TEMA 1.- INTRODUCCIÓN (Faltan algunas cosas vistas en clase)

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

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

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

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

# TEMA 2.- CÁLCULO DE CONSTRUCCCIONES INDUCTIVAS

## 2.1.- INTRODUCCION

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

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

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

  Ej.-

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

  se puede leer

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

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

## 2.2.- TÉRMINOS

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

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

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

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

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

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

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

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

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

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

2.2.1.- Sorts

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

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

- El conjunto de sorts se denota:

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

2.2.2.- Constantes

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

2.2.3.- Descripción formal del lenguaje

- Tipos.

  Los tipos pueden dividirse en:

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

- Términos.

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

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

- Notación.

  La aplicación es asociativa por la izquierda:

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

  y el producto y las flechas por la derecha:

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

  $\forall x\ y : A, B$ o $\lambda x\ y : A, B$ denota el producto o la abstracción de variables del mismo tipo, es decir, $\forall x : A, \forall y : A, B$ o $\lambda x : A, \lambda y : A, B$.

- Sustituciones.

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

2.3.- TÉRMINOS TIPADOS

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

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

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

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

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

- W-E:

$$\mathcal{W}F([])[[]]$$

- W-$s$:

$$\frac{E[\Gamma] \vdash T : s \quad s \in \mathcal{S} \quad x \notin \Gamma \cup E}{\mathcal{WF}(E)[\Gamma :: (x : T)]}$$

  Esto significa que si no tengo declarada $x$ en mi contexto y $T$ es válido, entonces puedo añadir $(x : T)$ a éste contexto.

- Regla de tipado Var:

$$\frac{\mathcal{WF}(E)[\Gamma] \quad (x : T) \in \Gamma}{E[\Gamma] \vdash x : T}$$

- Regla de tipado Const:

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

- Prod:

$$\frac{E[\Gamma] \vdash T : s_1 \quad E[\Gamma :: (x : T)] \vdash U : s_2 \quad s_1, s_2 \in \{\texttt{Prop}, \texttt{Set}\}}{E[\Gamma] \vdash (x : T)U : s_2}$$

  Esto simplemente nos permite definir el producto en el entorno.

- Lam:

$$\frac{E[\Gamma] \vdash (x : T)U : s \quad E[\Gamma :: (x : T)] \vdash t : U}{E[\Gamma] \vdash [x : T]t : (x : T)U}$$

  Esto simplemente nos permite definir las $\lambda$-expresiones en el entorno.

- App:

$$\frac{E[\Gamma] \vdash t : (x : U)T \quad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{x/u\}}$$

  Esto define la aplicación en el entorno.

### 2.4.- CCI EN COQ

- Inicialización del intérprete COQ:

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

Coq <
```

- Las órdenes a COQ acaban en punto.

```
Coq < Quit.
```

  Cierra el intérprete COQ.

- COQ es sensible a mayúsculas/minúsculas.

#### 2.4.1.- Declaraciones

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

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

```
Coq < Check O.

O
     : nat
```

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

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

- También hay algunas funciones predefinidas.

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

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

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

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

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

2.4.2.- Definiciones

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

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

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

Coq < Check one.
one
     : nat
```

- Hay varias sintaxis posibles:

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

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

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

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

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

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

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

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

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

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


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

- También podemos expresar el cuantificador universal. Por ejemplo:

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

  viene a significar $\forall m \in \mathbb{N}.m > 0$, ya que `m` aparece en el segundo término. Esta construcción se denomina *producto dependiente*.

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

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

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

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

- Resumen: en Coq las construcciones permitidas son

  `x | (M N) | fun x : T => U | forall x:T, P`

  donde

  - `x` denota constantes o variables.
  - `(M N)` denota la aplicación.
  - `fun x : T =>U` representa la $\lambda$-expresión de parámetro `x` y cuerpo `M`.
  - `forall x:T, P` representa el producto. Si es dependiente, expresa la cuantificación universal. Si es independiente, representa el tipo de las funciones entre ambos tipos.

## 2.5.- EJEMPLOS DEL CÁLCULO DE PROPOSICIONES

### 2.5.1.- Ejemplo I

- Definición de variables.

```
Coq < Variable A:Prop.
Warning: A is declared as a parameter because it is at a global level
A is assumed

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

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

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

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

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

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

- Táctica `intro`.

```
var < intro.
1 subgoal

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

- `intro` hace lo siguiente:

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

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

- Assumption.

```
var < assumption.
Proof completed.
```

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

- Qed.

```
var < Qed.
intro.
assumption.
var is defined
```

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

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

2.5.2.- Ejemplo II

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

```
Coq < Lemma trivial: forall p:Prop, p->p.
1 subgoal

  ============================
   forall p : Prop, p -> p
```

- `intros` ejecuta `intro` todas las veces que pueda.

```
trivial < intros.
1 subgoal

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

- `Undo` da un paso atrás.

```
trivial < Undo.
1 subgoal

  ============================
   forall p : Prop, p -> p
```

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

```
trivial < intros q H.
1 subgoal

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

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

```
trivial < exact H.
Proof completed.

trivial < Qed.
intros q H.
exact H.
trivial is defined
```

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

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

Argument scopes are [type_scope _]
```

2.5.3.- Ejemplo III

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

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

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

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

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

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

```
Unnamed_thm < intros.
1 subgoal

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

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

```
Unnamed_thm < exact ((H H1) (H0 H1)).
Proof completed.

Unnamed_thm < Qed.
intros.
exact (H H1 (H0 H1)).
Unnamed_thm is defined
```

- Chequeamos.

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

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

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

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

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

Unnamed_thm < auto.
Proof completed.

Unnamed_thm < Save prueba.
auto.
prueba is defined
```

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

## 2.6.- EJEMPLOS DE CÁLCULO DE PREDICADOS

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

```
Coq < Section prueba.

Coq < Variable A:Prop.
A is assumed

Coq < Check A.
A
     : Prop

Coq < End prueba.

Coq < Check A.
Toplevel input, characters 61-62
```

```
>  Check A.
>        ^
Error: The reference A was not found in the current environment

Coq <
```

- Las secciones permiten anidamientos.

2.6.1.- Ejemplo IV

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

```
Coq < Section CalculoPredicados.

Coq < Variable D:Set.
D is assumed

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

Coq < Section RSimTrans.

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

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

- Ahora introducimos el lema a probar:

```
Coq < Lemma RReflex: forall x : D , (exists y, R x y) -> R x x.
1 subgoal

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

RReflex <
```

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

  ```
  (exists x : D, P x)
  ```

  no es más que una sintaxis de `(ex D (fun x:D =>P x))`. Esta notación expresa el cuantificador existencial ($\exists x \in D, P(x)$) y se construye a partir del operador `ex`, que espera un predicado como argumento.

  ```
  Coq < Check ex.
  ex
        : forall A : Type, (A -> Prop) -> Prop
  ```

- Como primer paso eliminamos los productos.

  ```
  RReflex < intros.
  1 subgoal

    D : Set
    R : D -> D -> Prop
    RSim : forall x y : D, R x y -> R y x
    RTrans : forall x y z : D, R x y -> R y z -> R x z
    x : D
    H : exists y : D, R x y
    ============================
     R x x
  ```

- Nótese que `intros` trata el cuantificador universal como las premisas de las implicaciones (ambas construcciones son productos en el CCI).

- Podríamos haber obtenido lo mismo con:

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

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

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

```
RReflex < Undo.
RReflex < intros.
1 subgoal

  D : Set
  R : D -> D -> Prop
  RSim : forall x y : D, R x y -> R y x
  RTrans : forall x y z : D, R x y -> R y z -> R x z
  x : D
  H : exists y : D, R x y
  ============================
   R x x
```

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

```
RReflex < elim H.
1 subgoal

  D : Set
  R : D -> D -> Prop
  RSim : forall x y : D, R x y -> R y x
  RTrans : forall x y z : D, R x y -> R y z -> R x z
  x : D
  H : exists y : D, R x y
  ============================
   forall x0 : D, R x x0 -> R x x

RReflex < intros y Rxy.
1 subgoal

  D : Set
  R : D -> D -> Prop
  RSim : forall x y : D, R x y -> R y x
  RTrans : forall x y z : D, R x y -> R y z -> R x z
  x : D
  H : exists y : D, R x y
  y : D
  Rxy : R x y
  ============================
   R x x
```

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

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

  ...

  ============================
   R x y

subgoal 2 is:
 R y x
```

- Se finaliza la primera rama de la demostración

```
RReflex < assumption.
1 subgoal

  D : Set
  R : D -> D -> Prop
  RSim : forall x y : D, R x y -> R y x
  RTrans : forall x y z : D, R x y -> R y z -> R x z
  x : D
  H : exists y : D, R x y
  y : D
  Rxy : R x y
  ============================
   R y x
```

- Y la segunda.

```
RReflex < apply RSim.
1 subgoal

  D : Set
  R : D -> D -> Prop
  RSim : forall x y : D, R x y -> R y x
  RTrans : forall x y z : D, R x y -> R y z -> R x z
  x : D
  H : exists y : D, R x y
  y : D
  Rxy : R x y
  ============================
   R x y

RReflex < assumption.
Proof completed.
```

2.6.2.- Ejemplo VI

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

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

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

Coq < Variable d:D.
d is assumed

Coq < Lemma cuantif: (forall x:D, P x) -> exists a, P a.
1 subgoal

  D : Set
  P : D -> Prop
  d : D
  ============================
   (forall x : D, P x) -> exists a : D, P a
```

- Eliminamos la implicación pasando al campo de las hipótesis el antecedente.

```
cuantif < intros.
1 subgoal

  D : Set
  P : D -> Prop
  d : D
  H : forall x : D, P x
  ============================
   exists a : D, P a

cuantif < exists d.
1 subgoal

  D : Set
  P : D -> Prop
  d : D
  H : forall x : D, P x
  ============================
   P d

cuantif < trivial.
Proof completed.

cuantif < Qed.
intros.
exists d.
trivial.
cuantif is defined
```

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

- `Print` *ident*. Proporciona información sobre el objeto *ident*.
- `Print All`. Proporciona información sobre el estado del entorno.
- `Search` *ident*. Proporciona el nombre y tipo de todos los teoremas del contexto actual cuyo consecuente tiene la forma `(ident t1 ... tn)`.
- `Load` *ident*. Carga el fichero de texto *ident.v* que no tiene que ir entre comillas. En vez de *ident* se puede especificar la ruta absoluta o relativa del fichero en una cadena (entre comillas).

- `Read Module` *ident*. Carga el módulo almacenado en el fichero de nombre *ident* pero no lo abre: sus contenidos son invisibles para el usuario.
- `Import` *ident*. Abre el módulo previamente cargado con `Read Module`. Sus contenidos resultan visibles para el usuario.
- `Require` *ident*. Carga y abre un mdulo. Si el módulo y ha sido cargado con `Read Module`, sólo lo abre.

  Variante: `Require` *ident ruta*, donde se define entre comillas la ruta al fichero.
- `Declare ML Module` $ruta_1$ ... $ruta_n$. Carga los ficheros compilados con definiciones de *Objective Caml*.
- `Pwd`. Muestra el directorio actual.
- `Cd` *ruta*.
- `AddPath` *ruta*. Añade una nueva ruta a la lista de directorios del sistema de ficheros en los que Coq busca módulos y librerías al invocar comandos como `Read Module` o `Require`.
- `DelPath` *ruta*. Lo contrario a lo anterior.
- `Print LoadPath`. Muestra la lista de directorios del sistema.
- `Locate File` *nombre*. Devuelve la ruta completa del fichero si está en algún directorio de la lista de directorios del sistema.
- `Reset` *ident*. Elimina todos los objetos que han sido introducidos en el entorno actual desde *ident*, él mismo incluído.
- `Save State` *ident*. Guarda el estado actual (los objetos definidos) de modo que se pueda volver a él desde más adelante si es necesario.
- `Print States`. Muestra la lista de estados definidos.
- `Restore State` *ident*. Vuelve al estado *ident*.

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

  `Theorem` *ident* `:` *term*

  `Lemma` *ident* `:` *term*

  `Remark` *ident* `:` *term*

`Fact` *ident* `:` *term*

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

- `Qed`

  `Save`

  `Save Theorem`

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

- `Proof` *term*. Cuando se está realizando una prueba, es equivalente a `Exact` *Term* `; Save`.

- `Abort`. Cancela la prueba actual volviendo al entorno anterior.

- `Suspend`. Vuelve al entorno anterior a la prueba actual, pero sin cancelarla.

- `Resume`. Vuelve a la última prueba suspendida.

- `Undo`. Cancela los efectos de la última táctica aplicada.

- `Restart`. Reinicia el proceso de prueba.

- `Focus`. Focaliza la prueba en el primer subobjetivo, haciendo invisibles los demás.

- `Unfocus`. Deshace los efectos del comando anterior.

- `Show`. Muestra los objetivos actuales.