egal_nat O n0)=false}
    ->{(egal_nat O (S n0))=true}+{(egal_nat O (S n0))=false}

subgoal 2 is:
 {(egal_nat (S n0) O)=true}+{(egal_nat (S n0) O)=false}
subgoal 3 is:
 (n1:nat)
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Clear n.
3 subgoals

  m : nat
  ============================
   (n:nat)
    {(egal_nat O n)=true}+{(egal_nat O n)=false}
    ->{(egal_nat O (S n))=true}+{(egal_nat O (S n))=false}

subgoal 2 is:
 {(egal_nat (S n0) O)=true}+{(egal_nat (S n0) O)=false}
subgoal 3 is:
 (n1:nat)
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}
```

```
dec_egal_nat < Intros.
3 subgoals

  m : nat
  n : nat
  H : {(egal_nat O n)=true}+{(egal_nat O n)=false}
  ============================
   {(egal_nat O (S n))=true}+{(egal_nat O (S n))=false}

subgoal 2 is:
 {(egal_nat (S n0) O)=true}+{(egal_nat (S n0) O)=false}
subgoal 3 is:
 (n1:nat)
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Simpl.
3 subgoals

  m : nat
  n : nat
  H : {(egal_nat O n)=true}+{(egal_nat O n)=false}
  ============================
   {false=true}+{false=false}

subgoal 2 is:
 {(egal_nat (S n0) O)=true}+{(egal_nat (S n0) O)=false}
subgoal 3 is:
 (n1:nat)
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Right.
3 subgoals

  m : nat
  n : nat
  H : {(egal_nat O n)=true}+{(egal_nat O n)=false}
  ============================
   false=false

subgoal 2 is:
 {(egal_nat (S n0) O)=true}+{(egal_nat (S n0) O)=false}
subgoal 3 is:
 (n1:nat)
```

```
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Trivial.
2 subgoals

  n : nat
  n0 : nat
  H : (m:nat){(egal_nat n0 m)=true}+{(egal_nat n0 m)=false}
  m : nat
  ============================
   {(egal_nat (S n0) O)=true}+{(egal_nat (S n0) O)=false}

subgoal 2 is:
 (n1:nat)
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Simpl.
2 subgoals

  n : nat
  n0 : nat
  H : (m:nat){(egal_nat n0 m)=true}+{(egal_nat n0 m)=false}
  m : nat
  ============================
   {false=true}+{false=false}

subgoal 2 is:
 (n1:nat)
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Right.
2 subgoals

  n : nat
  n0 : nat
  H : (m:nat){(egal_nat n0 m)=true}+{(egal_nat n0 m)=false}
  m : nat
  ============================
   false=false

subgoal 2 is:
 (n1:nat)
```

```
  {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
  ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Trivial.
1 subgoal

  n : nat
  n0 : nat
  H : (m:nat){(egal_nat n0 m)=true}+{(egal_nat n0 m)=false}
  m : nat
  ============================
   (n1:nat)
    {(egal_nat (S n0) n1)=true}+{(egal_nat (S n0) n1)=false}
    ->{(egal_nat (S n0) (S n1))=true}+{(egal_nat (S n0) (S n1))=false}

dec_egal_nat < Clear n.
1 subgoal

  n0 : nat
  H : (m:nat){(egal_nat n0 m)=true}+{(egal_nat n0 m)=false}
  m : nat
  ============================
   (n:nat)
    {(egal_nat (S n0) n)=true}+{(egal_nat (S n0) n)=false}
    ->{(egal_nat (S n0) (S n))=true}+{(egal_nat (S n0) (S n))=false}

dec_egal_nat < Intros.
1 subgoal

  n0 : nat
  H : (m:nat){(egal_nat n0 m)=true}+{(egal_nat n0 m)=false}
  m : nat
  n : nat
  H0 : {(egal_nat (S n0) n)=true}+{(egal_nat (S n0) n)=false}
  ============================
   {(egal_nat (S n0) (S n))=true}+{(egal_nat (S n0) (S n))=false}

dec_egal_nat < Simpl.
1 subgoal

  n0 : nat
  H : (m:nat){(egal_nat n0 m)=true}+{(egal_nat n0 m)=false}
  m : nat
  n : nat
  H0 : {(egal_nat (S n0) n)=true}+{(egal_nat (S n0) n)=false}
```

```
  ============================
   {(egal_nat n0 n)=true}+{(egal_nat n0 n)=false}

dec_egal_nat < Apply (H n).
Subtree proved!

dec_egal_nat < Qed.
Induction n; Induction m.
Simpl.
Left.
Trivial.

Clear n.
Intros.
Simpl.
Right.
Trivial.

Simpl.
Right.
Trivial.

Clear n.
Intros.
Simpl.
Apply (H n).

dec_egal_nat is defined
```

4.2. Algunas otras cosas generales

Sea el teorema:

```
Theorem Resolution : (P,Q:Type->Prop) (a:Type)(P a)->
   ((x:Type)(P x) -> (Q x)) -> (Q a).
```

Vamos a demostrarlo utilizando sólo `Apply` y `Exact`. Para ello aplicamos que para todo x si $(P\ \ x)$ entonces $(Q\ \ x)$, lo que nos deja como objetivo $(P\ \ a)$. Como $(P\ \ a)$ ya está en el contexto con nombre H, se termina la demostración con un `Exact H`.

```
1 subgoal

  ============================
   (P,Q:(Type->Prop); a:Type)(P a)->((x:Type)(P x)->(Q x))->(Q a)
```

```
Resolution < Intros.
1 subgoal

  P : Type->Prop
  Q : Type->Prop
  a : Type
  H : (P a)
  H0 : (x:Type)(P x)->(Q x)
  ============================
   (Q a)

Resolution < Apply H0.
1 subgoal

  P : Type->Prop
  Q : Type->Prop
  a : Type
  H : (P a)
  H0 : (x:Type)(P x)->(Q x)
  ============================
   (P a)

Resolution < Exact H.
Subtree proved!
```

Definición de los números naturales, con dos constructores, el del cero y el que da el sucesor de un natural:

```
Coq < Inductive Nat:Set:=O:Nat | S:Nat->Nat.
Nat_ind is defined
Nat_rec is defined
Nat_rect is defined
Nat is defined
```

Definición del `and` para dos variables A y B concretas.

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

Coq < Inductive and:Prop:=Conj:A->B->and.
and_ind is defined
and_rec is defined
```

```
and is defined
```

Esta definición del `and` es válida para dos X,Y cualesquiera, es un tipo inductivo con parámetros. El constructor toma una prueba de X, una prueba de Y, y nos da una prueba del `and`.

```
Coq < Inductive And[X:Prop;Y:Prop]:Prop:=conj:X->Y->(And X Y).
And_ind is defined
And_rec is defined
And is defined
```

Demostración de la conmutatividad del `and`. Primero se introducen las variables en el contexto, Se hace inducción sobre el `and`, lo que nos da una prueba de cada lado del `and`. La táctica `Split` a partir de un `and` deja dos subobjetivos, la parte izquierda, y la derecha (viene a ser un `Apply conj`). Con esto quedan como subobjetivos dos cosas que ya tenemos.

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

  ============================
   (A,B:Prop)A/\B->B/\A

And_Commutativo < Intros.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
  ============================
   B/\A

And_Commutativo < Elim H.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
  ============================
   A->B->B/\A

And_Commutativo < Intros.
1 subgoal
```

```
  A : Prop
  B : Prop
  H : A/\B
  H0 : A
  H1 : B
  ============================
   B/\A

And_Commutativo < Split.
2 subgoals

  A : Prop
  B : Prop
  H : A/\B
  H0 : A
  H1 : B
  ============================
   B

subgoal 2 is:
 A

And_Commutativo < Assumption.
1 subgoal

  A : Prop
  B : Prop
  H : A/\B
  H0 : A
  H1 : B
  ============================
   A

And_Commutativo < Assumption.
Subtree proved!
```

Demostración de la conmutatividad del `or`. Esta demostración es similar a la anterior con un par de matices: la inducción sobre el `or` nos deja dos subojetivos, demostrar el original suponiendo la parte izquierda cierta, y demostrarlo suponiendo la derecha. Con las tácticas `Left` y `Right` dejamos como objetivo demostrar una de las dos partes del `or` (`Apply or_left` y `Apply or_right` respectivamente).

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

```
  ============================
   (A,B:Prop)A\/B->B\/A

Or_Commutativo < Intros.
1 subgoal

  A : Prop
  B : Prop
  H : A\/B
  ============================
   B\/A

Or_Commutativo < Elim H.
2 subgoals

  A : Prop
  B : Prop
  H : A\/B
  ============================
   A->B\/A

subgoal 2 is:
 B->B\/A

Or_Commutativo < Intros.
2 subgoals

  A : Prop
  B : Prop
  H : A\/B
  H0 : A
  ============================
   B\/A

subgoal 2 is:
 B->B\/A

Or_Commutativo < Right.
2 subgoals

  A : Prop
  B : Prop
  H : A\/B
  H0 : A
  ============================
```

```
   A

subgoal 2 is:
 B->B\/A

Or_Commutativo < Assumption.
1 subgoal

  A : Prop
  B : Prop
  H : A\/B
  ============================
   B->B\/A

Or_Commutativo < Intros.
1 subgoal

  A : Prop
  B : Prop
  H : A\/B
  H0 : B
  ============================
   B\/A

Or_Commutativo < Left.
1 subgoal

  A : Prop
  B : Prop
  H : A\/B
  H0 : B
  ============================
   B

Or_Commutativo < Assumption.
Subtree proved!
```

Definición de la suma. Decimos que es una función que a un `nat` siempre le devuelve una función de `nat` en `nat`, que sumar 0 a algo es la función identidad, y que para pasar de la función que suma n a la que suma $(S \quad n)$ sólo hay que aplicar el sucesor al resultado.

```
Coq < Definition suma:=(nat_rec ([_:nat](nat->nat))
Coq <                           ([x:nat]x)
Coq <                           ([_:nat][f:nat->nat][x:nat](S (f x)))).
suma is defined
```

Estas son las propiedades de la suma, sumar 0 a un número da ese número, y si la suma de p y q es r, la suma de $(S\quad p)$ y q es $(S\quad r)$.

```
Coq < Inductive Suma:nat->nat->nat->Prop:=
Coq < SumaO:(n:nat)(Suma O n n)
Coq < | SumaS:(p,q,r:nat)(Suma p q r)->(Suma (S p) q (S r)).
Suma_ind is defined
Suma is defined
```

Vamos a comprobar que la función que definimos antes cumple estas propiedades. Para ello se hace inducción sobre m, que nos deja como objetivos demostrar la suma con el cero, y suponiendo que la suma de n es correcta, demostrar la suma de $n+1$. Se hace que evalúe la función definida, y deja el axioma `sumaO`.

El otro subobjetivo se demuestra haciendo evaluar la función, aplicando `sumaS`, con lo que nos queda demostrar la parte izquierda de `sumaS`, que ya está en el contexto.

```
Coq < Lemma Testsuma:(m,n:nat)(Suma m n (suma m n)).
1 subgoal

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

Testsuma < Intros.
1 subgoal

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

Testsuma < Elim m.
2 subgoals

  m : nat
  n : nat
  ============================
   (Suma O n (suma O n))

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

Testsuma < Simpl.
2 subgoals
```

```
  m : nat
  n : nat
  ============================
   (Suma O n n)

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

Testsuma < Apply SumaO.
1 subgoal

  m : nat
  n : nat
  ============================
   (n0:nat)(Suma n0 n (suma n0 n))->(Suma (S n0) n (suma (S n0) n))

Testsuma < Intros.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (Suma n0 n (suma n0 n))
  ============================
   (Suma (S n0) n (suma (S n0) n))

Testsuma < Simpl.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (Suma n0 n (suma n0 n))
  ============================
   (Suma (S n0) n (S (suma n0 n)))

Testsuma < Apply SumaS.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (Suma n0 n (suma n0 n))
  ============================
```

```
   (Suma n0 n (suma n0 n))

Testsuma < Assumption.
Subtree proved!
```

Definición de la función predecesor. Es una función de `nat->nat`, que en 0 vale 0, y que para pasar de la función en n a la función en $n+1$ sólo tiene que devolver n.

```
Coq < Definition Pred:=(nat_rec ([_:nat]nat) (O) ([x:nat][y:nat]x)).
Pred is defined
```

Especificación de la función predecesor. Decimos que el predecesor de 0 es 0, y que el predecesor del sucesor de un número es el propio número.

```
Coq < Inductive Predecesor:nat->nat->Prop:=
Coq < PredecO:(Predecesor O O)
Coq < | PredecS:(n:nat)(Predecesor (S n) n).
Predecesor_ind is defined
Predecesor is defined
```

Ahora comprobamos que la función predecesor definida antes cumple la especificación. Se hace inducción sobre m, que deja dos subobjetivos, como simpre con `nat`: demostrar para el 0, y suponiendo cierto el de n, para $(S \quad n)$.

Al evaluar, como el predecesor de 0 es el 0, ya queda el axioma `PredecO`. El segundo subobjetivo, al evaluar `predecesor (S n)` ya devuelve n, por lo que queda el axioma `predecS`.

```
Coq < Lemma Testpred:(m:nat)(Predecesor m (Pred m)).
1 subgoal

  ============================
   (m:nat)(Predecesor m (Pred m))

Testpred < Intros.
1 subgoal

  m : nat
  ============================
   (Predecesor m (Pred m))

Testpred < Elim m.
2 subgoals
```

```
  m : nat
  ============================
   (Predecesor O (Pred O))

subgoal 2 is:
 (n:nat)(Predecesor n (Pred n))->(Predecesor (S n) (Pred (S n)))

Testpred < Simpl.
2 subgoals

  m : nat
  ============================
   (Predecesor O O)

subgoal 2 is:
 (n:nat)(Predecesor n (Pred n))->(Predecesor (S n) (Pred (S n)))

Testpred < Apply PredecO.
1 subgoal

  m : nat
  ============================
   (n:nat)(Predecesor n (Pred n))->(Predecesor (S n) (Pred (S n)))

Testpred < Intros.
1 subgoal

  m : nat
  n : nat
  H : (Predecesor n (Pred n))
  ============================
   (Predecesor (S n) (Pred (Sn)))

Testpred < Simpl.
1 subgoal

  m : nat
  n : nat
  H : (Predecesor n (Pred n))
  ============================
   (Predecesor (S n) n)

Testpred < Apply PredecS.
Subtree proved!
```

Definición del producto. Es una función que, al igual que la suma, devuelve siempre una función de `(nat->nat)`. El producto por cero siempre da cero, por lo que el producto en cero es la función constante 0. Para pasar de producto por n a producto por $(S \quad n)$ sólo hay que sumar el resultado otra vez.

```
Coq < Definition producto:nat->nat->nat:=(nat_rec
Coq <           ([_:nat](nat->nat))
Coq <           ([_:nat]0)
Coq <           ([n:nat][f:nat->nat][x:nat](plus x (f x)))).
Producto is defined
```

Especificación del producto. El producto por 0 es cero, y si el producto de m por n es p, el producto de $(S \quad m)$ por n es $(n + p)$.

```
Coq < Inductive Producto:nat->nat->nat->Prop:=
Coq < Producto0:(m:nat)(Producto 0 m 0)
Coq < | ProductoS:(m,n,p:nat)(Producto m n p)->(Producto (S m) n (plus n p)).
producto_ind is defined
producto is defined
```

Comprobación de que el producto cumple la especificación. Para esto se hace inducción sobre m, lo que vuelve a dejar dos subobjetivos, el del 0 y el del sucesor. Al evaluar en 0 el producto ya vale 0, lo que deja el axioma `Producto0`. Evaluando la función simplifica `(producto (S n0) n)` a `(plus n (producto n0 n))`. Con lo que queda podemos aplicar `ProductoS`, y nos queda algo que ya tenemos.

```
Coq < Lemma Testproducto:(m,n:nat)(Producto m n (producto m n)).
1 subgoal

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

Testproducto < Intros.
1 subgoal

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

Testproducto < Elim m.
2 subgoals

  m : nat
```

```
  n : nat
  ============================
   (Producto 0 n (producto 0 n))

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

Testproducto < Simpl.
2 subgoals

  m : nat
  n : nat
  ============================
   (Producto 0 n 0)

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

Testproducto < Apply Producto0.
1 subgoal

  m : nat
  n : nat
  ============================
   (n0:nat)
    (Producto n0 n (producto n0 n))
    ->(Producto (S n0) n (producto (S n0) n))

Testproducto < Intros.
1 subgoal

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

Testproducto < Simpl.
1 subgoal
```

```
  m : nat
  n : nat
  n0 : nat
  H : (Producto n0 n (producto n0 n))
  ============================
   (Producto (S n0) n (plus n (producto n0 n)))

Testproducto < Apply ProductoS.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (Producto n0 n (producto n0 n))
  ============================
   (Producto n0 n (producto n0 n))

Testproducto < Assumption.
Subtree proved!
```

Definición de la función potencia. Es una función que a un `nat` le devuelve una función de `nat->nat`. `(potencia 0)` es la función que eleva a cero, por tanto la función constante 1. Para pasar de elevar a n a elevar a $n+1$ sólo hay que multiplicar por el resultado.

```
Coq < Definition potencia:nat->nat->nat:=(nat_rec
Coq <             ([_:nat](nat->nat))
Coq <             ([_:nat](S O))
Coq <             ([n:nat][f:nat->nat][x:nat](producto x (f x)))).
potencia is defined
```

Especificación de la función potencia. Elevar a 0 da 1. Si y elevado a x es z, y elevado a $(S \quad x)$ es $y \times z$.

```
Coq < Inductive Potencia:nat->nat->nat->Prop:=
Coq < PotenciaO:(x:nat)(Potencia O x (S O))
Coq < | PotenciaS:(x,y,z:nat)(Potencia x y z)
Coq <                       -> (Potencia (S x) y (producto y z)).
Potencia_ind is defined
Potencia is defined
```

Vamos a comprobar que la función potencia cumple la especificación. Para ello hacemos inducción sobre x. Nos deja como objetivos demostrar para el 0 y para el sucesor. Si evaluamos `potencia` queda el axioma `PotenciaO`.

El segundo subobjetivo se demuestra evaluando la función, lo que pasa de escribir el término en función de $(S \quad n)$ a hacerlo en funcion de n. Aplicando `PotenciaS` nos queda la parte izquierda por demostrar, pero ya está en el contexto.

```
Coq < Lemma Testpotencia:(x,y:nat)(Potencia x y (potencia x y)).
1 subgoal

  ============================
   (x,y:nat)(Potencia x y (potencia x y))

Testpotencia < Intros.
1 subgoal

  x : nat
  y : nat
  ============================
   (Potencia x y (potencia x y))

Testpotencia < Elim x.
2 subgoals

  x : nat
  y : nat
  ============================
   (Potencia (0) y (potencia (0) y))

subgoal 2 is:
 (n:nat)
  (Potencia n y (potencia n y))->(Potencia (S n) y (potencia (S n) y))

Testpotencia < Simpl.
2 subgoals

  x : nat
  y : nat
  ============================
   (Potencia (0) y (1))

subgoal 2 is:
 (n:nat)
  (Potencia n y (potencia n y))->(Potencia (S n) y (potencia (S n) y))

Testpotencia < Apply PotenciaO.
1 subgoal
```

```
  x : nat
  y : nat
  ============================
   (n:nat)
    (Potencia n y (potencia n y))
    ->(Potencia (S n) y (potencia (S n) y))

Testpotencia < Intros.
1 subgoal

  x : nat
  y : nat
  n : nat
  H : (Potencia n y (potencia n y))
  ============================
   (Potencia (S n) y (potencia (S n) y))

Testpotencia < Simpl.
1 subgoal

  x : nat
  y : nat
  n : nat
  H : (Potencia n y (potencia n y))
  ============================
   (Potencia (S n) y (producto y (potencia n y)))

Testpotencia < Apply PotenciaS.
1 subgoal

  x : nat
  y : nat
  n : nat
  H : (Potencia n y (potencia n y))
  ============================
   (Potencia n y (potencia n y))

Testpotencia < Assumption.
Subtree proved!
```

Definición de la funcion factorial. Es una función que a un natural devuelve un natural. El factorial en 0 vale 1, y dado un número n, para obtener el factorial de $n+1$ se multiplica el resultado anterior por $n+1$.

```
Coq < Definition Factorial:nat->nat:=(nat_rec
```

```
Coq <            ([_:nat]nat)
Coq <            (S O)
Coq <            ([x:nat][y:nat](producto (S x) y))).
Factorial is defined
```

La especificación del factorial. El factorial de 0 es 1, y si el factorial de m es n, el de $m+1$ es $(m+1) \times n$.

```
Coq < Inductive factorial:nat->nat->Prop:=
Coq < factorial0:(factorial O (S O))
Coq < | factorialS:(m,n:nat)(factorial m n) ->
Coq <                       (factorial (S m) (producto (S m) n)).
factorial_ind is defined
factorial is defined
```

Vamos a comprobar que el factorial cumple la especificación. Se hace inducción sobre m. Al evaluar la función vale 1 (factorial en cero). Esto ya deja como objetivo el factorial para el sucesor. Para esto se evalúa la función, pero ponemos antes al producto como opaco, para que saque la suma al evaluar. Aplicando `factorialS` nos queda por demostrar su parte izquierda, que ya tenemos en el contexto.

```
Coq < Lemma Testfactorial:(m:nat)(factorial m (Factorial m)).
1 subgoal

  ============================
   (m:nat)(factorial m (Factorial m))

Testfactorial < Intros.
1 subgoal

  m : nat
  ============================
   (factorial m (Factorial m))

Testfactorial < Elim m.
2 subgoals

  m : nat
  ============================
   (factorial (0) (Factorial (0)))

subgoal 2 is:
 (n:nat)
  (factorial n (Factorial n))->(factorial (S n) (Factorial (S n)))
```

```
Testfactorial < Simpl.
2 subgoals

  m : nat
  ============================
   (factorial (0) (1))

subgoal 2 is:
 (n:nat)
  (factorial n (Factorial n))->(factorial (S n) (Factorial (S n)))

Testfactorial < Apply factorial0.
1 subgoal

  m : nat
  ============================
   (n:nat)
    (factorial n (Factorial n))->(factorial (S n) (Factorial (S n)))

Testfactorial < Intros.
1 subgoal

  m : nat
  n : nat
  H : (factorial n (Factorial n))
  ============================
   (factorial (S n) (Factorial (S n)))

Testfactorial < Opaque producto.

Testfactorial < Simpl.
1 subgoal

  m : nat
  n : nat
  H : (factorial n (Factorial n))
  ============================
   (factorial (S n) (producto (S n) (Factorial n)))

Testfactorial < Apply factorialS.
1 subgoal

  m : nat
  n : nat
```

```
  H : (factorial n (Factorial n))
  ============================
   (factorial n (Factorial n))

Testfactorial < Assumption.
Subtree proved!
```

Definición inductiva con parámetros de una lista. Define la lista nula, sin elementos, y `cons`, que a partir de un elemento y una lista nos da otra lista (la lista con el elemento añadido).

```
Coq < Inductive List [A:Set]:Set:=
Coq < Nil:(List A)
Coq < |Cons:A->(List A)->(List A).
List_ind is defined
List_rec is defined
List_rect is defined
List is defined
```

Especificación de la longitud de una lista. La longitud de la lista vacía es 0, y dada la longitud de una lista l, la longitud de l añadiéndole otro elemento es el sucesor.

```
Coq < Inductive long [A:Set]:(List A)->nat->Prop:=
Coq < longO:(long A (Nil A) O)
Coq < | longS:(a:A)(l:(List A))(n:nat)(long A l n)
Coq <                                -> (long A (Cons A a l) (S n)).
long_ind is defined
long is defined
```

Definición de la función longitud. Es una función que toma una lista y devuelve un `nat`. La longitud de la lista nula es 0, y data una lista de longitud n, la longitud de la lista añadiéndole otro elemento es el sucesor de n.

```
Coq < Definition Long:=[A:Set](List_rec
Coq <            (A)
Coq <            ([_:(List A)]nat)
Coq <            (O)
Coq <            ([_:A][_:(List A)][n:nat](S n))).
Long is defined
```

Vamos a comprobar que cumple la especificación. Se hace inducción sobre la lista, lo que deja dos subobjetivos, demostrar el teorema para la lista nula, y considerándolo cierto para una lista l, demostrarlo para l añadiendo otro elemento.

El primer subobjetivo se demuestra evaluando `Long`, que para nil vale 0, y nos deja como objetivo `Long0`, que se aplica.

El segundo subobjetivo se también demuestra evaluando `Long`, que para una lista construida con `cons` devuelve sucesor de la longitud de la lista original.

Ahora aplicando `LongS` nos queda su parte izquierda por demostrar, pero ya está en el contexto.

```
Coq < Lemma TestLong:(A:Set)(l:(List A))(long A l (Long A l)).
1 subgoal

  ============================
   (A:Set; l:(List A))(long A l (Long A l))

TestLong < Intros.
1 subgoal

  A : Set
  l : (List A)
  ============================
   (long A l (Long A l))

TestLong < Elim l.
2 subgoals

  A : Set
  l : (List A)
  ============================
   (long A (Nil A) (Long A (Nil A)))

subgoal 2 is:
 (y:A; l0:(List A))
  (long A l0 (Long A l0))
  ->(long A (Cons A y l0) (Long A (Cons A y l0)))

TestLong < Simpl.
2 subgoals

  A : Set
  l : (List A)
  ============================
   (long A (Nil A) O)

subgoal 2 is:
 (y:A; l0:(List A))
  (long A l0 (Long A l0))
```

```
  ->(long A (Cons A y l0) (Long A (Cons A y l0)))

TestLong < Apply longO.
1 subgoal

  A : Set
  l : (List A)
  ============================
   (y:A; l0:(List A))
    (long A l0 (Long A l0))
    ->(long A (Cons A y l0) (Long A (Cons A y l0)))

TestLong < Intros.
1 subgoal

  A : Se
  l : (List A)
  y : A
  l0 : (List A)
  H : (long A l0 (Long A l0))
  ============================
   (long A (Cons A y l0) (Long A (Cons A y l0)))

TestLong < Simpl.
1 subgoal

  A : Set
  l : (List A)
  y : A
  l0 : (List A)
  H : (long A l0 (Long A l0))
  ============================
   (long A (Cons A y l0) (S (Long A l0)))

TestLong < Apply longS.
1 subgoal

  A : Set
  l : (List A)
  y : A
  l0 : (List A)
  H : (long A l0 (Long A l0))
  ============================
   (long A l0 (Long A l0))
```

```
TestLong < Assumption.
Subtree proved!
```

Vamos a demostrar la conmutatividad de la suma. Para ello se hace inducción sobre las dos variables. Nos quedan 3 subobjetivos, aunque como el primero es demostrar que algo es igual a sí mismo ya se resuelve con un `Trivial`. El segundo se demuestra evaluando, lo que da el resultado de una de las sumas, y nos permite usar uno de los axiomas para reescribir el otro término de la igualdad, que al evaluarlo da también el resultado.

El tercer subobjetivo se demuestra evaluando, reescribiendo uno de los términos, lo que nos deja por demostrar que `(S (plus a b))=(plus a (S b))`.

Para demostrar esto se hace inducción sobre a. El caso del cero es casi inmediato, al evaluar deja lo mismo en los dos lados. El otro es un poco más complicado, hay que evaluar la función, lo que nos permite usar una igualdad del contexto para reescribir un término y dejar lo mismo en los dos lados.

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

  ============================
   (n,m:nat)(plus n m)=(plus m n)

Sumacom < Intros.
1 subgoal

  n : nat
  m : nat
  ============================
   (plus n m)=(plus m n)

Sumacom < Elim n.
2 subgoals

  n : nat
  m : nat
  ============================
   (plus O m)=(plus m O)

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

Sumacom < Elim m.
3 subgoals

  n : nat
```

```
  m : nat
  ============================
   (plus O O)=(plus O O)

subgoal 2 is:
 (n0:nat)(plus O n0)=(plus n0 O)->(plus O (S n0))=(plus (S n0) O)
subgoal 3 is:
 (n0:nat)(plus n0 m)=(plus m n0)->(plus (S n0) m)=(plus m (S n0))

Sumacom < Trivial.
2 subgoals

  n : nat
  m : nat
  ============================
   (n0:nat)(plus O n0)=(plus n0 O)->(plus O (S n0))=(plus (S n0) O)

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

Sumacom < Intros.
2 subgoals

  n : nat
  m : nat
  n0 : nat
  H : (plus O n0)=(plus n0 O)
  ============================
   (plus O (S n0))=(plus (S n0) O)

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

Sumacom < Simpl.
2 subgoals

  n : nat
  m : nat
  n0 : nat
  H : (plus O n0)=(plus n0 O)
  ============================
   (S n0)=(S (plus n0 O))

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

```
Sumacom < Rewrite <- H.
2 subgoals

  n : nat
  m : nat
  n0 : nat
  H : (plus O n0)=(plus n0 O)
  ============================
   (S n0)=(S (plus O n0))

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

Sumacom < Simpl.
2 subgoals

  n : nat
  m : nat
  n0 : nat
  H : (plus O n0)=(plus n0 O)
  ============================
   (S n0)=(S n0)

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

Sumacom < Trivial.
1 subgoal

  n : nat
  m : nat
  ============================
   (n0:nat)(plus n0 m)=(plus m n0)->(plus (S n0) m)=(plus m (S n0))

Sumacom < Intros.
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : (plus n0 m)=(plus m n0)
  ============================
   (plus (S n0) m)=(plus m (S n0))
```

```
Sumacom < Simpl.
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : (plus n0 m)=(plus m n0)
  ============================
   (S (plus n0 m))=(plus m (S n0))

Sumacom < Rewrite H.
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : (plus n0 m)=(plus m n0)
  ============================
   (S (plus m n0))=(plus m (S n0))

Sumacom < Elim m.
2 subgoals

  n : nat
  m : nat
  n0 : nat
  H : (plus n0 m)=(plus m n0)
  ============================
   (S (plus O n0))=(plus O (S n0))

subgoal 2 is:
 (n1:nat)
  (S (plus n1 n0))=(plus n1 (S n0))
  ->(S (plus (S n1) n0))=(plus (S n1) (S n0))

Sumacom < Simpl.
2 subgoals

  n : nat
  m : nat
  n0 : nat
  H : (plus n0 m)=(plus m n0)
  ============================
   (S n0)=(S n0)
```

```
subgoal 2 is:
 (n1:nat)
  (S (plus n1 n0))=(plus n1 (S n0))
  ->(S (plus (S n1) n0))=(plus (S n1) (S n0))

Sumacom < Trivial.
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : (plus n0 m)=(plus m n0)
  ============================
   (n1:nat)
    (S (plus n1 n0))=(plus n1 (S n0))
    ->(S (plus (S n1) n0))=(plus (S n1) (S n0))

Sumacom < Intros.
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : (plus n0 m)=(plus m n0)
  n1 : nat
  H0 : (S (plus n1 n0))=(plus n1 (S n0))
  ============================
   (S (plus (S n1) n0))=(plus (S n1) (S n0))

Sumacom < Simpl.
1 subgoal

  n : nat
  m : nat
  n0 : nat
  H : (plus n0 m)=(plus m n0)
  n1 : nat
  H0 : (S (plus n1 n0))=(plus n1 (S n0))
  ============================
   (S (S (plus n1 n0)))=(S (plus n1 (S n0)))

Sumacom < Rewrite H0.
1 subgoal

  n : nat
```

```
  m : nat
  n0 : nat
  H : (plus n0 m)=(plus m n0)
  n1 : nat
  H0 : (S (plus n1 n0))=(plus n1 (S n0))
  ============================
   (S (plus n1 (S n0)))=(S (plus n1 (S n0)))

Sumacom < Trivial.
Subtree proved!
```

Vamos a demostrar la asociatividad de la suma. Para ello se hace inducción sobre x. El caso del cero al evaluar ya se resuelve, pues queda `(plus y z)` en los dos lados. Para el sucesor es casi igual, se evalúa la función, y hay que reescribir `(plus n (plus y z))` a `(plus (plus n y) z)`. Hecho esto queda lo mismo en los dos lados de la igualdad.

```
Coq < Lemma Sumaasoc:(x,y,z:nat)(plus x (plus y z))=(plus (plus x y) z).
1 subgoal

  ============================
   (x,y,z:nat)(plus x (plus y z))=(plus (plus x y) z)

Sumaasoc < Intros.
1 subgoal

  x : nat
  y : nat
  z : nat
  ============================
   (plus x (plus y z))=(plus (plus x y) z)

Sumaasoc < Elim x.
2 subgoals

  x : nat
  y : nat
  z : nat
  ============================
   (plus O (plus y z))=(plus (plus O y) z)

subgoal 2 is:
 (n:nat)
  (plus n (plus y z))=(plus (plus n y) z)
  ->(plus (S n) (plus y z))=(plus (plus (S n) y) z)
```

```
Sumaasoc < Simpl.
2 subgoals

  x : nat
  y : nat
  z : nat
  ============================
   (plus y z)=(plus y z)

subgoal 2 is:
 (n:nat)
  (plus n (plus y z))=(plus (plus n y) z)
  ->(plus (S n) (plus y z))=(plus (plus (S n) y) z)

Sumaasoc < Trivial.
1 subgoal

  x : nat
  y : nat
  z : nat
  ============================
   (n:nat)
    (plus n (plus y z))=(plus (plus n y) z)
    ->(plus (S n) (plus y z))=(plus (plus (S n) y) z)

Sumaasoc < Intros.
1 subgoal

  x : nat
  y : nat
  z : nat
  n : nat
  H : (plus n (plus y z))=(plus (plus n y) z)
  ============================
   (plus (S n) (plus y z))=(plus (plus (S n) y) z)

Sumaasoc < Simpl.
1 subgoal

  x : nat
  y : nat
  z : nat
  n : nat
  H : (plus n (plus y z))=(plus (plus n y) z)
  ============================
```

```
   (S (plus n (plus y z)))=(S (plus (plus n y) z))

Sumaasoc < Rewrite H.
1 subgoal

  x : nat
  y : nat
  z : nat
  n : nat
  H : (plus n (plus y z))=(plus (plus n y) z)
  ============================
   (S (plus (plus n y) z))=(S (plus (plus n y) z))

Sumaasoc < Trivial.
Subtree proved!
```

Demostración de la conmutatividad del producto. Esta demostración es muy parecida a la de la conmutatividad de la suma. Para esta demostración, sin embargo, es necesario demostrar la asociatividad del producto, debido a unas sumas mal agrupadas que quedan al final. Se basa, al igual que la de la suma, en hacer inducción sobre las dos variables.

```
Coq < Lemma ProductoCom:(m,n:nat)(mult m n)=(mult n m).
1 subgoal

  ============================
   (m,n:nat)(mult m n)=(mult n m)

ProductoCom < Intros.
1 subgoal

  m : nat
  n : nat
  ============================
   (mult m n)=(mult n m)

ProductoCom < Elim m.
2 subgoals

  m : nat
  n : nat
  ============================
   (mult O n)=(mult n O)

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

```
ProductoCom < Elim n.
3 subgoals

  m : nat
  n : nat
  ============================
   (mult O O)=(mult O O)

subgoal 2 is:
 (n0:nat)(mult O n0)=(mult n0 O)->(mult O (S n0))=(mult (S n0) O)
subgoal 3 is:
 (n0:nat)(mult n0 n)=(mult n n0)->(mult (S n0) n)=(mult n (S n0))

ProductoCom < Trivial.
2 subgoals

  m : nat
  n : nat
  ============================
   (n0:nat)(mult O n0)=(mult n0 O)->(mult O (S n0))=(mult (S n0) O)

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

ProductoCom < Intros.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (mult O n0)=(mult n0 O)
  ============================
   (mult O (S n0))=(mult (S n0) O)

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

ProductoCom < Simpl.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (mult O n0)=(mult n0 O)
```

```
  ============================
   0=(mult n0 0)

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

ProductoCom < Rewrite <- H.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (mult 0 n0)=(mult n0 0)
  ============================
   0=(mult 0 n0)

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

ProductoCom < Simpl.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (mult 0 n0)=(mult n0 0)
  ============================
   0=0

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

ProductoCom < Trivial.
1 subgoal

  m : nat
  n : nat
  ============================
   (n0:nat)(mult n0 n)=(mult n n0)->(mult (S n0) n)=(mult n (S n0))

ProductoCom < Intros.
1 subgoal

  m : nat
  n : nat
```

```
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  ============================
   (mult (S n0) n)=(mult n (S n0))

ProductoCom < Simpl.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  ============================
   (plus n (mult n0 n))=(mult n (S n0))

ProductoCom < Rewrite H.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  ============================
   (plus n (mult n n0))=(mult n (S n0))

ProductoCom < Elim n.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  ============================
   (plus O (mult O n0))=(mult O (S n0))

subgoal 2 is:
 (n1:nat)
  (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ->(plus (S n1) (mult (S n1) n0))=(mult (S n1) (S n0))

ProductoCom < Simpl.
2 subgoals

  m : nat
  n : nat
```

```
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  ============================
   0=0

subgoal 2 is:
 (n1:nat)
  (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ->(plus (S n1) (mult (S n1) n0))=(mult (S n1) (S n0))

ProductoCom < Trivial.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  ============================
   (n1:nat)
    (plus n1 (mult n1 n0))=(mult n1 (S n0))
    ->(plus (S n1) (mult (S n1) n0))=(mult (S n1) (S n0))

ProductoCom < Intros.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  n1 : nat
  H0 : (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ============================
   (plus (S n1) (mult (S n1) n0))=(mult (S n1) (S n0))

ProductoCom < Simpl.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  n1 : nat
  H0 : (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ============================
   (S (plus n1 (plus n0 (mult n1 n0))))=(S (plus n0 (mult n1 (S n0))))
```

```
ProductoCom < Rewrite <- H0.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  n1 : nat
  H0 : (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ============================
   (S (plus n1 (plus n0 (mult n1 n0))))
    =(S (plus n0 (plus n1 (mult n1 n0))))

ProductoCom < Rewrite Sumaasoc.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  n1 : nat
  H0 : (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ============================
   (S (plus (plus n1 n0) (mult n1 n0)))
    =(S (plus n0 (plus n1 (mult n1 n0))))

ProductoCom < Rewrite Sumaasoc.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  n1 : nat
  H0 : (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ============================
   (S (plus (plus n1 n0) (mult n1 n0)))
    =(S (plus (plus n0 n1) (mult n1 n0)))

ProductoCom < Replace (plus n1 n0) with (plus n0 n1).
2 subgoals

  m : nat
  n : nat
```

```
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  n1 : nat
  H0 : (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ============================
   (S (plus (plus n0 n1) (mult n1 n0)))
    =(S (plus (plus n0 n1) (mult n1 n0)))

subgoal 2 is:
 (plus n0 n1)=(plus n1 n0)

ProductoCom < Trivial.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  n1 : nat
  H0 : (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ============================
   (plus n0 n1)=(plus n1 n0)

ProductoCom < Rewrite Sumacom.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (mult n0 n)=(mult n n0)
  n1 : nat
  H0 : (plus n1 (mult n1 n0))=(mult n1 (S n0))
  ============================
   (plus n1 n0)=(plus n1 n0)

ProductoCom < Trivial.
Subtree proved!
```

Vamos a demostrar que el cero es menor o igual que cualquier natural. Para ello se hace inducción sobre n, que para el caso del 0 deja por demostrar que el cero es menor o igual que sí mismo –lo que dice `le_n`– por lo que aplicando `le_n` ya se demuestra.

Para el sucesor tenemos como objetivo la parte derecha de `ls_S`, de modo que aplicándolo nos queda como objetivo su parte izquierda, que ya tenemos en el contexto.

```
Coq < Lemma Ceromenor:(n:nat)(le O n).
```

```
1 subgoal
   ============================
   (n:nat)(le O n)

Ceromenor < Intros.
1 subgoal

  n : nat
  ============================
   (le O n)

Ceromenor < Elim n.
2 subgoals

  n : nat
  ============================
   (le O O)

subgoal 2 is:
 (n0:nat)(le O n0)->(le O (S n0))

Ceromenor < Apply le_n.
1 subgoal

  n : nat
  ============================
   (n0:nat)(le O n0)->(le O (S n0))

Ceromenor < Intros.
1 subgoal

  n : nat
  n0 : nat
  H : (le O n0)
  ============================
   (le O (S n0))

Ceromenor < Apply le_S.
1 subgoal

  n : nat
  n0 : nat
  H : (le O n0)
  ============================
   (le O n0)
```

```
Ceromenor < Assumption.
Subtree proved!
```

Vamos a demostrar que un número natural n siempre es menor o igual que él mismo sumado a cualquier otro natural.

Se hace inducción sobre n. Para el caso del cero sólo hay que evaluar, y queda el teorema anterior (cuya demostración repetimos completa, aunque un `Apply Ceromenor` hubiera bastado).

El otro subojetivo se demuestra haciendo inducción sobre m, y evaluando la función para el cero, y evaluando la función y aplicando `le_S` en el otro. La demostración es larga porque además hay que aplicar varias veces la conmutatividad para que evalúe la función.

```
Coq < Lemma Menorqueplus:(m,n:nat)(le n (plus n m)).
1 subgoal

  ============================
   (m,n:nat)(le n (plus n m))

Menorqueplus < Intros.
1 subgoal

  m : nat
  n : nat
  ============================
   (le n (plus n m))

Menorqueplus < Elim n.
2 subgoals

  m : nat
  n : nat
  ============================
   (le O (plus O m))

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

Menorqueplus < Simpl.
2 subgoals

  m : nat
  n : nat
  ============================
```

```
   (le O m)

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

Menorqueplus < Elim m.
3 subgoals

  m : nat
  n : nat
  ============================
   (le O O)

subgoal 2 is:
 (n0:nat)(le O n0)->(le O (S n0))
subgoal 3 is:
 (n0:nat)(le n0 (plus n0 m))->(le (S n0) (plus (S n0) m))

Menorqueplus < Apply le_n.
2 subgoals

  m : nat
  n : nat
  ============================
   (n0:nat)(le O n0)->(le O (S n0))

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

Menorqueplus < Intros.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (le O n0)
  ============================
   (le O (S n0))

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

Menorqueplus < Apply le_S.
2 subgoals
```

```
  m : nat
  n : nat
  n0 : nat
  H : (le O n0)
  ============================
   (le O n0)

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

Menorqueplus < Assumption.
1 subgoal

  m : nat
  n : nat
  ============================
   (n0:nat)(le n0 (plus n0 m))->(le (S n0) (plus (S n0) m))

Menorqueplus < Intros.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  ============================
   (le (S n0) (plus (S n0) m))

Menorqueplus < Elim m.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  ============================
   (le (S n0) (plus (S n0) O))

subgoal 2 is:
 (n1:nat)(le (S n0) (plus (S n0) n1))->(le (S n0) (plus (S n0) (S n1)))

Menorqueplus < Rewrite Sumacom.
2 subgoals

  m : nat
```

```
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  ============================
   (le (S n0) (plus 0 (S n0)))

subgoal 2 is:
 (n1:nat)(le (S n0) (plus (S n0) n1))->(le (S n0) (plus (S n0) (S n1)))

Menorqueplus < Simpl.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  ============================
   (le (S n0) (S n0))

subgoal 2 is:
 (n1:nat)(le (S n0) (plus (S n0) n1))->(le (S n0) (plus (S n0) (S n1)))

Menorqueplus < Apply le_n.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  ============================
   (n1:nat)
    (le (S n0) (plus (S n0) n1))->(le (S n0) (plus (S n0) (S n1)))

Menorqueplus < Intros.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (le (S n0) (plus (S n0) (S n1)))
```

```
Menorqueplus < Simpl.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (le (S n0) (S (plus n0 (S n1))))

Menorqueplus < Replace (plus n0 (S n1)) with (plus (S n0) n1).
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (le (S n0) (S (plus (S n0) n1)))

subgoal 2 is:
 (plus (S n0) n1)=(plus n0 (S n1))

Menorqueplus < Apply le_S.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (le (S n0) (plus (S n0) n1))

subgoal 2 is:
 (plus (S n0) n1)=(plus n0 (S n1))

Menorqueplus < Assumption.
1 subgoal
```

```
  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (plus (S n0) n1)=(plus n0 (S n1))

Menorqueplus < Replace (plus n0 (S n1)) with (plus (S n1) n0).
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (plus (S n0) n1)=(plus (S n1) n0)

subgoal 2 is:
 (plus (S n1) n0)=(plus n0 (S n1))

Menorqueplus < Simpl.
2 subgoals

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (S (plus n0 n1))=(S (plus n1 n0))

subgoal 2 is:
 (plus (S n1) n0)=(plus n0 (S n1))

Menorqueplus < Rewrite Sumacom.
2 subgoals

  m : nat
  n : nat
  n0 : nat
```

```
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (S (plus n1 n0))=(S (plus n1 n0))

subgoal 2 is:
 (plus (S n1) n0)=(plus n0 (S n1))

Menorqueplus < Trivial.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (plus (S n1) n0)=(plus n0 (S n1))

Menorqueplus < Rewrite Sumacom.
1 subgoal

  m : nat
  n : nat
  n0 : nat
  H : (le n0 (plus n0 m))
  n1 : nat
  H0 : (le (S n0) (plus (S n0) n1))
  ============================
   (plus n0 (S n1))=(plus n0 (S n1))

Menorqueplus < Trivial.
Subtree proved!
```

Reflexiones sobre `cata_nat`. Dada la función `cata_nat` (ya conocida), puede escribirse:

```
Coq < Definition Cata_nat:=[C:Set][f0:C][fc:C->C]
Coq < (nat_rec ([_:nat]C) f0 ([_:nat]fc)).
Cata_nat is defined
```

y sabiendo que la función factorial es:

```
Definition Factorial:nat->nat:=(nat_rec
                        ([_:nat]nat)
                        (S O)
                        ([x:nat][y:nat](producto (S x) y))).
```

Como se ve, `Cata_nat` no permite definir funciones que para pasar al paso $n+1$ necesiten saber quién es n; `Factorial` utiliza $(S \quad x)$ para el cálculo, y por tanto no puede definirse así.

5. Lista de los tipos inductivos de Coq

```
Coq < Print nat.
Inductive nat  : Set :=  O : nat | S : nat->nat

Coq < Check nat.
nat
     : Set

Coq < Print nat_ind.
nat_ind =
[P:(nat->Prop); f:(P O); f0:((n:nat)(P n)->(P (S n)))]
 Fix F
   {F [n:nat] : (P n) :=
      Cases n of
        O => f
      | (S n0) => (f0 n0 (F n0))
      end}
     : (P:(nat->Prop))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

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

Coq < Print nat_rec.
nat_rec =
[P:(nat->Set); f:(P O); f0:((n:nat)(P n)->(P (S n)))]
 Fix F
   {F [n:nat] : (P n) :=
      Cases n of
        O => f
      | (S n0) => (f0 n0 (F n0))
      end}
     : (P:(nat->Set))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

Coq < Check nat_rec.
nat_rec
     : (P:(nat->Set))(P O)->((n:nat)(P n)->(P (S n)))->(n:nat)(P n)

Coq < Print bool.
Inductive bool  : Set :=  true : bool | false : bool

Coq < Check bool.
bool
     : Set
```

```
Coq < Print bool_ind.
bool_ind =
[P:(bool->Prop); f:(P true); f0:(P false); b:bool](if b then f else f0)
     : (P:(bool->Prop))(P true)->(P false)->(b:bool)(P b)

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

Coq < Print bool_rec.
bool_rec =
[P:(bool->Set); f:(P true); f0:(P false); b:bool](if b then f else f0)
     : (P:(bool->Set))(P true)->(P false)->(b:bool)(P b)

Coq < Check bool_rec.
bool_rec
     : (P:(bool->Set))(P true)->(P false)->(b:bool)(P b)

Coq < Print and.
Inductive and [A : Prop; B : Prop]  : Prop :=  conj : A->B->A/\B

Coq < Check and.
and
     : Prop->Prop->Prop

Coq < Print and_ind.
and_ind =
[A,B,P:Prop; f:(A->B->P); a:(A/\B)]
 Cases a of (conj x x0) => (f x x0) end
     : (A,B,P:Prop)(A->B->P)->A/\B->P

Coq < Check and_ind.
and_ind
     : (A,B,P:Prop)(A->B->P)->A/\B->P

Coq < Print and_rec.
and_rec =
[A,B:Prop; C:Set; F:(A->B->C); AB:(A/\B)]
 (F (and_ind A B A [H:A; _:B]H AB) (and_ind A B B [_:A; H0:B]H0 AB))
     : (A,B:Prop; C:Set)(A->B->C)->A/\B->C

Coq < Check and_rec.
and_rec
     : (A,B:Prop; C:Set)(A->B->C)->A/\B->C
```

```
Coq < Print or.
Inductive or [A : Prop; B : Prop]  : Prop :=
      or_introl : A->A\/B | or_intror : B->A\/B

Coq < Check or.
or
     : Prop->Prop->Prop

Coq < Print or_ind.
or_ind =
[A,B,P:Prop; f:(A->P); f0:(B->P); o:(A\/B)]
 Cases o of
   (or_introl x) => (f x)
 | (or_intror x) => (f0 x)
 end
     : (A,B,P:Prop)(A->P)->(B->P)->A\/B->P

Coq < Check or_ind.
or_ind
     : (A,B,P:Prop)(A->P)->(B->P)->A\/B->P

Coq < Print or_rec.
Error: or_rec not a defined object

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

Coq < Check exist.
exist
     : (A:Set; P:(A->Prop); x:A)(P x)->(sig A P)

Coq < Print ex_ind.
ex_ind =
[A:Set; P:(A->Prop); P0:Prop; f:((x:A)(P x)->P0); e:(Ex P)]
 Cases e of (ex_intro x x0) => (f x x0) end
     : (A:Set; P:(A->Prop); P0:Prop)((x:A)(P x)->P0)->(Ex P)->P0

Coq < Check ex_ind.
ex_ind
     : (A:Set; P:(A->Prop); P0:Prop)((x:A)(P x)->P0)->(Ex P)->P0

Coq < Print ex_rec.
Error: ex_rec not a defined object
```

```
Coq < Print eq.
Inductive eq [A : Set; x : A]  : A->Prop :=  refl_equal : x=x

Coq < Check eq.
eq
     : (A:Set)A->A->Prop

Coq < Print eq_ind.
eq_ind =
[A:Set; x:A; P:(A->Prop); f:(P x); y:A; e:(x=y)]
 Cases e of refl_equal => f end
     : (A:Set; x:A; P:(A->Prop))(P x)->(y:A)x=y->(P y)

Coq < Check eq_ind.
eq_ind
     : (A:Set; x:A; P:(A->Prop))(P x)->(y:A)x=y->(P y)

Coq < Print eq_rec.
eq_rec =
[A:Set; x:A; P:(A->Set); f:(P x); y:A; e:(x=y)]
 Cases e of refl_equal => f end
     : (A:Set; x:A; P:(A->Set))(P x)->(y:A)x=y->(P y)

Coq < Check eq_rec.
eq_rec
     : (A:Set; x:A; P:(A->Set))(P x)->(y:A)x=y->(P y)

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

Coq < Check sig.
sig
     : (A:Set)(A->Prop)->Set

Coq < Print sig_ind.
sig_ind =
[A:Set;
 P:(A->Prop);
 P0:((sig A P)->Prop);
 f:((x:A; p:(P x))(P0 (exist A P x p)));
 s:((sig A P))](let (x, x0) = s in (f x x0))
     : (A:Set; P:(A->Prop); P0:((sig A P)->Prop))
        ((x:A; p:(P x))(P0 (exist A P x p)))->(s:((sig A P)))(P0 s)
```

```
Coq < Check sig_ind.
sig_ind
     : (A:Set; P:(A->Prop); P0:((sig A P)->Prop))
        ((x:A; p:(P x))(P0 (exist A P x p)))->(s:((sig A P)))(P0 s)

Coq < Print sig_rec.
sig_rec =
[A:Set;
 P:(A->Prop);
 P0:((sig A P)->Set);
 f:((x:A; p:(P x))(P0 (exist A P x p)));
 s:((sig A P))](let (x, x0) = s in (f x x0))
     : (A:Set; P:(A->Prop); P0:((sig A P)->Set))
        ((x:A; p:(P x))(P0 (exist A P x p)))->(s:((sig A P)))(P0 s)

Coq < Check sig_rec.
sig_rec
     : (A:Set; P:(A->Prop); P0:((sig A P)->Set))
        ((x:A; p:(P x))(P0 (exist A P x p)))->(s:((sig A P)))(P0 s)

Coq < Inductive list[A:Set]:Set:=
Coq <   nil:(list A)
Coq < | cons:A->(list A)->(list A).
list is defined
list_ind is defined
list_rec is defined
list_rect is defined

Coq < Print list.
Inductive list [A : Set]  : Set :=
      nil : (list A) | cons : A->(list A)->(list A)

Coq < Check list.
list
     : Set->Set

Coq < Print list_ind.
list_ind =
[A:Set;
 P:((list A)->Prop);
 f:(P (nil A));
 f0:((a:A; l:(list A))(P l)->(P (cons A a l)))]
 Fix F
   {F [l:(list A)] : (P l) :=
```

```
      Cases l of
        nil => f
      | (cons y l0) => (f0 y l0 (F l0))
      end}
     : (A:Set; P:((list A)->Prop))
        (P (nil A))
        ->((a:A; l:(list A))(P l)->(P (cons A a l)))
        ->(l:(list A))(P l)

Coq < Check list_ind.
list_ind
     : (A:Set; P:((list A)->Prop))
        (P (nil A))
        ->((a:A; l:(list A))(P l)->(P (cons A a l)))
        ->(l:(list A))(P l)

Coq < Print list_rec.
list_rec =
[A:Set;
 P:((list A)->Set);
 f:(P (nil A));
 f0:((a:A; l:(list A))(P l)->(P (cons A a l)))]
 Fix F
   {F [l:(list A)] : (P l) :=
      Cases l of
        nil => f
      | (cons y l0) => (f0 y l0 (F l0))
      end}
     : (A:Set; P:((list A)->Set))
        (P (nil A))
        ->((a:A; l:(list A))(P l)->(P (cons A a l)))
        ->(l:(list A))(P l)

Coq < Check list_rec.
list_rec
     : (A:Set; P:((list A)->Set))
        (P (nil A))
        ->((a:A; l:(list A))(P l)->(P (cons A a l)))
        ->(l:(list A))(P l)

Coq < Print True.
Inductive True  : Prop :=  I : True

Coq < Check True.
True
```

```
     : Prop

Coq < Print True_ind.
True_ind =
[P:Prop; f:P; t:True]Cases t of I => f end
     : (P:Prop)P->True->P

Coq < Check True_ind.
True_ind
     : (P:Prop)P->True->P

Coq < Print True_rec.
True_rec =
[P:Set; f:P; t:True]Cases t of I => f end
     : (P:Set)P->True->P

Coq < Check True_rec.
True_rec
     : (P:Set)P->True->P

Coq < Print False.
Inductive False  : Prop :=

Coq < Check False.
False
     : Prop

Coq < Print False_ind.
False_ind = [P:Prop; f:False]<P>Cases f of end
     : (P:Prop)False->P

Coq < Check False_ind.
False_ind
     : (P:Prop)False->P

Coq < Print False_rec.
False_rec =
[P:Set; H:False]
 <[b:bool](Q P b)>
   Cases (False_ind true=false H) of refl_equal => tt end
     : (P:Set)False->P

Coq < Check False_rec.
False_rec
     : (P:Set)False->P
```

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

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

Coq < Print le_ind.
le_ind =
[n:nat; P:(nat->Prop); f:(P n); f0:((m:nat)(le n m)->(P m)->(P (S m)))]
 Fix F
   {F [n0:nat; l:(le n n0)] : (P n0) :=
      Cases l of
        le_n => f
      | (le_S m l0) => (f0 m l0 (F m l0))
      end}
     : (n:nat; P:(nat->Prop))
        (P n)
        ->((m:nat)(le n m)->(P m)->(P (S m)))
        ->(n0:nat)(le n n0)->(P n0)

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)

Coq < Print le_rec.
Error: le_rec not a defined object

Coq < Print lt.
lt = [n,m:nat](le (S n) m)
     : nat->nat->Prop


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

Coq < Print lt_ind.
Error: lt_ind not a defined object
```

```
Coq < Print lt_rec.
Error: lt_rec not a defined object

Coq < Print prod.
Inductive prod [A : Set; B : Set]  : Set :=  pair : A->B->A*B

Coq < Check prod.
prod
     : Set->Set->Set

Coq < Print prod_ind.
prod_ind =
[A,B:Set; P:(A*B->Prop); f:((a:A; b:B)(P (a,b))); p:(A*B)]
 (let (x, x0) = p in (f x x0))
     : (A,B:Set; P:(A*B->Prop))((a:A; b:B)(P (a,b)))->(p:(A*B))(P p)

Coq < Check prod_ind.
prod_ind
     : (A,B:Set; P:(A*B->Prop))((a:A; b:B)(P (a,b)))->(p:(A*B))(P p)

Coq < Print prod_rec.
prod_rec =
[A,B:Set; P:(A*B->Set); f:((a:A; b:B)(P (a,b))); p:(A*B)]
 (let (x, x0) = p in (f x x0))
     : (A,B:Set; P:(A*B->Set))((a:A; b:B)(P (a,b)))->(p:(A*B))(P p)

Coq < Check prod_rec.
prod_rec
     : (A,B:Set; P:(A*B->Set))((a:A; b:B)(P (a,b)))->(p:(A*B))(P p)
```

6. Anexo: Algoritmo de Euclides

6.1. División euclídea: el algoritmo de la división entera

```
Coq < Lemma div_euclid : (a,b:nat) {q:(nat*nat) |
Coq <    (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 q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Require Export Omega.

div_euclid < Require Export Compare.

div_euclid < Intro.
1 subgoal

  a : nat
  ============================
   (b:nat)
    {q:(nat*nat) |
       (a=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Elim a.
2 subgoals

  a : nat
  ============================
   (b:nat)
    {q:(nat*nat) |
       ((0)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

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

```
div_euclid < Intros.
2 subgoals

  a : nat
  b : nat
  ============================
   {q:(nat*nat) |
      ((0)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

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

div_euclid < Exists (0,0).
2 subgoals

  a : nat
  b : nat
  ============================
   (0)=(plus (Snd ((0),(0))) (mult (Fst ((0),(0))) (S b)))
   /\(lt (Snd ((0),(0))) (S b))

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

div_euclid < Simpl.
2 subgoals

  a : nat
  b : nat
  ============================
   (0)=(0)/\(lt (0) (S b))

subgoal 2 is:
```

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

div_euclid < Split.
3 subgoals

  a : nat
  b : nat
  ============================
   (0)=(0)

subgoal 2 is:
 (lt (0) (S b))
subgoal 3 is:
 (n:nat)
  ((b:nat)
    {q:(nat*nat) |
       (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))})
  ->(b:nat)
     {q:(nat*nat) |
        ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Trivial.
2 subgoals

  a : nat
  b : nat
  ============================
   (lt (0) (S b))

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

div_euclid < Apply lt_O_Sn.
```

```
1 subgoal

  a : nat
  ============================
   (n:nat)
    ((b:nat)
      {q:(nat*nat) |
         (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))})
    ->(b:nat)
       {q:(nat*nat) |
          ((S n)=(plus (Snd q) (mult (Fst q) (S b)))
           /\(lt (Snd q) (S b)))}

div_euclid < Intros.
1 subgoal

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  ============================
   {q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Elim (H b).
1 subgoal

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  ============================
   (x:(nat*nat))
    n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
    ->{q:(nat*nat) |
         ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Intros.
1 subgoal

  a : nat
```

```
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  ============================
   {q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Elim p.
1 subgoal

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  ============================
   n=(plus (Snd x) (mult (Fst x) (S b)))
   ->(lt (Snd x) (S b))
   ->{q:(nat*nat) |
        ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Intros.
1 subgoal

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  ============================
   {q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
```

```
div_euclid < Elim (lt_eq_lt_dec (S (Snd x)) (S b)).
2 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  ============================
   {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
   ->{q:(nat*nat) |
        ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

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

div_euclid < Intros.
2 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  ============================
   {q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

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

```
div_euclid < Elim a0.
3 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  ============================
   (lt (S (Snd x)) (S b))
   ->{q:(nat*nat) |
        ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

subgoal 2 is:
 (S (Snd x))=(S b)
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd x)))
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Intros.
3 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  a1 : (lt (S (Snd x)) (S b))
  ============================
```

```
   {q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

subgoal 2 is:
 (S (Snd x))=(S b)
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd x)))
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Exists ((Fst x), (S (Snd x))).
3 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  a1 : (lt (S (Snd x)) (S b))
  ============================
   (S n)
    =(plus (Snd ((Fst x),(S (Snd x))))
       (mult (Fst ((Fst x),(S (Snd x)))) (S b)))
   /\(lt (Snd ((Fst x),(S (Snd x)))) (S b))

subgoal 2 is:
 (S (Snd x))=(S b)
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd x)))
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Simpl.
3 subgoals
```

```
  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  a1 : (lt (S (Snd x)) (S b))
  ============================
   (S n)=(S (plus (Snd x) (mult (Fst x) (S b))))
   /\(lt (S (Snd x)) (S b))

subgoal 2 is:
 (S (Snd x))=(S b)
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd x)))
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Split.
4 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  a1 : (lt (S (Snd x)) (S b))
  ============================
   (S n)=(S (plus (Snd x) (mult (Fst x) (S b))))

subgoal 2 is:
 (lt (S (Snd x)) (S b))
```

```
subgoal 3 is:
 (S (Snd x))=(S b)
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 4 is:
 (lt (S b) (S (Snd x)))
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Auto.
3 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  a1 : (lt (S (Snd x)) (S b))
  ============================
   (lt (S (Snd x)) (S b))

subgoal 2 is:
 (S (Snd x))=(S b)
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
subgoal 3 is:
 (lt (S b) (S (Snd x)))
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Assumption.
2 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
```

```
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  ============================
   (S (Snd x))=(S b)
   ->{q:(nat*nat) |
        ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

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

div_euclid < Intros.
2 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  b0 : (S (Snd x))=(S b)
  ============================
   {q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

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

div_euclid < Exists ((S (Fst x)), O).
2 subgoals

  a : nat
  n : nat
  H : (b:nat)
```

```
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  b0 : (S (Snd x))=(S b)
  ============================
   (S n)
    =(plus (Snd ((S (Fst x)),(0))) (mult (Fst ((S (Fst x)),(0))) (S b)))
   /\(lt (Snd ((S (Fst x)),(0))) (S b))

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

div_euclid < Simpl.
2 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  b0 : (S (Snd x))=(S b)
  ============================
   (S n)=(S (plus b (mult (Fst x) (S b))))/\(lt (0) (S b))

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

div_euclid < Split.
3 subgoals
```

```
  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  b0 : (S (Snd x))=(S b)
  ============================
   (S n)=(S (plus b (mult (Fst x) (S b))))

subgoal 2 is:
 (lt (0) (S b))
subgoal 3 is:
 (lt (S b) (S (Snd x)))
 ->{q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Omega.
2 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  a0 : {(lt (S (Snd x)) (S b))}+{(S (Snd x))=(S b)}
  b0 : (S (Snd x))=(S b)
  ============================
   (lt (0) (S b))

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

```
div_euclid < Apply lt_O_Sn.
1 subgoal

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  ============================
   (lt (S b) (S (Snd x)))
   ->{q:(nat*nat) |
        ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Intros.
1 subgoal

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  b0 : (lt (S b) (S (Snd x)))
  ============================
   {q:(nat*nat) |
      ((S n)=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}

div_euclid < Absurd (lt b (Snd x)).
2 subgoals

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
```

```
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  b0 : (lt (S b) (S (Snd x)))
  ============================
   ~(lt b (Snd x))

subgoal 2 is:
 (lt b (Snd x))

div_euclid < Omega.
1 subgoal

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  b0 : (lt (S b) (S (Snd x)))
  ============================
   (lt b (Snd x))

div_euclid < Apply lt_S_n.
1 subgoal

  a : nat
  n : nat
  H : (b:nat)
       {q:(nat*nat) |
          (n=(plus (Snd q) (mult (Fst q) (S b)))/\(lt (Snd q) (S b)))}
  b : nat
  x : nat*nat
  p : n=(plus (Snd x) (mult (Fst x) (S b)))/\(lt (Snd x) (S b))
  H0 : n=(plus (Snd x) (mult (Fst x) (S b)))
  H1 : (lt (Snd x) (S b))
  b0 : (lt (S b) (S (Snd x)))
  ============================
   (lt (S b) (S (Snd x)))
```

```
div_euclid < Assumption.
Subtree proved!

div_euclid < Qed.
Intro.
Elim a.
Intros.
Split with (0,0).
Simpl.
Split.
Trivial.

Apply lt_0_Sn.

Intros.
Elim (H b).
Intros.
Elim p.
Intros.
Elim (lt_eq_lt_dec (S (Snd x)) (S b)).
Intros.
Elim a0.
Intros.
Split with ((Fst x),(S (Snd x))).
Simpl.
Split.
Auto.

Assumption.

Intros.
Split with ((S (Fst x)),0).
Simpl.
Split.
Omega.

Apply lt_0_Sn.

Intros.
Absurd (lt b (Snd x)).
Omega.

Apply lt_S_n.
Assumption.
```

```
div_euclid is defined

Coq < Recursive Extraction div_euclid.
type nat =
    O
  | S of nat

let nat_rec f0 f =
  let rec f1 = function
    O -> f0
  | S n0 -> f n0 (f1 n0)
  in f1

type ('A, 'B) prod =
    pair of 'A * 'B

let sig_rec f s =
  f s prop

let Logic.and_rec f _ =
  f prop prop

type 'A sumor =
    inleft of 'A
  | inright

let sumor_rec f0 f = function
  inleft a -> f0 a
| inright -> f prop

type sumbool =
    left
  | right

let sumbool_rec f0 f = function
  left -> f0 prop
| right -> f prop

let fst = function
  pair (x, y) -> x

let snd = function
  pair (x, y) -> y

let False_rec _ =
```

```
  failwith "False_rec"

let lt_eq_lt_dec n =
  nat_rec (fun m -> nat_rec (inleft right) (fun n0 h -> inleft left) m)
    (fun n0 h m ->
    nat_rec inright (fun q h' ->
      sumor_rec (fun a ->
        sumbool_rec (fun _ -> inleft left) (fun _ -> inleft right) a)
        (fun _ -> inright) (h q)) m) n

let div_euclid a =
  nat_rec (fun b -> pair (O, O)) (fun n h b ->
    sig_rec (fun x _ ->
      Logic.and_rec (fun _ _ ->
        sumor_rec (fun a0 ->
          sumbool_rec (fun _ -> pair ((fst x), (S (snd x)))) (fun _ ->
            pair ((S (fst x)), O)) a0) (fun _ -> False_rec prop)
          (lt_eq_lt_dec (S (snd x)) (S b))) prop) (h b)) a
```

Índice

Referencias

[1] Freire Nistal, José Luis. *Apuntes de Matemáticas Discretas II.* Departamento de Computación. Facultad de Informática, Universidad de La Coruña, 2001.

[2] Giménez, Eduardo. *A Tutorial on Recursive Types in Coq.* INRIA, Mayo 1998.

[3] Barras, Samuel y Boutin, Samuel y Cornes, Cristina y al. *The Coq Proof Assistant. A Tutorial (Version 6.2).* INRIA, Mayo 1998.

[4] Huet, Gérard y Kahn, Gilles y Paulin-Mohring, Christine. *The Coq Proof Assistant. A Tutorial (Version 6.3.1).* INRIA, Diciembre 1999.

Índice alfabético

Matemática Discreta II

Curso: 2001/2002
Cuatrimestre: Primero
Alumna: Laura M. Castro Souto
Profesor: José Luis Doncel Juárez

1. Combinatoria

1.1. Funciones Generatrices o Generadoras

Ya en la primera parte de esta asignatura (**Matemática Discreta**) vimos algunos conceptos de combinatoria: *variaciones*, *permutaciones* y *combinaciones* (todas ellas con y sin repetición).

Ejemplos típicos de aplicación de estos conceptos, por ejemplo, eran: "*Calcular el número de soluciones enteras de la ecuación* $c_1 + c_2 + c_3 + c_4 = 25$ *con* $0 \leq c_i$, $1 \leq i \leq 4$".

Entonces hacíamos:

$$CR(4, 25) = C(4 + 25 - 1, 25) = C(28, 25) = \binom{28}{25}$$

En general,

$$CR(n, r) = C(n + r - 1, r) = \binom{n + r - 1}{r}$$

Si introducíamos más restricciones el ejemplo podía complicarse: "*Calcular el número de soluciones enteras de la ecuación* $c_1 + c_2 + c_3 + c_4 = 25$ *con* $0 \leq c_i \leq 10$, $1 \leq i \leq 4$".

La ventaja de las **funciones generatrices** es que nos permitirán estudiar estos casos conocidos y otros con muchas más restricciones, del estilo de: "*Calcular el número de soluciones enteras de la ecuación* $c_1 + c_2 + c_3 + c_4 = 25$ *con* $0 \leq c_i$, c_2 *par y* c_3 *múltiplo de 3*".

Sea el siguiente caso: "*Hallar las formas de distribuir 12 bolas idénticas en 3 cajas distintas A, B y C, de forma que la caja A contenga al menos 4 bolas, las cajas B y C al menos 2 y la caja C no más de 5*".

Llamemos

$$\begin{array}{rcl} c_1 & = & \text{número de bolas en la caja A} \\ c_2 & = & \text{número de bolas en la caja B} \\ c_3 & = & \text{número de bolas en la caja C} \end{array}$$

entonces tenemos la ecuación

$$c_1 + c_2 + c_3 = 12 \qquad 4 \leq c_1 \quad 2 \leq c_2 \quad 2 \leq c_3 \leq 5$$

Analizando el problema, vemos que las posibles soluciones son:

A	B	C
4	3	5
4	4	4
4	5	3
4	6	2
5	2	5
5	3	4
5	4	3
5	5	2
6	2	4
6	3	3
6	4	2
7	2	3
7	3	2
8	2	2

Que coinciden con los términos cuyo producto es x^{12} en la expresión

$$(x^4 + x^5 + x^6 + x^7 + x^8) \cdot (x^2 + x^3 + x^4 + x^5 + x^6) \cdot (x^2 + x^3 + x^4 + x^5)$$

que son

$1^{er} polinomio$	$2^{o} polinomio$	$3^{er} polinomio$
x^4	x^3	x^5
x^4	x^4	x^4
$\dots$		

Es decir, cada aparición de x^{12} en el producto anterior corresponde a una de las ternas que son solución de la ecuación ($x^i x^j x^k$, con $i + j + k = 12$). El número de posibles distribuciones buscado será, pues, igual al coeficiente del término x^{12} en el producto de polinomios anterior.

Lo interesante es, pues, conocer de dónde obtenemos los mencionados polinomios. Es sencillo: los rangos de sus exponentes se obtienen del análisis de las restricciones del problema. Este producto será, cuando la definamos, la **función generatriz**. La ventaja es que este único polinomio nos servirá para hallar la solución tanto para 12 bolas como para cualquier otro número de ellas, por lo que la potencia de este método se ve realmente en circunstancias de gran número de restricciones de mayor complejidad.

Veamos un par de ejemplos más:

"*Si existen un número ilimitado (o al menos 24 de cada) de bolas de color rojo, verde, blanco y negro, ¿de cuántas formas se pueden seleccionar 24 de estas bolas de tal manera que haya un número par de bolas blancas y al menos 6 bolas negras?*".

Tendremos

bolas verdes o rojas	$(1 + x + x^2 + x^3 + \ldots + x^{24})$
bolas blancas	$(x^0 + x^2 + x^4 + \ldots + x^{24})$
bolas negras	$(x^6 + x^7 + x^8 + \ldots + x^{24})$

de modo que en este caso multiplicaríamos

$$(1 + x + x^2 + x^3 + \ldots + x^{24})^2 \cdot (1 + x^2 + x^4 + \ldots + x^{24}) \cdot (x^6 + x^7 + x^8 + \ldots + x^{24})$$

y el coeficiente buscado sería el de x^{24}.

"*¿De cuántas formas podemos distribuir 25 céntimos de euro (en monedas de 1c €) entre 4 personas?*".

La ecuación es la ya conocida

$$c_1 + c_2 + c_3 + c_4 = 25 \qquad 0 \leq c_i \quad 1 \leq i \leq 4$$

el polinomio que se obtiene es

$$(1 + x + x^2 + x^3 + \ldots + x^{25})^4$$

y el coeficiente buscado, x^{25}.

En realidad, daría igual utilizar la **serie de potencias** $(1 + x + x^2 + x^3 + \ldots + x^{25} + x^{26} + x^{27} + \ldots)^4$ pues el coeficiente de x^{25} no va a variar y su cálculo se hará más sencillo (aunque a priori pueda parecer que complicamos las cosas).

Lo que sí debemos tener en cuenta es que no estamos hablando de una suma infinita de *valores reales*, sino que los términos x_i son indicativos de *posiciones*, de un orden. Es por ello que no son *series* en el sentido en que las entendíamos en Cálculo y no tendría sentido hablar de su convergencia ni cosas por el estilo.

1.1.1. Serie Formal de Potencias y Función Generatriz

Sea $a_0, a_1, a_2 \ldots$ una sucesión de números reales, en forma abreviada $\{a_n\}_{n\in\mathbb{N}}$.

- La función $f(x) = a_0 + a_1 x + a_2 x^2 + \ldots = \sum_{i=0}^{\infty} a_i x^i$ se denomina **función generatriz de la sucesión** $\{a_n\}_{n\in\mathbb{N}}$.
- La expresión $\sum_{i=0}^{\infty} a_i x^i$ se denomina **serie (formal) de potencias**[1].

[1] La matización *formal* se hace para distinguir esta definición de *serie* de la manejada en Cálculo.

Veamos ahora algunos ejemplos de funciones generatrices:

√ Como sabemos,

$$(1+x)^n = \binom{n}{0} + \binom{n}{1}x + \binom{n}{2}x^2 + \ldots + \binom{n}{n}x^n$$

de modo que la función $f(x) = (1+x)^n$ es la generatriz de la sucesión

$$\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \ldots, \binom{n}{n}, 0, 0, \ldots$$

√ Tenemos que $(1-x^{n+1}) = (1-x)(1+x+x^2+\ldots+x^n)$. Si despejamos,

$$\frac{1-x^{n+1}}{1-x} = 1 + x + x^2 + \ldots + x^n$$

así que

$$f(x) = \frac{1-x^{n+1}}{1-x}$$

es la función generatriz de la sucesión $\underbrace{1, 1, 1, \ldots, 1, 1}_{n+1}, 0, 0, \ldots$.

√ Si hacemos tender n a infinito en el ejemplo anterior,

$$1 = (1-x)(1+x+x^2+\ldots+x^n+\ldots)$$

y despejamos

$$\frac{1}{1-x} = 1 + x + x^2 + \ldots + x^n + \ldots$$

entonces

$$f(x) = \frac{1}{1-x}$$

es la función generatriz de la sucesión $1, 1, 1, \ldots$, esto es, $\{1\}_{i\in\mathbb{N}}$, la *sucesión constante uno*.

√ Asimismo, teniendo

$$\frac{1}{1-x} = 1 + x + x^2 + \ldots$$

si derivamos

$$\frac{1}{(1-x)^2} = 1 + 2x + 3x^2 + \ldots$$

entonces

$$f(x) = \frac{1}{(1-x)^2}$$

es la función generatriz de la sucesión $1, 2, 3, 4, \ldots$, es decir, $\{(i+1)\}_{i\in\mathbb{N}}$.

√ Y del mismo modo

$$\frac{x}{(1-x)^2}$$

es la función generatriz de la sucesión $0, 1, 2, 3, \ldots$, esto es, $\{i\}_{i\in\mathbb{N}}$. Multiplicar por x supone, por tanto, desplazar una serie formal hacia la izquierda, pues

$$\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + \ldots$$

√ Además, a partir de

$$\frac{x}{(1-x)^2} = x + 2x^2 + 3x^3 + 4x^4 + \ldots = \sum_{i=0}^{\infty} ix^i$$

derivando obtenemos

$$\frac{x+1}{(1-x)^3} = 1 + 2^2x + 3^2x^2 + 4^2x^3 + \ldots = \sum_{i=0}^{\infty}(i+1)^2x^i$$

es decir,

$$\frac{x+1}{(1-x)^3}$$

es la función generatriz de $\{(i+1)^2\}_{i\in\mathbb{N}}$.

√ Del mismo modo, si desplazamos la expresión anterior, obtenemos

$$\frac{x(x+1)}{(1-x)^3}$$

que es la función generatriz de $\{0, 1^2, 2^2, 3^2, \ldots\} = \{i^2\}_{i\in\mathbb{N}}$.

√ También son posibles cualquier otro tipo de combinaciones, como por ejemplo

$$\frac{1}{1-x} - x^2$$

que es la función generatriz de $\{1, 1, 0, 1, 1, 1, 1, \ldots\}$, obtenida a partir de

$$\frac{1}{1-x}$$

que es la función generatriz de la sucesión constante uno, quitándole el término correspondiente a la posición 2.

1.1.2. Generalización del concepto de número combinatorio

Dados $n, r \in \mathbb{Z}^+$, se define un **número combinatorio**

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{n(n-1)(n-2)\dots(n-r+1)}{r!}$$

De modo que, siendo n un número real cualquiera, definimos

$$\binom{n}{r} = \frac{n(n-1)(n-2)\dots(n-r+1)}{r!}$$

y, en particular, si $n \in \mathbb{Z}^+$,

$$\begin{aligned}
\binom{-n}{r} &= \frac{(-n)(-n-1)(-n-2)\dots(-n-r+1)}{r!} \\
&= \frac{(-n)(-(n+1))(-(n+2))\dots(-(n+r-1))}{r!} \\
&= (-1)^r \frac{(n+r-1)(n+r-2)\dots(n+2)(n+1)n}{r!} \\
&= (-1)^r \binom{n+r-1}{r}
\end{aligned}$$

Además, para todo número real n se define $\binom{n}{0} = 1$.

A raíz de esta ampliación del concepto de número combinatorio, podemos obtener algunos resultados más:

$\surd$ Por el desarrollo de McLaurin, se tiene que

$$(1+x)^{-n} = \sum_{i=0}^{\infty} (-1)^i \binom{n+i-1}{i} = \sum_{i=0}^{\infty} \binom{-n}{i} x^i$$

siendo $n \in \mathbb{Z}^+$. En consecuencia,

$$f(x) = (1+x)^{-n} = \frac{1}{(1+x)^n} \quad n \in \mathbb{Z}^+$$

es la función generatriz de la sucesión

$$\left\{ (-1)^i \binom{n+i-1}{i} \right\}_{i \in \mathbb{N}} = \left\{ \binom{-n}{i} \right\}_{i \in \mathbb{N}}$$

√ Por el desarrollo de McLaurin, para un número real cualquiera n

$$(1+x)^n = \sum_{i=0}^{\infty} \binom{n}{i} x^i$$

de modo que, en consecuencia, $f(x) = (1+x)^n$ con $n \in \mathbb{R}$ es la función generatriz de la sucesión

$$\left\{ \binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \dots \right\}$$

abreviadamente

$$\left\{ \binom{n}{i} \right\}_{i \in \mathbb{N}}$$

La tabla 1 resume las funciones generatrices más importantes.

Cuadro 1: Tabla de funciones generatrices.

Sucesión	**Expresión**	**Función generatriz**
$\left\{ \binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \dots, \binom{n}{n}, 0, \dots \right\}$	$\sum_{i=0}^{n} \binom{n}{i} x^i$	$(1+n)^n \quad n \in \mathbb{N}$
$\{1, 1, 1, \dots, 1, \dots\}$	$\sum_{i=0}^{\infty} x^i$	$\frac{1}{1-x}$
$\{1, a, a^2, \dots, a^i, \dots\}$	$\sum_{i=0}^{\infty} a^i x^i$	$\frac{1}{1-ax}$
$\{(-1)^i\}$	$\sum_{i=0}^{\infty} (-1)^i x^i$	$\frac{1}{1+x}$
$\{(-a)^i\}$	$\sum_{i=0}^{\infty} (-a)^i x^i$	$\frac{1}{1+ax}$
$\left\{ \binom{n}{i} \right\}$	$\sum_{i=0}^{\infty} \binom{n}{i} x^i$	$(1+x)^r \quad r \in \mathbb{R}$
$\left\{ \binom{n+i-1}{i} \right\}$	$\sum_{i=0}^{\infty} \binom{n+i-1}{i} x^i$	$\frac{1}{(1-x)^n} \quad n \in \mathbb{N}$

. . . . . Continúa en la página siguiente . . . . . .

Cuadro 1: Tabla de funciones generatrices.

. . . . . Continuación . . . . .

$\left\{\binom{n+i-1}{i}a^i\right\}$	$\sum_{i=0}^{\infty}\binom{n+i-1}{i}a^ix^i$	$\frac{1}{(1-ax)^n}\quad n\in\mathbb{N}$
$\left\{\binom{n+i-1}{i}(-a)^i\right\}$	$\sum_{i=0}^{\infty}\binom{n+i-1}{i}(-a)^ix^i$	$\frac{1}{(1+ax)^n}\quad n\in\mathbb{N}$
$\{(i+1)\}=\{1,2,3,\dots\}$	$\sum_{i=0}^{\infty}(i+1)x^i$	$\frac{1}{(1-x)^2}$
$\{i\}=\{0,1,2,\dots\}$	$\sum_{i=0}^{\infty}ix^i$	$\frac{x}{(1-x)^2}$
$\{(i+2)(i+1)\}$	$\sum_{i=0}^{\infty}(i+2)(i+1)x^i$	$\frac{2}{(1-x)^3}$
$\{(i+1)i\}$	$\sum_{i=0}^{\infty}(i+1)ix^i$	$\frac{2x}{(1-x)^3}$
$\{i^2\}$	$\sum_{i=0}^{\infty}i^2x^i$	$\frac{x(1+x)}{(1-x)^3}$
$\{(i+3)(i+2)(i+1)\}$	$\sum_{i=0}^{\infty}(i+3)(i+2)(i+1)x^i$	$\frac{6}{(1-x)^4}$
$\{i^3\}$	$\sum_{i=0}^{\infty}i^3x^i$	$\frac{x(1+4x+x^2)}{(1-x)^4}$
$\{ia^i\}$	$\sum_{i=0}^{\infty}ia^ix^i$	$\frac{ax}{(1-ax)^2}$
$\{i^2a^i\}$	$\sum_{i=0}^{\infty}i^2a^ix^i$	$\frac{ax(1+ax)}{(1-ax)^3}$
$\{i^3a^i\}$	$\sum_{i=0}^{\infty}i^3a^ix^i$	$\frac{ax(1+4ax+a^2x^2)}{(1-ax)^4}$

. . . . . Continúa en la página siguiente . . . . .

Cuadro 1: Tabla de funciones generatrices.

. . . . . Continuación . . . . .

$\left\{\frac{i}{i!}\right\}$	$\sum_{i=0}^{\infty}\frac{x^i}{i!}$	e^x
$\left\{0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots\right\}$	$\sum_{i=1}^{\infty}\frac{1}{i}x^i$	$\ln\left(\frac{1}{1-x}\right)$
$\left\{0, 1, -\frac{1}{2}, \frac{1}{3}, -\frac{1}{4}, \dots\right\}$	$\sum_{i=1}^{\infty}\frac{(-1)^{i+1}}{i}x^i$	$\ln(1+x)$

1.1.3. Propiedades de las operaciones entre funciones generatrices

Sean

$$f(x) = \sum_{i=0}^{\infty} a_i x^i \quad \text{y} \quad g(x) = \sum_{i=0}^{\infty} b_i x^i$$

dos series de polinomios correspondientes a las sucesiones $\{a_i\}_{i\in\mathbb{N}}$ y $\{b_i\}_{i\in\mathbb{N}}$.

Se cumple que:

1. $f(x) = g(x) \Leftrightarrow a_i = b_i \quad \forall i \in \mathbb{N}$
2. Dado un número real c,

$$c \cdot f(x) = \sum_{i=0}^{\infty}(c \cdot a_i)x^i$$

corresponde a la sucesión $\{c \cdot a_i\}_{i\in\mathbb{N}}$.

3. La suma

$$f(x) + g(x) = \sum_{i=0}^{\infty}(a_i + b_i)x^i$$

corresponde a la sucesión $\{(a_i + b_i)\}_{i\in\mathbb{N}}$.

4. El producto

$$f(x) \cdot g(x) = \sum_{i=0}^{\infty} c_i x^i$$

donde

$$c_i = \sum_{\substack{j,k=1 \\ j+k=i}}^{\infty} a_j b_k$$

cumple que cada término $c_i x^i$ es la suma de todos aquellos productos de un término $a_j x^j$ por un término $b_k x^k$ tales que $j + k = i$, o bien, escrito de otro modo,

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

La sucesión $\{c_i\}_{i\in\mathbb{N}}$ corresponde al producto $f(x) \cdot g(x)$ y se denomina **convolución** de las sucesiones $\{a_i\}$ y $\{b_i\}$.

Veamos algunos ejemplos:

- Sea

$$f(x) = \frac{x}{(1-x)^2}$$

la función generatriz de $\{a_i\} = \{i\}$ y

$$g(x) = \frac{x(x+1)}{(1-x)^3}$$

la función generatriz de $\{b_i\} = \{i^2\}$. Entonces,

$$f(x) \cdot g(x) = \frac{x^2(x+1)}{(1-x)^5}$$

es la función generatriz de la sucesión $\{c_k\}$ con

$$c_k = \sum_{i=0}^{\infty} i(k-i)^2$$

- Sea

$$f(x) = \frac{1}{1-x}$$

la función generatriz de $\{i\}$ y

$$g(x) = \frac{1}{1+x}$$

la función generatriz de $\{1, -1, 1, -1, \dots\}$. Entonces,

$$f(x) \cdot g(x) = \frac{1}{1-x^2}$$

es la función generatriz de la sucesión $\{1, 0, 1, 0, \dots\}$.

- Sea $f(x) = 1+x+x^2+x^3$ la función generatriz de la sucesión $\{1, 1, 1, 1, 0, 0, \dots\}$ y
$$g(x) = \frac{1}{1-3x}$$
la función generatriz de la sucesión $\{3^i\}$. Entonces,
$$f(x) \cdot g(x) = \sum_{i=0}^{\infty} c_i x^i$$
con $\{c_i\}$ la convolución de $\{1, 1, 1, 1, 0, 0, \dots\}$ y $\{3^i\}$, que resulta ser
$$\begin{array}{rcll} c_0 & = & 1 & \\ c_1 & = & 1+3=4 & \\ c_2 & = & 1+3+9=13 & \\ c_i & = & 3^{i-3}+3^{i-2}+3^{i-1}+3^i & i \geq 3 \end{array}$$

Resultado

Sean $c(x)$ y $a(x)$ dos polinomios en $\mathbb{R}[x]$ tales que
$$a(x) = a_n(x-\alpha_1)^{r_1} \cdot (x-\alpha_2)^{r_2} \dots (x-\alpha_k)^{r_k}$$
y el grado de $c(x)$ menor que el grado de $a(x)$. Entonces el cociente $\dfrac{c(x)}{a(x)}$ se puede expresar como suma de fracciones elementales en la forma siguiente:
$$\begin{aligned} \frac{c(x)}{a(x)} = & \frac{a_{11}}{(x-\alpha_1)} + \frac{a_{12}}{(x-\alpha_1)^2} + \dots + \frac{a_{1r_1}}{(x-\alpha_1)^{r_1}} + \\ & + \frac{a_{21}}{(x-\alpha_2)} + \frac{a_{22}}{(x-\alpha)^2} + \dots + \frac{a_{2r_2}}{(x-\alpha_2)^{r_2}} + \\ & \dots \\ & + \frac{a_{k1}}{(x-\alpha_k)} + \frac{a_{k2}}{(x-\alpha_k)^2} + \dots + \frac{a_{kr_k}}{(x-\alpha_k)^{r_k}} \end{aligned}$$
con $a_{ij} \in \mathbb{R}$.

1.1.4. Particiones de enteros

Los problemas de **partición de enteros** son problemas en los que se busca hallar un número entero como suma de enteros positivos sin que el orden influya.

Sea $n \in \mathbb{Z}^+$, se define $p(n)$ como el número de formas de expresar n como suma de enteros positivos sin importar el orden.

Por ejemplo,

$$\begin{array}{ccc} p(1)=1 & p(2)=2 & p(3)=3 \\ & & \\ 1=1 & 2=2 & 3=3 \\ & 2=1+1 & 3=2+1 \\ & & 3=1+1+1 \end{array}$$

Si pensamos que

la serie $1+x+x^2+x^3+\dots$ cuenta el número de *unos* (f.g. $\dfrac{1}{1-x}$)

la serie $1+x^2+x^4+x^6+\dots$ cuenta el número de *doses* (f.g. $\dfrac{1}{1-x^2}$)

la serie $1+x^3+x^6+x^9+\dots$ cuenta el número de *treses* (f.g. $\dfrac{1}{1-x^3}$)

$\dots$

la serie $1+x^{10}+x^{20}+x^{30}+\dots$ cuenta el número de *dieces* (f.g. $\dfrac{1}{1-x^{10}}$)

En general,

la serie $1+x^n+x^{2n}+x^{3n}+\dots$ cuenta el número de *enes* (f.g. $\dfrac{1}{1-x^n}$)

Podemos ver que la función generatriz de la sucesión $p(n)$,

$$\{p(0),p(1),\dots\}=\{p(i)\}_{i\in\mathbb{N}}$$

es

$$\prod_{i=0}^{\infty}\frac{1}{1-x^i}$$

definiéndose $p(0)=1$.

Llamaremos además $p_d(n)$ al número de formas de expresar n como suma de enteros positivos distintos. Así, para $k\in\mathbb{Z}^+$,

$$f(x)=(1+x)(1+x^2)(1+x^3)\dots=\prod_{i=1}^{\infty}(1+x^i)$$

y $p_d(n)$ sería el coeficiente de x^n en

$$\prod_{i=1}^{(n)\ \infty}(1+x^i)$$

Denotaremos por $p_o(n)$ el número de formas de expresar n como suma de enteros positivos impares. Entonces

$$f(x) = \prod_{i=1}^{\infty} \frac{1}{1+x^{2i-1}}$$

es decir

$$\begin{aligned} f(x) &= (1 + x + x^2 + x^3 + \dots) \cdot \\ &\quad (1 + x^3 + x^6 + x^9 + \dots) \cdot \\ &\quad (1 + x^5 + x^{10} + x^{15} + \dots) \dots \end{aligned}$$

1.1.5. Grafo de Ferrer

El **grafo de Ferrer** es una pruea de que el número de formas de descomponer un entero positivo n como suma de m enteros positivos coincide con el número de formas de descomponer el entero n como suma de enteros siendo m el mayor sumando.

Ejemplo:

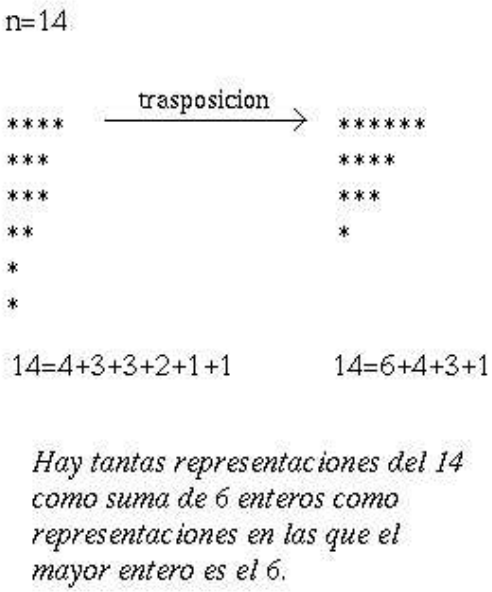

Figura 1: El grafo de Ferrer.

1.1.6. Funciones generatrices exponenciales

Las funciones generatrices que hemos visto hasta ahora, correspondientes a problemas en los que el orden no importa, se denominan **funciones generatrices ordinarias**. Tendremos ahora en cuenta el orden, y veremos un nuevo tipo de funciones generatrices, que

denominaremos **funciones generatrices exponenciales**.

Recordemos que

$$(1+x)^n = \binom{n}{0} + \binom{n}{1}x + \binom{n}{2}x^2 + \ldots + \binom{n}{n}x^n$$

y también que

$$\binom{n}{i} = C(n,i) = \frac{n!}{i!(n-i)!} = \frac{1}{i!}V(n,i)$$

Por tanto podemos escribir

$$(1+x)^n = V(n,0) + V(n,1)x + V(n,2)\frac{x^2}{2!} + \ldots + V(n,n)\frac{x^n}{n!}$$

donde el coeficiente $\dfrac{x^i}{i!}$ de $(1+x)^n$ es $V(n,i)$.

Así pues, definimos:

Para una sucesión de números reales $a_0, a_1, a_2, \ldots$, la función

$$f(x) = a_0 + a_1\frac{x}{1!} + a_2\frac{x^2}{2!} + a_3\frac{x^3}{3!} + \ldots = \sum_{i=0}^{\infty} a_i\frac{x^i}{i!}$$

es la **función generatriz exponencial** de la sucesión dada, $\{a_i\}_{i\in\mathbb{N}}$.

Cuadro 2: Tabla de funciones generatrices exponenciales.

Sucesión	**Expresión**	**Función generatriz exponencial**
$\{1,1,1,\ldots\}$	$\displaystyle\sum_{i=0}^{\infty}\frac{x^i}{i!}$	e^x
$\{a,a^2,a^3,\ldots\} = \{a^i\}_{i\in\mathbb{N}}$	$\displaystyle\sum_{i=0}^{\infty}\frac{(ax)^i}{i!} = \sum_{i=0}^{\infty}a^i\frac{x^i}{i!}$	e^{ax}
$\{(-1)^i\}_{i\in\mathbb{N}}$	$\displaystyle\sum_{i=0}^{\infty}(-1)^i\frac{x^i}{i!}$	e^{-x}

. . . . . Continúa en la página siguiente . . . . . .

Cuadro 2: Tabla de funciones generatrices exponenciales.

. . . . . Continuación . . . . . .

$\{1, 0, 1, 0, 1, \dots\}$	$\sum_{i=0}^{\infty} \frac{x^{2i}}{i!}$	$\frac{e^x + e^{-x}}{2}$
$\{0, 1, 0, 1, \dots\}$	$\sum_{i=0}^{\infty} \frac{x^{2i+1}}{(2i+1)!}$	$\frac{e^x - e^{-x}}{2}$

1.1.7. El operador suma

Sea

$$f(x) = \sum_{i=0}^{\infty} a_i x^i$$

la función generatriz de la sucesión $\{a_i\}_{i\in\mathbb{N}}$.

Entonces, la función generatriz de la sucesión

$$\{a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots\} = \left\{\sum_{k=0}^{i} a_k\right\}_{i\in\mathbb{N}}$$

puede conocerse sabiendo que

$$g(x) = \frac{1}{1-x}$$

es la función generatriz de $\{1\}$ y calculando la convolución de ambas sucesiones, pues su término i-ésimo sería

$$c_i = \sum_{k=0}^{i} a_k \cdot 1 = \sum_{k=0}^{i} a_k$$

y por tanto la función generatriz de

$$\left\{\sum_{k=0}^{i} a_k\right\}_{i\in\mathbb{N}}$$

es

$$h(x) = \frac{f(x)}{(1-x)}$$

1.2. Relaciones de Recurrencia

Una **relación de recurrencia** es una fórmula que, para un entero $n \geq 1$, relaciona el n-ésimo término de una sucesión $A = \{a_i\}_{i=0}^{\infty}$ con uno o varios de los términos anteriores $a_0, a_1, \ldots, a_{n-1}$.

Por ejemplo:

$\surd$ La suma de los n primeros naturales, S_n:

$$S_n = n + S_{n-1}$$

$\surd$ Progresión aritmética de distancia d:

$$a_n = a_{n-1} + d$$

$\surd$ Progresión geométrica de razón r:

$$a_n = a_{n-1} \cdot r$$

Sean n y k enteros positivos. Una relación de recurrencia del tipo

$$f(n) = c_0(n)a_n + c_1(n)a_{n-1} + \ldots + c_k(n)a_{n-k}$$

para $n \geq k$, donde $c_0(n), c_1(n), \ldots c_k(n)$ y $f(n)$ son funciones de n, se denomina **relación de recurrencia lineal**.

- Si $c_0(n)$ y $c_k(n)$ no son nulos se dirá que la relación es de *grado* k.
- Si las funciones $c_0(n), c_1(n), \ldots c_k(n)$ son constantes se dirá que es una **relación de recurrencia lineal con coeficientes constantes**.
- Si $f(n)$ es nula se dirá que es una relación de recurrencia **homogénea**, en otro caso se dirá que no es homogénea.

Así, por ejemplo, $c_n = 3c_{n-1}$ es una relación de recurrencia lineal con coeficientes constantes homogénea, para la que tanto $\{1, 3, 9, \ldots\}$ como $\{2, 6, 18, \ldots\}$ son soluciones.

De modo que para encontrar una solución concreta es necesario que se proporcionen los k primeros términos de la sucesión, con k el grado de la relación de recurrencia. Estos términos se denominan **condiciones frontera** o **condiciones iniciales**.

1.2.1. Resolución de relaciones de recurrencia por el método de las funciones generatrices

Para resolver una recurrencia como

$$a_n - 3a_{n-1} = n$$

con $n \geq 1$ y $a_0 = 1$ (condición inicial), multiplicamos la igualdad que representa la recurrencia por x^n para $n \geq 1$, donde n es el mayor subíndice que aparece en la expresión:

$$\forall n \geq 1 \qquad a_n x^n - 3a_{n-1}x^n = nx^n$$

Sumamos todas ellas desde $n = 1$ hasta infinito,

$$\sum_{n=1}^{\infty} a_n x^n - 3\sum_{n=1}^{\infty} a_{n-1}x^n = \sum_{n=1}^{\infty} nx^n$$

Definimos

$$f(x) = \sum_{n=0}^{\infty} a_n x^n$$

como la función generatriz de la sucesión $\{a_0, a_1, a_2, \dots\}$ y sustituimos la igualdad anterior con las oportunas adecuaciones, esto es, haciendo coincidir los subíndices de las as con los exponentes de las xs:

$$(f(x) - a_0)\, 3xf(x) = \frac{x}{(1-x)^2}$$

Sustituyendo el valor de a_0,

$$(f(x) - 1) - 3xf(x) = \frac{x}{(1-x)^2}$$

Despejando

$$(1 - 3x)f(x) = \frac{x}{(1-x)^2} + 1$$

$$f(x) = \frac{x}{(1-3x)(1-x)^2} + \frac{1}{(1-3x)}$$

Expresando la primera de las fracciones como suma de fracciones simples

$$f(x) = \frac{\frac{7}{4}}{(1-3x)} + \frac{\frac{-1}{4}}{1-x} + \frac{\frac{-1}{2}}{(1-x)^2}$$

Y así,

$$a_n = \frac{7}{4}3^n + \frac{-1}{4} + \frac{-1}{2}\binom{n+1}{n} = \frac{7}{4}3^n - \left(\frac{n+1}{2} + \frac{1}{4}\right) \qquad n \geq 0$$

1.2.2. Recurrencias no lineales

No hay un método general para resolver **recurrencias no lineales**. Lo que se hace es intentar *transformarlas* para que sean lineales.

√ Para resolver

$$a_{n+2}^2 - 5a_{n+1}^2 + 6a_n^2 = 7n \qquad n \geq 0 \quad a_0 = a_1 = 1$$

se hace el cambio $a_n^2 = b_n$ y queda

$$b_{n+2} - 5b_{n+1} + 6b_n = 7n \qquad n \geq 0 \quad b_0 = b_1 = 1^2 = 1$$

√ Para resolver

$$a_n^2 - 2a_{n-1} = 0 \qquad n \geq 1 \quad a_0 = 2$$

se toman logaritmos $b_n = \log_2 a_n$ y queda

$$2b_n - b_{n-1} = 0 \qquad n \geq 0 \quad b_0 = 1$$

Índice

Referencias

[1] Freire Nistal, José Luis. *Apuntes de Matemáticas Discretas II.* Departamento de Computación. Facultad de Informática, Universidad de La Coruña, 2001.

[2] John Grimaldi. *Matemáticas discreta y combinatoria.* Prentice Hall, 1978.