TEMA 3.- TIPOS DE DATOS INDUCTIVOS

3.1.- Definición de tipos inductivos

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

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

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

  Por ejemplo:

```
Coq < Check bool_ind.
bool_ind
     : forall P : bool -> Prop, P true -> P false -> forall b : bool, P b
```

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

- Probamos que todo booleano es cierto o falso:

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

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

```
dualidad < intro b.
1 subgoal

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

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

```
dualidad < elim b.
2 subgoals

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

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

dualidad <
```

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

```
dualidad < left.
2 subgoals

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

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

dualidad < trivial.
1 subgoal

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

- Y después `right`

```
dualidad < right.
1 subgoal

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

dualidad < trivial.
Proof completed.
```

- En cuanto a los números naturales:

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

Coq < Check nat.
nat
     : Set

Coq < Check O.
O
     : nat

Coq < Check S.
S
     : nat->nat

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

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

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

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

Coq < Lemma tonto: forall n:nat, predicado n.
1 subgoal

  ============================
   forall n : nat, predicado n

tonto < intro.
1 subgoal

  n : nat
  ============================
   predicado n

tonto < elim n.
2 subgoals

  n : nat
  ============================
   predicado O

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

- En cuanto a las listas:

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

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

```
Coq < Check Lista.
Lista
     : Set

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

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

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

Coq < Lemma tonto: forall L:Lista, propiedad_lista L.
1 subgoal

  ============================
   forall L : Lista, propiedad_lista L

tonto < intro L.
1 subgoal

  L : Lista
  ============================
   propiedad_lista L

tonto < elim L.
2 subgoals

  L : Lista
  ============================
   propiedad_lista nula

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

3.2.- Demostraciones con tipos inductivos.

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

```
Coq < Lemma suma_n_0: forall n:nat, n = plus n 0.
1 subgoal

  ============================
   forall n : nat, n = n + 0

suma_n_0 < intro n.
1 subgoal

  n : nat
  ============================
   n = n + 0

suma_n_0 < elim n.
2 subgoals

  n : nat
  ============================
   0 = 0 + 0

subgoal 2 is:
 forall n0 : nat, n0 = n0 + 0 -> S n0 = S n0 + 0
```

- La táctica `simpl` reescribe el objetivo.

```
suma_n_0 < simpl.
2 subgoals

  n : nat
  ============================
   0 = 0

subgoal 2 is:
 forall n0 : nat, n0 = n0 + 0 -> S n0 = S n0 + 0
```

```
suma_n_0 < trivial.
1 subgoal

  n : nat
  ============================
   forall n0 : nat, n0 = n0 + 0 -> S n0 = S n0 + 0
```

- Usamos `intros` y `simpl`, para poder utilizar una táctica nueva: `rewrite`

```
suma_n_0 < intros.
1 subgoal

  n : nat
  n0 : nat
  H : n0 = n0 + 0
  ============================
   S n0 = S n0 + 0

suma_n_0 < simpl.
1 subgoal

  n : nat
  n0 : nat
  H : n0 = n0 + 0
  ============================
   S n0 = S (n0 + 0)
```

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

```
suma_n_0 < rewrite <- H.
1 subgoal

  n : nat
  n0 : nat
  H : n0 = n0 + 0
  ============================
   S n0 = S n0

suma_n_0 < trivial.
Proof completed.
```

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

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

Coq <
```

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

```
Coq < Hint Resolve suma_n_O.
```

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

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

  ============================
   forall m n : nat, S (n + m) = n + S m

suma_n_S < intros.
1 subgoal

  m : nat
  n : nat
  ============================
   S (n + m) = n + S m

suma_n_S < elim n.
2 subgoals

  m : nat
  n : nat
  ============================
   S (0 + m) = 0 + S m

subgoal 2 is:
 forall n0 : nat, S (n0 + m) = n0 + S m -> S (S n0 + m) = S n0 + S m
```

- O también, con `induction`, que aplica los `intros` necesarios antes de aplicar `elim`.

```
suma_n_S < Restart.
1 subgoal

  ============================
   forall m n : nat, S (n + m) = n + S m

suma_n_S < induction n.
2 subgoals

  m : nat
  ============================
   S (0 + m) = 0 + S m

subgoal 2 is:
 S (S n + m) = S n + S m
```

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

```
suma_n_S < simpl.
2 subgoals

  m : nat
  ============================
   S m = S m

subgoal 2 is:
 S (S n + m) = S n + S m

suma_n_S < trivial.
1 subgoal

  m : nat
  n : nat
  IHn : S (n + m) = n + S m
  ============================
   S (S n + m) = S n + S m
```

- Aunque con `auto` podemos terminar aquí, seguiremos manualmente:

```
suma_n_S < simpl.
1 subgoal

  m : nat
  n : nat
  IHn : S (n + m) = n + S m
  ============================
   S (S (n + m)) = S (n + S m)
```

- Terminamos con `apply` y `trivial`.

```
suma_n_S < apply eq_S.
1 subgoal

  m : nat
  n : nat
  IHn : S (n + m) = n + S m
  ============================
   S (n + m) = n + S m

suma_n_S < trivial.
Proof completed.

suma_n_S < Save.
induction n.
 simpl in |- *.
   trivial.
 intros.
   simpl in |- *.
   apply eq_S.
   trivial.
suma_n_S is defined

Coq < Hint Resolve suma_n_S.
```

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

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

  ============================
   forall n m : nat, n + m = m + n

suma_com < induction m.
2 subgoals

  n : nat
  ============================
   n + 0 = 0 + n

subgoal 2 is:
 n + S m = S m + n

suma_com < simpl.
2 subgoals

  n : nat
  ============================
   n + 0 = n

subgoal 2 is:
 n + S m = S m + n

suma_com < auto.
1 subgoal

  n : nat
  m : nat
  IHm : n + m = m + n
  ============================
   n + S m = S m + n
```

- `auto` funciona porque tenemos `suma_n_O` metido como `Hint`. Manualmente:

```
suma_com < Undo.
2 subgoals

  n : nat
  ============================
   n + 0 = n

subgoal 2 is:
 n + S m = S m + n

suma_com < symmetry.
2 subgoals

  n : nat
  ============================
   n = n + 0

subgoal 2 is:
 n + S m = S m + n

suma_com < apply suma_n_0.
1 subgoal

  n : nat
  m : nat
  IHm : n + m = m + n
  ============================
   n + S m = S m + n
```

- Vamos a por el segundo subobjetivo.

```
suma_com < simpl.
1 subgoal

  n : nat
  m : nat
  IHm : n + m = m + n
  ============================
   n + S m = S (m + n)
```

```
suma_com < rewrite <- IHm.
1 subgoal

  n : nat
  m : nat
  IHm : n + m = m + n
  ============================
   n + S m = S (n + m)

suma_com < auto.
Proof completed.
```

3.3.- Funciones y predicados recursivos e inductivos.

- Sintaxis de la definición por casos:

  `[anotacion] match` *term0* `with |` *term1* => *patrón1* `| ... |` *termN* => *patrónN*

- Sintaxis de las funciones recursivas:

  `Fixpoint` *ident* `(`*ident1* `:` *tipo1* `) ... (` *identN* `:` *tipoN* `) {struct` *identI* `}:` *tipo0* `:=` *term0*

  `Fixpoint` *ident1 ligadura1* `{struct` *ident1I* `}:` *tipo1* `:=` *term1*

  `with ...`

  `with` *identN ligaduraN* `{struct` *identNJ* `}:` *tipoN* `:=` *termN*

  Ejemplo:

```
Coq < Fixpoint plus (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | O => m
Coq <   | S p => S (plus p m)
Coq <   end.
plus is recursively defined
```

- Definición de Predicados recursivos.

```
Coq < Definition Is_S (n:nat) := match n with
Coq <                              | O => False
Coq <                              | S p => True
Coq <                            end.
Is_S is defined
```

- Demostremos que `Is_S (S n)` es cierto para todo n.

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

  ============================
   forall n : nat, Is_S (S n)

S_Is_S < simpl.
1 subgoal

  ============================
   nat -> True

S_Is_S < trivial.
Proof completed.
```

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

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

  ============================
   forall n : nat, 0 <> S n
```

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

```
no_confuso < red.
1 subgoal

  ============================
   forall n : nat, 0 = S n -> False

no_confuso < intros.
1 subgoal

  n : nat
  H : 0 = S n
  ============================
   False
```

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

```
no_confuso < change (Is_S 0).
1 subgoal

  n : nat
  H : 0 = S n
  ============================
   Is_S 0
```

- Usamos `rewrite` y `simpl` para obtener `True`.

```
no_confuso < rewrite H.
1 subgoal

  n : nat
  H : 0 = S n
  ============================
   Is_S (S n)

no_confuso < simpl.
1 subgoal

  n : nat
  H : 0 = S n
  ============================
   True

no_confuso < trivial.
Proof completed.
```

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

```
no_confuso < Restart.
1 subgoal

  ============================
   forall n : nat, 0 <> S n
```

```
no_confuso < intros.
1 subgoal

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

no_confuso < discriminate.
Proof completed.
```

- También podemos definir predicados inductivos.

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

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

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

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

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

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

```
le_n_S < intros.
1 subgoal

  n : nat
  m : nat
  H : le n m
  ============================
   le (S n) (S m)
```

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

```
le_n_S < elim H.
2 subgoals

  n : nat
  m : nat
  H : le n m
  ============================
   le (S n) (S n)

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

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

```
le_n_S < apply le_n.
1 subgoal

  n : nat
  m : nat
  H : le n m
  ============================
   forall m0 : nat, le n m0 -> le (S n) (S m0) -> le (S n) (S (S m0))
```

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

```
le_n_S < intros.
1 subgoal

  n : nat
  m : nat
  H : le n m
  m0 : nat
  H0 : le n m0
  H1 : le (S n) (S m0)
  ============================
   le (S n) (S (S m0))

le_n_S < apply le_S.
1 subgoal

  n : nat
  m : nat
  H : le n m
  m0 : nat
  H0 : le n m0
  H1 : le (S n) (S m0)
  ============================
   le (S n) (S m0)

le_n_S < assumption.
Proof completed.
```

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

```
le_n_S < Restart.
1 subgoal

  ============================
   forall n m : nat, le n m -> le (S n) (S m)

le_n_S < Hint Resolve le_n le_S.

le_n_S < induction 1;auto.
Proof completed.
```

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

TEMA 4.- MÁS COQ

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

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

4.2.- Tácticas (ver capítulo 8 del manual)

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

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

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

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

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

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

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

```
Unnamed_thm < eapply RTrans.
2 subgoals

  ============================
   R n ?3

subgoal 2 is:
 R ?3 p

Unnamed_thm < apply Rnm.
1 subgoal

  ============================
   R m p
```

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

```
Coq < Lemma hoho: forall x y : nat, le 0 (plus (plus x y) y).
1 subgoal

  ============================
   forall x y : nat, 0 <= x + y + y

hoho < intros.
1 subgoal

  x : nat
  y : nat
  ============================
   0 <= x + y + y
```

```
hoho < generalize (plus (plus x y) y).
1 subgoal

  x : nat
  y : nat
  ============================
   forall n : nat, 0 <= n

hoho < induction n.
2 subgoals

  x : nat
  y : nat
  ============================
   0 <= 0

subgoal 2 is:
 0 <= S n

hoho < trivial.
1 subgoal

  x : nat
  y : nat
  n : nat
  IHn : 0 <= n
  ============================
   0 <= S n

hoho < info auto.
 == apply le_S; exact IHn.

Proof completed.
```

- `change` *term*. Se aplica a cualquier objetivo. `change U` reemplaza el objetivo actual `T` por `U` si `U` está bien formado y `T` y `U` son convertibles mediante β-reducción o reglas de reescritura.
- `contradiction`. Applicable a cualquier objetivo. Trata de encontrar en el contexto actual (después de aplicados todos los intros) alguna hipótesis equivalente a `False`.

- `red`. Se aplica a cualquier objetivo de la forma

  ```
  forall x1:T1 ... Xk:Tk,  (c x1 ... xn)
  ```

  donde `c` es una constante. Se sustituye `c` por su definición (por ejemplo `t` y se aplica `(t x1 ... xn)` mediante β-reducción o la aplicación de una definición inductiva.

- `simpl`. Aplicable a cualquier objetivo. Primero aplica β-reducción o una definición inductiva. Después intenta sustituír las constantes por su definición y aplicar de nuevo β-reducción o la definición inductiva.

- `unfold` *ident*. Aplicable a cualquier objetivo. ident debe ser el nombre de una constante transparente. Sustituye *ident* por su definición en el objetivo y aplica β-reducción y definiciones inductivas hasta lograr una forma normal.

- `fold` *term*. Aplicable a cualquier objetivo. Se aplica la táctica `red` a cada ocurrencia de *term* en el objetivo.

- `constructor` *n*. Aplicable a un objetivo cuya conclusión sea una constante inductiva. Constructor Aplica `intros` y `apply` sobre el constructor n-ésimo de dicha conclusión.

- `elim` *term*. Aplicable a cualquier objetivo. Escoge el destructor apropiado y lo aplica sobre el objetivo.

- `elim` *term* `with` *term1* `...` *termN*. Lo mismo que lo anterior pero permitiendo dar valores a las premisas dependientes en el esquema de eliminación.

- `case` term. Equivalente a `elim` *term* `with` *term1* `...` *termN*.

- `decompose` *term*. Esta táctica descompone recursivamente una proposición compleja para obtener proposiciones atómicas.

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

    ============================
     forall A B C : Prop, A /\ B /\ C \/ B /\ C \/ C /\ A -> C
  ```

```
decomp < intros A B C H.
1 subgoal

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

decomp < decompose [and or] H.
3 subgoals

  A : Prop
  B : Prop
  C : Prop
  H : A /\ B /\ C \/ B /\ C \/ C /\ A
  H1 : A
  H0 : B
  H3 : C
  ============================
   C

subgoal 2 is:
 C
subgoal 3 is:
 C
```

- `rewrite` *term*. El tipo de *term* (una hipótesis) debe ser

  ```
  forall x1:T1 ... xN:TN, term1 = term2
  ```

  Se reemplazan todas las ocurrencias de *term1* en el objetivo por *term2*.

- `rewrite <-` *term*. Lo mismo que lo anterior, pero es *term2* quien es sustituído por *term1*.

- `replace` *term1* `with` *term2*. Se aplica a cualquier objetivo. Reemplaza las ocurrencias libres de *term1* por *term2* y añade la igualdad *term1*=*term2* como nuevo subobjetivo.

- `reflexivity`. Aplicable a cualquier objetivo con la forma $t = u$. Chequea si t y u son convertibles y resuelve el objetivo.

- `symmetry`. Convierte un objetivo con la forma *t*=*u* a *u*=*t*.
- `compare` *term1 term2*. *term1* y *term2* tienen que ser del mismo tipo inductivo. Si *G* es el subobjetivo actual, es sustituído por los subobjetivos *term1*=*term2* -> *G* y ∼*term1*=*term2* `->G`.
- `discriminate` *ident*. Esta táctica prueba cualquier objetivo constituído por una hipótesis absurda que afirma que dos términos estructuralmente diferentes de un mismo tipo son iguales. Por ejemplo, a partir de la hipótesis `(S (S O))=(S O)`, podemos demostrar por reducción al absurdo cualquier fórmula. Si *ident* es una hipótesis de tipo `term1 = term2` en el contexto actual, `discriminate` busca en la forma normal de `term1` y `term2` dos subtérminos `u` y `w` (resp. de `term1` y `term2`), en las mismas posiciones y con diferentes constructores como símbolos cabeza. Si estos subtérminos existen, la táctica probará el objetivo actual.
- `injection` *ident*. Si ident es una hipótesis de tipo *term1* `=` *term2*, `injection` intenta derivar la igualdad de los subtérminos colocados en la misma posición en ambos. Por ejemplo, de `(S (S n))=(S (S (S m)))` se puede derivar la igualdad `n=(S m)`.
- `simplify_eq` *ident*. Sea *ident* una hipótesis de la forma *term1 term2*. Si *term1* y *term2* son estructuralmente diferentes, se aplica `discriminate`, mientras que si no lo son se aplica `injection`.
- `auto`. Implementa un procedimiento de resolución al estilo Prolog para demostrar el objetivo actual.
- `auto` *num*. Fuerza la profundidad de búsqueda a *num*. El máximo por defecto es 5.
- `auto with` *ident1 ... identN*. Hace uso de las bases de datos de teoremas ya demostrados *ident1 ... identN* además de la habitual.
- `trivial`. Es una restricción de `auto` que no es recursiva y usa sólo teoremas con coste 0. Se utiliza para resolver igualdades triviales del tipo `X = X`.
- `eauto`. Generalización de `auto` que hace uso de un procedimiento de unificación en lugar de emparejamiento de patrones entre el objetivo y los teoremas de la base de datos. En otras palabras, usa `eapply` en lugar de `apply`.