Demostración Asistida por Ordenador con Isabelle/HOL

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Sevilla, 29 de enero de 2013 (versión del 6 de agosto de 2018)

Índice general

Introducción

En este libro se presenta una introducción a la demostación asistida por ordenador usando Isabelle/HOL[1].
La presentación se realiza mediante teorías de ejemplos y de ejercicios.
El contenido del libro se ha utilizado en los cursos de Razonamiento automático[2] del Máster Universitario en Lógica, computación e inteligencia artificial de la Universidad de Sevilla.

Cuaderno de bitácora

En esta sección se registran los cambios realizados en las sucesivas versiones del libro.

Versión del 6 de agosto de 2018

- Se ha cambiado el título de "DAO: Demostración Asistida por Ordenador" a "Demostración asistida por ordenador con Isabelle/HOL".
- Se ha añadido la introducción.
- Se ha resaltado el código usando la librería minted[3]

[1] https://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html
[2] http://www.cs.us.es/~jalonso/cursos/m-ra
[3] https://github.com/gpoore/minted

Capítulo 1

Programación funcional con Isabelle/HOL

1.1. Tema: Programación funcional con Isabelle/HOL

```
chapter {* Tema 1: Programación funcional en Isabelle *}

theory T1
imports Main
begin

section {* Introducción *}

text {* Esta notas son una introducción a la demostración asistida
  utilizando el sistema Isabelle/HOL/Isar. La versión de Isabelle
  utilizada es la 2012.

  En este capítulos se presenta el lenguaje funcional que está
  incluido en Isabelle. El lenguaje funcional es muy parecido a
  Haskell. *}

section {* Números naturales, enteros y booleanos *}

text {* En Isabelle están definidos los número naturales con la sintaxis
  de Peano usando dos constructores: 0 (cero) y Suc (el sucesor).

  Los números como el 1 son abreviaturas de los correspondientes en la
  notación de Peano, en este caso "Suc 0".
```

```
  El tipo de los números naturales es nat.

  Por ejemplo, el siguiente del 0 es el 1. *}

value "Suc 0" -- "= 1"

text {* En Isabelle está definida la suma de los números naturales:
  (x + y) es la suma de x e y.

  Por ejemplo, la suma de los números naturales 1 y 2 es el número
  natural 3. *}

value "(1::nat) + 2" -- "= 3"

text {* La notación del par de dos puntos se usa para asignar un tipo a
  un término (por ejemplo, (1::nat) significa que se considera que 1 es
  un número natural).

  En Isabelle está definida el producto de los números naturales:
  (x * y) es el producto de x e y.

  Por ejemplo, el producto de los números naturales 2 y 3 es el número
  natural 6. *}

value "(2::nat) * 3" -- "= 6"

text {* En Isabelle está definida la división de números naturales:
  (n div m) es el cociente entero de x entre y.

  Por ejemplo, la división natural de 7 entre 3 es 2. *}

value "(7::nat) div 3" -- "= 2"

text {* En Isabelle está definida el resto de división de números
  naturales: (n mod m) es el resto de dividir n entre m.

  Por ejemplo, el resto de dividir 7 entre 3 es 1. *}

value "(7::nat) mod 3" -- "= 1"
```

```
text {* En Isabelle también están definidos los números enteros. El tipo
  de los enteros se representa por int.

  Por ejemplo, la suma de 1 y -2 es el número entero -1. *}

value "(1::int) + -2" -- "= -1"

text {* Los numerales están sobrecargados. Por ejemplo, el 1 puede ser
  un natural o un entero, dependiendo del contexto.

  Isabelle resuelve ambigüedades mediante inferencia de tipos.

  A veces, es necesario usar declaraciones de tipo para resolver la
  ambigüedad.

  En Isabelle están definidos los valores booleanos (True y False), las
  conectivas (ň, , ,  y ) y los cuantificadores ( y ).

  El tipo de los booleanos es bool. *}

text {* La conjunción de dos fórmulas verdaderas es verdadera. *}
value "True  True" -- "= True"

text {* La conjunción de un fórmula verdadera y una falsa es falsa. *}
value "True  False" -- "= False"

text {* La disyunción de una fórmula verdadera y una falsa es
  verdadera. *}
value "True  False" -- "= True"

text {* La disyunción de dos fórmulas falsas es falsa. *}
value "False  False" -- "= False"

text {* La negación de una fórmula verdadera es falsa. *}
value "ňTrue" -- "= False"

text {* Una fórmula falsa implica una fórmula verdadera. *}
value "False  True" -- "= True"

text {* Un lema introduce una proposición seguida de una demostración.
```

```
  Isabelle dispone de varios procedimientos automáticos para generar
  demostraciones, uno de los cuales es el de simplificación (llamado
  simp).

  El procedimiento simp aplica un conjunto de reglas de reescritura, que
  inicialmente contiene un gran número de reglas relativas a los objetos
  definidos. *}

text {* Ej. de simp: Todo elemento es igual a sí mismo. *}
lemma "x. x = x"
by simp

text {* Ej. de simp: Existe un elemento igual a 1. *}
lemma "x. x = 1"
by simp

section {* Definiciones no recursivas *}

text {* La disyunción exclusiva de A y B se verifica si una es verdadera
  y la otra no lo es. *}

definition xor :: "bool  bool  bool" where
  "xor A B  (A  ¬B)  (¬A  B)"

text {* Prop.: La disyunción exclusiva de dos fórmulas verdaderas es
  falsa.

  Dem.: Por simplificación, usando la definición de la disyunción
  exclusiva.
*}

lemma "xor True True = False"
by (simp add: xor_def)

text {* Se añade la definición de la disyunción exclusiva al conjunto de
  reglas de simplificación automáticas. *}

declare xor_def[simp]
```

```
lemma "xor True False = True"
by simp

section {* Definiciones locales *}

text {* Se puede asignar valores a variables locales mediante 'let' y
  usarlo en las expresiones dentro de 'in'.

  Por ejemplo, si x es el número natural 3, entonces "x*x=9". *}

value "let x = 3::nat in x * x" -- "= 9"

section {* Pares *}

text {* Un par se representa escribiendo los elementos entre paréntesis
  y separados por coma.

  El tipo de los pares es el producto de los tipos.

  La función fst devuelve el primer elemento de un par y la snd el
  segundo.

  Por ejemplo, si p es el par de números naturales (2,3), entonces la
  suma del primer elemento de p y 1 es igual al segundo elemento de
  p. *}

value "let p = (2,3)::nat × nat in fst p + 1 = snd p"

section {* Listas *}

text {* Una lista se representa escribiendo los elementos entre
  corchetes y separados por comas.

  La lista vacía se representa por [].

  Todos los elementos de una lista tienen que ser del mismo tipo.

  El tipo de las listas de elementos del tipo a es (a list).

  El término (x#xs) representa la lista obtenida añadiendo el elemento x
```

```
  al principio de la lista xs.

  Por ejemplo, la lista obtenida añadiendo sucesivamente a la lista
  vacía los elementos c, b y a es [a,b,c]. *}

value "a#(b#(c#[]))" -- "= [a,b,c]"

text {* Funciones de descomposición de listas:
  · (hd xs) es el primer elemento de la lista xs.
  · (tl xs) es el resto de la lista xs.

  Por ejemplo, si xs es la lista [a,b,c], entonces el primero de xs es a
  y el resto de xs es [b,c]. *}

value "let xs = [a,b,c] in hd xs = a  tl xs = [b,c]" -- "= True"

text {* (length xs) es la longitud de la lista xs. Por ejemplo, la
  longitud de la lista [1,2,3] es 3. *}

value "length [1,2,3]" -- "= 3"

text {* En la sesión 47 de "Isabelle/HOL  Higher-Order Logic"
  http://goo.gl/sFsFF se encuentran más definiciones y propiedades de
  las listas. *}

section {* Registros *}

text {* Un registro es una colección de campos y valores.

  Por ejemplo, los puntos del plano se pueden representar mediante
  registros con dos campos, las coordenadas, con valores enteros. *}

record punto =
  coordenada_x :: int
  coordenada_y :: int

text {* Ejemplo, el punto pt tiene de coordenadas 3 y 7. *}

definition pt :: punto where
  "pt  (|coordenada_x = 3, coordenada_y = 7|)"
```

```
text {* Ejemplo, la coordenada x del punto pt es 3. *}

value "coordenada_x pt" -- "= 3"

text {* Ejemplo, sea pt2 el punto obtenido a partir del punto pt
  cambiando el valor de su coordenada x por 4. Entonces la coordenada x
  del punto pt2 es 4. *}

value "let pt2 = pt(|coordenada_x:=4|) in coordenada_x (pt2)" -- "= 4"

section {* Funciones anónimas *}

text {* En Isabelle pueden definirse funciones anónimas.

  Por ejemplo, el valor de la función que a un número le asigna su doble
  aplicada a 1 es 2. *}

value "(x. x + x) 1::nat" -- "= 2"

section {* Condicionales *}

text {* El valor absoluto del entero x es x, si "x  0" y es -x en caso
  contrario. *}

definition absoluto :: "int  int" where
  "absoluto x  (if x  0 then x else -x)"

text {* Ejemplo, el valor absoluto de -3 es 3. *}

value "absoluto(-3)" -- "= 3"

text {* Def.: Un número natural n es un sucesor si es de la forma
  (Suc m). *}

definition es_sucesor :: "nat  bool" where
  "es_sucesor n  (case n of
    0      False
  | Suc m  True)"
```

```
text {* Ejemplo, el número 3 es sucesor. *}

value "es_sucesor 3" -- "= True"

section {* Tipos de datos y definiciones recursivas *}

text {* Una lista de elementos de tipo a es la lista Vacia o se obtiene
  añadiendo, con Cons, un elemento de tipo a a una lista de elementos de
  tipo a. *}

datatype 'a Lista = Vacia | Cons 'a "'a Lista"

text {* (conc xs ys) es la concatenación de las lista xs e ys. Por
  ejemplo,
     conc (Cons a (Cons b Vacia)) (Cons c Vacia)
     = Cons a (Cons b (Cons c Vacia))
*}

fun conc :: "'a Lista  'a Lista  'a Lista" where
  "conc Vacia ys       = ys"
| "conc (Cons x xs) ys = Cons x (conc xs ys)"

value "conc (Cons a (Cons b Vacia)) (Cons c Vacia)"
-- "= Cons a (Cons b (Cons c Vacia))"

text {* (suma n) es la suma de los primeros n números naturales. Por
  ejemplo,
     suma 3 = 6
*}

fun suma :: "nat  nat" where
  "suma 0       = 0"
| "suma (Suc m) = (Suc m) + suma m"

value "suma 3" -- "= 6"

text {* (sumaImpares n) es la suma de los n primeros números
  impares. Por ejemplo,
     sumaImpares 3 = 9
*}
```

```
fun sumaImpares :: "nat  nat" where
  "sumaImpares 0       = 0"
| "sumaImpares (Suc n) = (2 * (Suc n) - 1) + sumaImpares n"

value "sumaImpares 3" -- "= 9"

end
```

1.2. Ejercicios: Programación funcional con Isabelle/HOL

```
chapter {* T1R1: Programación funcional en Isabelle *}

theory T1R1
imports Main
begin

text {* ----------------------------------------------------------------
  Ejercicio 1. Definir, por recursión, la función
     longitud :: 'a list  nat
  tal que (longitud xs) es la longitud de la listas xs. Por ejemplo,
     longitud [4,2,5] = 3
  ---------------------------------------------------------------- *}

fun longitud :: "'a list  nat" where
  "longitud []     = 0"
| "longitud (x#xs) = 1 + longitud xs"

value "longitud [4,2,5]" -- "= 3"

text {* ----------------------------------------------------------------
  Ejercicio 2. Definir la función
     fun intercambia :: 'a × 'b  'b × 'a
  tal que (intercambia p) es el par obtenido intercambiando las
  componentes del par p. Por ejemplo,
     intercambia (u,v) = (v,u)
  ---------------------------------------------------------------- *}

fun intercambia :: "'a × 'b  'b × 'a" where
  "intercambia (x,y) = (y,x)"
```

```
value "intercambia (u,v)" -- "= (v,u)"

text {* ---------------------------------------------------------------
  Ejercicio 3. Definir, por recursión, la función
     inversa :: 'a list  'a list
  tal que (inversa xs) es la lista obtenida invirtiendo el orden de los
  elementos de xs. Por ejemplo,
     inversa [a,d,c] = [c,d,a]
  --------------------------------------------------------------- *}

fun inversa :: "'a list  'a list" where
  "inversa []     = []"
| "inversa (x#xs) = inversa xs @ [x]"

value "inversa [a,d,c]" -- "= [c,d,a]"

text {* ---------------------------------------------------------------
  Ejercicio 4. Definir la función
     repite :: nat  'a  'a list
  tal que (repite n x) es la lista formada por n copias del elemento
  x. Por ejemplo,
     repite 3 a = [a,a,a]
  --------------------------------------------------------------- *}

fun repite :: "nat  'a  'a list" where
  "repite 0 x       = []"
| "repite (Suc n) x = x # (repite n x)"

value "repite 3 a" -- "= [a,a,a]"

text {* ---------------------------------------------------------------
  Ejercicio 5. Definir la función
     conc :: 'a list  'a list  'a list
  tal que (conc xs ys) es la concatención de las listas xs e ys. Por
  ejemplo,
     conc [a,d] [b,d,a,c] = [a,d,b,d,a,c]
  --------------------------------------------------------------- *}

fun conc :: "'a list  'a list  'a list" where
```

```
  "conc []     ys = ys"
| "conc (x#xs) ys = x # (conc xs ys)"

value "conc [a,d] [b,d,a,c]" -- "= [a,d,b,d,a,c]"

text {* ---------------------------------------------------------------
  Ejercicio 6. Definir la función
     coge :: nat  'a list  'a list
  tal que (coge n xs) es la lista de los n primeros elementos de xs. Por
  ejemplo,
     coge 2 [a,c,d,b,e] = [a,c]
  ---------------------------------------------------------------- *}

fun coge :: "nat  'a list  'a list" where
  "coge n []           = []"
| "coge 0 xs           = []"
| "coge (Suc n) (x#xs) = x # (coge n xs)"

value "coge 2 [a,c,d,b,e]" -- "= [a,c]"

text {* ---------------------------------------------------------------
  Ejercicio 7. Definir la función
     elimina :: nat  'a list  'a list
  tal que (elimina n xs) es la lista obtenida eliminando los n primeros
  elementos de xs. Por ejemplo,
     elimina 2 [a,c,d,b,e] = [d,b,e]
  ---------------------------------------------------------------- *}

fun elimina :: "nat  'a list  'a list" where
  "elimina n []           = []"
| "elimina 0 xs           = xs"
| "elimina (Suc n) (x#xs) = elimina n xs"

value "elimina 2 [a,c,d,b,e]" -- "= [d,b,e]"

text {* ---------------------------------------------------------------
  Ejercicio 8. Definir la función
     esVacia :: 'a list  bool
  tal que (esVacia xs) se verifica si xs es la lista vacía. Por ejemplo,
     esVacia []  = True
```

```
     esVacia [1] = False
  ------------------------------------------------------------------ *}

fun esVacia :: "'a list  bool" where
  "esVacia []     = True"
| "esVacia (x#xs) = False"

value "esVacia []"  -- "= True"
value "esVacia [1]" -- "= False"

text {* ------------------------------------------------------------------
  Ejercicio 9. Definir la función
     inversaAc :: 'a list  'a list
  tal que (inversaAc xs) es a inversa de xs calculada usando
  acumuladores. Por ejemplo,
     inversaAc [a,c,b,e] = [e,b,c,a]
  ------------------------------------------------------------------ *}

fun inversaAcAux :: "'a list  'a list  'a list" where
  "inversaAcAux [] ys     = ys"
| "inversaAcAux (x#xs) ys = inversaAcAux xs (x#ys)"

fun inversaAc :: "'a list  'a list" where
  "inversaAc xs = inversaAcAux xs []"

value "inversaAc [a,c,b,e]" -- "= [e,b,c,a]"

text {* ------------------------------------------------------------------
  Ejercicio 10. Definir la función
     sum :: nat list  nat
  tal que (sum xs) es la suma de los elementos de xs. Por ejemplo,
     sum [3,2,5] = 10
  ------------------------------------------------------------------ *}

fun sum :: "nat list  nat" where
  "sum []     = 0"
| "sum (x#xs) = x + sum xs"

value "sum [3,2,5]" -- "= 10"
```

```
text {* ----------------------------------------------------------
  Ejercicio 11. Definir la función
     map :: ('a  'b)  'a list  'b list
  tal que (map f xs) es la lista obtenida aplicando la función f a los
  elementos de xs. Por ejemplo,
     map (x. 2*x) [3,2,5] = [6,4,10]
  ---------------------------------------------------------- *}

fun map :: "('a  'b)  'a list  'b list" where
  "map f []     = []"
| "map f (x#xs) = (f x) # map f xs"

value "map (x. 2*x) [3::nat,2,5]" -- "= [6,4,10]"

end
```

Capítulo 2

Razonamiento sobre programas

2.1. Tema: Razonamiento sobre programas

```
chapter {* Tema 2: Razonamiento sobre programas *}

theory T2
imports Main
begin

text {*
  En este tema se demuestra con Isabelle las propiedades de los
  programas funcionales como se expone en el tema 8 del curso
  "Informática" que puede leerse en
  http://www.cs.us.es/~jalonso/cursos/i1m/temas/tema-8t.pdf

  Todas las demostraciones se hacen automáticamente por simplificación e
  inducción.
*}

text {* ---------------------------------------------------------------
  Ejemplo 1. Definir, por recursión, la función
     longitud :: 'a list  nat
  tal que (longitud xs) es la longitud de la listas xs. Por ejemplo,
     longitud [4,2,5] = 3
  --------------------------------------------------------------- *}

fun longitud :: "'a list  nat" where
  "longitud []     = 0"
| "longitud (x#xs) = 1 + longitud xs"
```

```
value "longitud [4,2,5]" -- "= 3"

text {* ---------------------------------------------------------------
  Ejemplo 2. Demostrar que
     longitud [4,2,5] = 3
  ---------------------------------------------------------------- *}

lemma "longitud [4,2,5] = 3"
by simp

text {* ---------------------------------------------------------------
  Ejemplo 3. Definir la función
     fun intercambia :: 'a × 'b  'b × 'a
  tal que (intercambia p) es el par obtenido intercambiando las
  componentes del par p. Por ejemplo,
     intercambia (u,v) = (v,u)
  ---------------------------------------------------------------- *}

fun intercambia :: "'a × 'b  'b × 'a" where
  "intercambia (x,y) = (y,x)"

value "intercambia (u,v)" -- "= (v,u)"

text {* ---------------------------------------------------------------
  Ejemplo 4. Demostrar que
     intercambia (intercambia (x,y)) = (x,y)
  ---------------------------------------------------------------- *}

lemma "intercambia (intercambia (x,y)) = (x,y)"
by simp

text {* ---------------------------------------------------------------
  Ejemplo 5. Definir, por recursión, la función
     inversa :: 'a list  'a list
  tal que (inversa xs) es la lista obtenida invirtiendo el orden de los
  elementos de xs. Por ejemplo,
     inversa [a,d,c] = [c,d,a]
  ---------------------------------------------------------------- *}
```

```
fun inversa :: "'a list  'a list" where
  "inversa []     = []"
| "inversa (x#xs) = inversa xs @ [x]"

value "inversa [a,d,c]" -- "= [c,d,a]"

text {* ----------------------------------------------------------
  Ejemplo 6. Demostrar que
     inversa [x] = [x]
  ---------------------------------------------------------- *}

lemma "inversa [x] = [x]"
by simp

text {* ----------------------------------------------------------
  Ejemplo 7. Definir la función
     repite :: nat  'a  'a list
  tal que (repite n x) es la lista formada por n copias del elemento
  x. Por ejemplo,
     repite 3 a = [a,a,a]
  ---------------------------------------------------------- *}

fun repite :: "nat  'a  'a list" where
  "repite 0 x       = []"
| "repite (Suc n) x = x # (repite n x)"

value "repite 3 a" -- "= [a,a,a]"

text {* ----------------------------------------------------------
  Ejemplo 8. Demostrar que
     longitud (repite n x) = n
  ---------------------------------------------------------- *}

lemma "longitud (repite n x) = n"
by (induct n) auto

text {* ----------------------------------------------------------
  Ejemplo 9. Definir la función
     conc :: 'a list  'a list  'a list
  tal que (conc xs ys) es la concatención de las listas xs e ys. Por
```

```
  ejemplo,
     conc [a,d] [b,d,a,c] = [a,d,b,d,a,c]
  ------------------------------------------------------------------ *}

fun conc :: "'a list  'a list  'a list" where
  "conc []     ys = ys"
| "conc (x#xs) ys = x # (conc xs ys)"

value "conc [a,d] [b,d,a,c]" -- "= [a,d,b,d,a,c]"

text {* ------------------------------------------------------------------
  Ejemplo 10. Demostrar que
     conc xs (conc ys zs) = (conc xs ys) zs
  ------------------------------------------------------------------ *}

lemma "conc xs (conc ys zs) = conc (conc xs ys) zs"
by (induct xs) auto

text {* ------------------------------------------------------------------
  Ejemplo 11. Refutar que
     conc xs ys = conc ys xs
  ------------------------------------------------------------------ *}

lemma "conc xs ys = conc ys xs"
quickcheck
oops

text {* Encuentra el contraejemplo,
  xs = [a\<^isub>2]
  ys = [a\<^isub>1] *}

text {* ------------------------------------------------------------------
  Ejemplo 12. Demostrar que
     conc xs [] = xs
  ------------------------------------------------------------------ *}

lemma "conc xs [] = xs"
by (induct xs) auto

text {* ------------------------------------------------------------------
```

```
  Ejemplo 13. Demostrar que
     longitud (conc xs ys) = longitud xs + longitud ys
  ------------------------------------------------------------------ *}

lemma "longitud (conc xs ys) = longitud xs + longitud ys"
by (induct xs) auto

text {* ------------------------------------------------------------------
  Ejemplo 14. Definir la función
     coge :: nat  'a list  'a list
  tal que (coge n xs) es la lista de los n primeros elementos de xs. Por
  ejemplo,
     coge 2 [a,c,d,b,e] = [a,c]
  ------------------------------------------------------------------ *}

fun coge :: "nat  'a list  'a list" where
  "coge n []           = []"
| "coge 0 xs           = []"
| "coge (Suc n) (x#xs) = x # (coge n xs)"

value "coge 2 [a,c,d,b,e]" -- "= [a,c]"

text {* ------------------------------------------------------------------
  Ejemplo 15. Definir la función
     elimina :: nat  'a list  'a list
  tal que (elimina n xs) es la lista obtenida eliminando los n primeros
  elementos de xs. Por ejemplo,
     elimina 2 [a,c,d,b,e] = [d,b,e]
  ------------------------------------------------------------------ *}

fun elimina :: "nat  'a list  'a list" where
  "elimina n []           = []"
| "elimina 0 xs           = xs"
| "elimina (Suc n) (x#xs) = elimina n xs"

value "elimina 2 [a,c,d,b,e]" -- "= [d,b,e]"

text {* ------------------------------------------------------------------
  Ejemplo 16. Demostrar que
     conc (coge n xs) (elimina n xs) = xs
```

```
  ------------------------------------------------------------------ *}

lemma "conc (coge n xs) (elimina n xs) = xs"
by (induct rule: coge.induct) auto

text {* coge.induct es el esquema de inducción asociado a la definición
  de la función coge.
     n. P n [];
      x xs. P 0 (x#xs);
      n x xs. P n xs  P (Suc n) (x#xs)
      P n x
  Puede verse usando "thm coge.induct". *}

text {* ------------------------------------------------------------------
  Ejemplo 17. Definir la función
     esVacia :: 'a list  bool
  tal que (esVacia xs) se verifica si xs es la lista vacía. Por ejemplo,
     esVacia []  = True
     esVacia [1] = False
  ------------------------------------------------------------------ *}

fun esVacia :: "'a list  bool" where
  "esVacia []     = True"
| "esVacia (x#xs) = False"

value "esVacia []"  -- "= True"
value "esVacia [1]" -- "= False"

text {* ------------------------------------------------------------------
  Ejemplo 18. Demostrar que
     esVacia xs = esVacia (conc xs xs)
  ------------------------------------------------------------------ *}

lemma "esVacia xs = esVacia (conc xs xs)"
by (induct xs) auto

text {* ------------------------------------------------------------------
  Ejemplo 19. Definir la función
     inversaAc :: 'a list  'a list
  tal que (inversaAc xs) es a inversa de xs calculada usando
```

```
  acumuladores. Por ejemplo,
     inversaAc [a,c,b,e] = [e,b,c,a]
  ---------------------------------------------------------------- *}

fun inversaAcAux :: "'a list  'a list  'a list" where
  "inversaAcAux [] ys     = ys"
| "inversaAcAux (x#xs) ys = inversaAcAux xs (x#ys)"

fun inversaAc :: "'a list  'a list" where
  "inversaAc xs = inversaAcAux xs []"

value "inversaAc [a,c,b,e]" -- "= [e,b,c,a]"

text {* ----------------------------------------------------------------
  Ejemplo 20. Demostrar que
     inversaAcAux xs ys = (inversa xs)@ys
  ---------------------------------------------------------------- *}

lemma inversaAcAux_es_inversa:
  "inversaAcAux xs ys = (inversa xs)@ys"
by (induct xs arbitrary: ys) auto

text {* ----------------------------------------------------------------
  Ejemplo 21. Demostrar que
     inversaAc xs = inversa xs
  ---------------------------------------------------------------- *}

corollary "inversaAc xs = inversa xs"
by (simp add: inversaAcAux_es_inversa)

text {* ----------------------------------------------------------------
  Ejemplo 22. Definir la función
     sum :: nat list  nat
  tal que (sum xs) es la suma de los elementos de xs. Por ejemplo,
     sum [3,2,5] = 10
  ---------------------------------------------------------------- *}

fun sum :: "nat list  nat" where
  "sum []     = 0"
| "sum (x#xs) = x + sum xs"
```

```
value "sum [3,2,5]" -- "= 10"

text {* ---------------------------------------------------------------
  Ejemplo 23. Definir la función
     map :: ('a  'b)  'a list  'b list
  tal que (map f xs) es la lista obtenida aplicando la función f a los
  elementos de xs. Por ejemplo,
     map (x. 2*x) [3,2,5] = [6,4,10]
  ---------------------------------------------------------------- *}

fun map :: "('a  'b)  'a list  'b list" where
  "map f []     = []"
| "map f (x#xs) = (f x) # map f xs"

value "map (x. 2*x) [3::nat,2,5]" -- "= [6,4,10]"

text {* ---------------------------------------------------------------
  Ejemplo 24. Demostrar que
     sum (map (x. 2*x) xs) = 2 * (sum xs)
  ---------------------------------------------------------------- *}

lemma "sum (map (x. 2*x) xs) = 2 * (sum xs)"
by (induct xs) auto

text {* ---------------------------------------------------------------
  Ejemplo 25. Demostrar que
     longitud (map f xs) = longitud xs
  ---------------------------------------------------------------- *}

lemma "longitud (map f xs) = longitud xs"
by (induct xs) auto

section {* Referencias *}

text {*
  ů J.A. Alonso. "Razonamiento sobre programas" http://goo.gl/R0603
  ů G. Hutton. "Programming in Haskell". Cap. 13 "Reasoning about
    programms".
  ů S. Thompson. "Haskell: the Craft of Functional Programming, 3rd
```

```
    Edition. Cap. 8 "Reasoning about programms".
  ů L. Paulson. "ML for the Working Programmer, 2nd Edition". Cap. 6.
    "Reasoning about functional programs".
*}

end
```

2.2. Ejercicios: Razonamiento sobre programas

```
chapter {* T2R1: Razonamiento sobre programas *}

theory T2R1
imports Main
begin

text {* ---------------------------------------------------------------
  Ejercicio 1. Definir la función
     sumaImpares :: nat  nat
  tal que (sumaImpares n) es la suma de los n primeros números
  impares. Por ejemplo,
     sumaImpares 5  =  25
  ---------------------------------------------------------------- *}

fun sumaImpares :: "nat  nat" where
  "sumaImpares 0 = 0"
| "sumaImpares (Suc n) = sumaImpares n + (2*n+1)"

value "sumaImpares 5" -- "= 25"

text {* ---------------------------------------------------------------
  Ejercicio 2. Demostrar que
     sumaImpares n = n*n
  ---------------------------------------------------------------- *}

lemma "sumaImpares n = n*n"
by (induct n) auto

text {* ---------------------------------------------------------------
  Ejercicio 3. Definir la función
     sumaPotenciasDeDosMasUno :: nat  nat
```

```
  tal que
     (sumaPotenciasDeDosMasUno n) = 1 + 2^0 + 2^1 + 2^2 + ... + 2^n.
  Por ejemplo,
     sumaPotenciasDeDosMasUno 3  =  16
  ------------------------------------------------------------------ *}

fun sumaPotenciasDeDosMasUno :: "nat  nat" where
  "sumaPotenciasDeDosMasUno 0 = 2"
| "sumaPotenciasDeDosMasUno (Suc n) =
      sumaPotenciasDeDosMasUno n + 2^(n+1)"

value "sumaPotenciasDeDosMasUno 3" -- "= 16"

text {* ------------------------------------------------------------------
  Ejercicio 4. Demostrar que
     sumaPotenciasDeDosMasUno n = 2^(n+1)
  ------------------------------------------------------------------ *}

lemma "sumaPotenciasDeDosMasUno n = 2^(n+1)"
by (induct n) auto

text {* ------------------------------------------------------------------
  Ejercicio 5. Definir la función
     copia :: nat  'a  'a list
  tal que (copia n x) es la lista formado por n copias del elemento
  x. Por ejemplo,
     copia 3 x = [x,x,x]
  ------------------------------------------------------------------ *}

fun copia :: "nat  'a  'a list" where
  "copia 0 x       = []"
| "copia (Suc n) x = x # copia n x"

value "copia 3 x" -- "= [x,x,x]"

text {* ------------------------------------------------------------------
  Ejercicio 6. Definir la función
     todos :: ('a  bool)  'a list  bool
  tal que (todos p xs) se verifica si todos los elementos de xs cumplen
  la propiedad p. Por ejemplo,
```

```
     todos (x. x>(1::nat)) [2,6,4] = True
     todos (x. x>(2::nat)) [2,6,4] = False
  Nota: La conjunción se representa por
  ------------------------------------------------------------ *}

fun todos :: "('a  bool)  'a list  bool" where
  "todos p []     = True"
| "todos p (x#xs) = (p x  todos p xs)"

value "todos (x. x>(1::nat)) [2,6,4]" -- "= True"
value "todos (x. x>(2::nat)) [2,6,4]" -- "= False"

text {* ------------------------------------------------------------
  Ejercicio 7. Demostrar que todos los elementos de (copia n x) son
  iguales a x.
  ------------------------------------------------------------ *}

lemma "todos (y. y=x) (copia n x)"
by (induct n) auto

text {* ------------------------------------------------------------
  Ejercicio 8. Definir la función
    factR :: nat  nat
  tal que (factR n) es el factorial de n. Por ejemplo,
    factR 4 = 24
  ------------------------------------------------------------ *}

fun factR :: "nat  nat" where
  "factR 0       = 1"
| "factR (Suc n) = Suc n * factR n"

value "factR 4" -- "= 24"

text {* ------------------------------------------------------------
  Ejercicio 9. Se considera la siguiente definición iterativa de la
  función factorial
     factI :: "nat  nat" where
     factI n = factI' n 1

     factI' :: nat  nat  nat" where
```

```
     factI' 0       x = x
     factI' (Suc n) x = factI' n (Suc n)*x
  Demostrar que, para todo n y todo x, se tiene
     factI' n x = x * factR n
  ---------------------------------------------------------------------- *}

fun factI' :: "nat  nat  nat" where
  "factI' 0       x = x"
| "factI' (Suc n) x = factI' n (Suc n)*x"

fun factI :: "nat  nat" where
  "factI n = factI' n 1"

value "factI 4" -- "= 24"

lemma fact: "factI' n x = x * factR n"
by (induct n arbitrary: x) auto

text {* ----------------------------------------------------------------
  Ejercicio 10. Demostrar que
     factI n = factR n
  ---------------------------------------------------------------------- *}

corollary "factI n = factR n"
by (simp add: fact)

text {* ----------------------------------------------------------------
  Ejercicio 11. Definir, recursivamente y sin usar (@), la función
     amplia :: 'a list  'a  'a list
  tal que (amplia xs y) es la lista obtenida añadiendo el elemento y al
  final de la lista xs. Por ejemplo,
     amplia [d,a] t = [d,a,t]
  ---------------------------------------------------------------------- *}

fun amplia :: "'a list  'a  'a list" where
  "amplia []     y = [y]"
| "amplia (x#xs) y = x # amplia xs y"

value "amplia [d,a] t" -- "= [d,a,t]"
```

```
text {* ----------------------------------------------------------------
  Ejercicio 12. Demostrar que
     amplia xs y = xs @ [y]
  ---------------------------------------------------------------- *}

lemma "amplia xs y = xs @ [y]"
by (induct xs) auto

end
```

Capítulo 3

Deducción natural en lógica proposicional

3.1. Tema: Deducción natural en lógica proposicional

```
chapter {* Tema 3: Deducción natural proposicional con Isabelle/HOL *}

theory T3
imports Main
begin

text {*
  En esta sección se presentan los ejemplos del tema de deducción natural
  proposicional siguiendo la presentación de Huth y Ryan en su libro
  "Logic in Computer Science" http://goo.gl/qsVpY y, más concretamente,
  a la forma como se explica en la asignatura de "Lógica informática" (LI)
  http://goo.gl/AwDiv

  La página al lado de cada ejemplo indica la página de las transparencias
  de LI donde se encuentra la demostración. *}

subsection {* Reglas de la conjunción *}

text {*
  Ejemplo 1 (p. 4). Demostrar que
     p ∧ q, r ∧ q ⊢ r.
  *}

-- "La demostración detallada es"
```

```
lemma ejemplo_1_1:
  assumes 1: "p  q" and
          2: "r"
  shows "q  r"
proof -
  have 3: "q" using 1 by (rule conjunct2)
  show 4: "q  r" using 3 2 by (rule conjI)
qed

text {*
  Notas sobre el lenguaje: En la demostración anterior se ha usado
  ů "assumes" para indicar las hipótesis,
  ů "and" para separar las hipótesis,
  ů "shows" para indicar la conclusión,
  ů "proof" para iniciar la prueba,
  ů "qed" para terminar la pruebas,
  ů "-" (después de "proof") para no usar el método por defecto,
  ů "have" para establecer un paso,
  ů "using" para usar hechos en un paso,
  ů "by (rule ..)" para indicar la regla con la que se peueba un hecho,
  ů "show" para establecer la conclusión.

  Notas sobre la lógica: Las reglas de la conjunción son
  ů conjI:      P; Q  P  Q
  ů conjunct1:  P  Q  P
  ů conjunct2:  P  Q  Q
*}

text {* Se pueden dejar implícitas las reglas como sigue *}

lemma ejemplo_1_2:
  assumes 1: "p  q" and
          2: "r"
  shows "q  r"
proof -
  have 3: "q" using 1 ..
  show 4: "q  r" using 3 2 ..
qed

text {*
```

```
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ů ".." para indicar que se prueba por la regla correspondiente. *}

text {* Se pueden eliminar las etiquetas como sigue *}

lemma ejemplo_1_3:
  assumes "p  q"
          "r"
  shows   "q  r"
proof -
  have "q" using assms(1) ..
  thus "q  r" using assms(2) ..
qed

text {*
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ů "assms(n)" para indicar la hipótesis n y
  ů "thus" para demostrar la conclusión usando el hecho anterior.
  Además, no es necesario usar and entre las hipótesis. *}

text {* Se puede automatizar la demostración como sigue *}

lemma ejemplo_1_4:
  assumes "p  q"
          "r"
  shows   "q  r"
using assms
by auto

text {*
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ů "assms" para indicar las hipótesis y
  ů "by auto" para demostrar la conclusión automáticamente. *}

text {* Se puede automatizar totalmente la demostración como sigue *}

lemma ejemplo_1_5:
  "p  q; r  q  r"
by auto
```

```
text {*
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ů " ... " para representar las hipótesis,
  ů ";" para separar las hipótesis y
  ů "" para separar las hipótesis de la conclusión. *}

text {* Se puede hacer la demostración por razonamiento hacia atrás,
  como sigue *}

lemma ejemplo_1_6:
  assumes "p  q"
      and "r"
  shows   "q  r"
proof (rule conjI)
  show "q" using assms(1) by (rule conjunct2)
next
  show "r" using assms(2) by this
qed

text {*
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ů "proof (rule r)" para indicar que se hará la demostración con la
    regla r,
  ů "next" para indicar el comienzo de la prueba del siguiente
    subobjetivo,
  ů "this" para indicar el hecho actual. *}

text {* Se pueden dejar implícitas las reglas como sigue *}

lemma ejemplo_1_7:
  assumes "p  q"
          "r"
  shows   "q  r"
proof
  show "q" using assms(1) ..
next
  show "r" using assms(2) .
qed

text {*
```

```
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ů "." para indicar por el hecho actual. *}

subsection {* Reglas de la doble negación *}

text {*
  La regla de eliminación de la doble negación es
  ů notnotD: ňň P  P

  Para ajustarnos al tema de LI vamos a introducir la siguiente regla de
  introducción de la doble negación
  ů notnotI: P  ňň P
  aunque, de momento, no detallamos su demostración.
*}

lemma notnotI [intro!]: "P  ňň P"
by auto

text {*
  Ejemplo 2. (p. 5)
       p, ňň(q  r)  ňňp  r
*}

-- "La demostración detallada es"
lemma ejemplo_2_1:
  assumes 1: "p" and
          2: "ňň(q  r)"
  shows      "ňňp  r"
proof -
  have 3: "ňňp" using 1 by (rule notnotI)
  have 4: "q  r" using 2 by (rule notnotD)
  have 5: "r" using 4 by (rule conjunct2)
  show 6: "ňňp  r" using 3 5 by (rule conjI)
qed

-- "La demostración estructurada es"
lemma ejemplo_2_2:
  assumes "p"
          "ňň(q  r)"
  shows   "ňňp  r"
```

```
proof -
  have  "¬¬p" using assms(1) ..
  have  "q  r" using assms(2) by (rule notnotD)
  hence "r" ..
  with ‘¬¬p‘ show  "¬¬p  r" ..
qed

text {*
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ů "hence" para indicar que se tiene por el hecho anterior,
  ů ‘...‘ para referenciar un hecho y
  ů "with P show Q" para indicar que con el hecho anterior junto con el
    hecho P se demuestra Q. *}

-- "La demostración automática es"
lemma ejemplo_2_3:
  assumes "p"
          "¬¬(q  r)"
  shows   "¬¬p  r"
using assms
by auto

text {* Se puede demostrar hacia atrás *}

lemma ejemplo_2_4:
  assumes "p"
          "¬¬(q  r)"
  shows   "¬¬p  r"
proof  (rule conjI)
  show "¬¬p" using assms(1) by (rule notnotI)
next
  have "q  r" using assms(2) by (rule notnotD)
  thus "r" by (rule conjunct2)
qed

text {* Se puede eliminar las reglas en la demostración anterior, como
  sigue: *}

lemma ejemplo_2_5:
  assumes "p"
```

```
          "¬¬(q ∧ r)"
  shows   "¬¬p ∧ r"
proof
  show "¬¬p" using assms(1) ..
next
  have "q ∧ r" using assms(2) by (rule notnotD)
  thus "r" ..
qed

subsection {* Regla de eliminación del condicional *}

text {*
  La regla de eliminación del condicional es la regla del modus ponens
  ú mp: P → Q; P ⊢ Q
*}

text {*
  Ejemplo 3. (p. 6) Demostrar que
     ¬p → q, ¬p → q → r ⊢ ¬p → r → ¬p
*}

-- "La demostración detallada es"
lemma ejemplo_3_1:
  assumes 1: "¬p → q" and
          2: "¬p → q → r → ¬p"
  shows      "r → ¬p"
proof -
  show "r → ¬p" using 2 1 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejemplo_3_2:
  assumes "¬p → q"
          "¬p → q → r → ¬p"
  shows   "r → ¬p"
proof -
  show "r → ¬p" using assms(2,1) ..
qed

-- "La demostración automática es"
```

```
lemma ejemplo_3_3:
  assumes "¬p ∨ q"
          "¬p → q → r ∧ ¬p"
  shows   "r ∧ ¬p"
using assms
by auto

text {*
  Ejemplo 4 (p. 6) Demostrar que
     p, p → q, p ∧ (q → r) ⊢ r
 *}

-- "La demostración detallada es"
lemma ejemplo_4_1:
  assumes 1: "p" and
          2: "p → q" and
          3: "p → (q → r)"
  shows "r"
proof -
  have 4: "q" using 2 1 by (rule mp)
  have 5: "q → r" using 3 1 by (rule mp)
  show 6: "r" using 5 4 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejemplo_4_2:
  assumes "p"
          "p → q"
          "p → (q → r)"
  shows "r"
proof -
  have "q" using assms(2,1) ..
  have "q → r" using assms(3,1) ..
  thus "r" using `q` ..
qed

-- "La demostración automática es"
lemma ejemplo_4_3:
  "p; p → q; p → (q → r) ⊢ r"
by auto
```

```
subsection {* Regla derivada del modus tollens *}

text {*
  Para ajustarnos al tema de LI vamos a introducir la regla del modus
  tollens
  ů mt: F  G; ňG  ňF
  aunque, de momento, sin detallar su demostración.
*}

lemma mt: "F  G; ňG  ňF"
by auto

text {*
  Ejemplo 5 (p. 7). Demostrar
     p  (q  r), p, ňr  ňq
 *}

-- "La demostración detallada es"
lemma ejemplo_5_1:
  assumes 1: "p  (q  r)" and
          2: "p" and
          3: "ňr"
  shows "ňq"
proof -
  have 4: "q  r" using 1 2 by (rule mp)
  show "ňq" using 4 3 by (rule mt)
qed

-- "La demostración estructurada es"
lemma ejemplo_5_2:
  assumes "p  (q  r)"
          "p"
          "ňr"
  shows   "ňq"
proof -
  have "q  r" using assms(1,2) ..
  thus "ňq" using assms(3) by (rule mt)
qed
```

```
-- "La demostración automática es"
lemma ejemplo_5_3:
  assumes "p  (q  r)"
          "p"
          "¬r"
  shows   "¬q"
using assms
by auto

text {*
  Ejemplo 6. (p. 7) Demostrar
     ¬p  q, ¬q  p
*}

-- "La demostración detallada es"
lemma ejemplo_6_1:
  assumes 1: "¬p  q" and
          2: "¬q"
  shows "p"
proof -
  have 3: "¬¬p" using 1 2 by (rule mt)
  show "p" using 3 by (rule notnotD)
qed

-- "La demostración estructurada es"
lemma ejemplo_6_2:
  assumes "¬p  q"
          "¬q"
  shows   "p"
proof -
  have "¬¬p" using assms(1,2) by (rule mt)
  thus "p" by (rule notnotD)
qed

-- "La demostración automática es"
lemma ejemplo_6_3:
  "¬p  q; ¬q  p"
by auto

text {*
```

```
  Ejemplo 7. (p. 7) Demostrar
     p  ¬¬q, q  ¬p
  *}

-- "La demostración detallada es"
lemma ejemplo_7_1:
  assumes 1: "p  ¬¬q" and
          2: "q"
  shows "¬p"
proof -
  have 3: "¬¬q" using 2 by (rule notnotI)
  show "¬p" using 1 3 by (rule mt)
qed

-- "La demostración detallada es"
lemma ejemplo_7_2:
  assumes "p  ¬¬q"
          "q"
  shows   "¬p"
proof -
  have "¬¬q" using assms(2) by (rule notnotI)
  with assms(1) show "¬p" by (rule mt)
qed

-- "La demostración estructurada es"
lemma ejemplo_7_3:
  "p  ¬¬q; q  ¬p"
by auto

subsection {* Regla de introducción del condicional *}

text {*
  La regla de introducción del condicional es
  ⇒I: (P  Q)  P  Q
*}

text {*
  Ejemplo 8. (p. 8) Demostrar
     p  q  ¬¬q  ¬p
*}
```

```
-- "La demostración detallada es"
lemma ejemplo_8_1:
  assumes 1: "p  q"
  shows "¬q  ¬p"
proof -
  { assume 2: "¬q"
    have "¬p" using 1 2 by (rule mt) }
  thus "¬q  ¬p" by (rule impI)
qed

text {*
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ú "{ ... }" para representar una caja. *}

-- "La demostración estructurada es"
lemma ejemplo_8_2:
  assumes "p  q"
  shows "¬q  ¬p"
proof
  assume "¬q"
  with assms show "¬p" by (rule mt)
qed

-- "La demostración automática es"
lemma ejemplo_8_3:
  assumes "p  q"
  shows "¬q  ¬p"
using assms
by auto

text {*
  Ejemplo 9. (p. 9) Demostrar
     ¬q  ¬p  p  ¬¬q
*}

-- "La demostración detallada es"
lemma ejemplo_9_1:
  assumes 1: "¬q  ¬p"
  shows "p  ¬¬q"
```

```
proof -
  { assume 2: "p"
    have 3: "¬¬p" using 2 by (rule notnotI)
    have "¬¬q" using 1 3 by (rule mt) }
  thus "p → ¬¬q" by (rule impI)
qed

-- "La demostración estructurada es"
lemma ejemplo_9_2:
  assumes "¬q → ¬p"
  shows   "p → ¬¬q"
proof
  assume "p"
  hence "¬¬p" by (rule notnotI)
  with assms show "¬¬q" by (rule mt)
qed

-- "La demostración automática es"
lemma ejemplo_9_3:
  assumes "¬q → ¬p"
  shows "p → ¬¬q"
using assms
by auto

text {*
  Ejemplo 10 (p. 9). Demostrar
      p ⊢ p
*}

-- "La demostración detallada es"
lemma ejemplo_10_1:
  "p ⊢ p"
proof -
  { assume 1: "p"
    have "p" using 1 by this }
  thus "p → p" by (rule impI)
qed

-- "La demostración estructurada es"
lemma ejemplo_10_2:
```

```
  "p  p"
proof (rule impI)
qed

-- "La demostración automática es"
lemma ejemplo_10_3:
  "p  p"
by auto

text {*
  Ejemplo 11 (p. 10) Demostrar
      (q  r)  ((¬q  ¬p)  (p  r))
 *}

-- "La demostración detallada es"
lemma ejemplo_11_1:
  "(q  r)  ((¬q  ¬p)  (p  r))"
proof -
  { assume 1: "q  r"
    { assume 2: "¬q  ¬p"
      { assume 3: "p"
        have 4: "¬¬p" using 3 by (rule notnotI)
        have 5: "¬¬q" using 2 4 by (rule mt)
        have 6: "q" using 5 by (rule notnotD)
        have "r" using 1 6 by (rule mp) }
      hence "p  r" by (rule impI) }
    hence "(¬q  ¬p)  p  r" by (rule impI) }
  thus "(q  r)  ((¬q  ¬p)  p  r)" by (rule impI)
qed

-- "La demostración hacia atrás es"
lemma ejemplo_11_2:
  "(q  r)  ((¬q  ¬p)  (p  r))"
proof (rule impI)
  assume 1: "q  r"
  show "(¬q  ¬p)  (p  r)"
  proof (rule impI)
    assume 2: "¬q  ¬p"
    show "p  r"
    proof (rule impI)
```

```
      assume 3: "p"
      have 4: "¬¬p" using 3 by (rule notnotI)
      have 5: "¬¬q" using 2 4 by (rule mt)
      have 6: "q" using 5 by (rule notnotD)
      show "r" using 1 6 by (rule mp)
    qed
  qed
qed

-- "La demostración hacia atrás con reglas implícitas es"
lemma ejemplo_11_3:
  "(q ⟶ r) ⟶ ((¬q ⟶ ¬p) ⟶ (p ⟶ r))"
proof
  assume 1: "q ⟶ r"
  show "(¬q ⟶ ¬p) ⟶ (p ⟶ r)"
  proof
    assume 2: "¬q ⟶ ¬p"
    show "p ⟶ r"
    proof
      assume 3: "p"
      have 4: "¬¬p" using 3 ..
      have 5: "¬¬q" using 2 4 by (rule mt)
      have 6: "q" using 5 by (rule notnotD)
      show "r" using 1 6 ..
    qed
  qed
qed

-- "La demostración sin etiquetas es"
lemma ejemplo_11_4:
  "(q ⟶ r) ⟶ ((¬q ⟶ ¬p) ⟶ (p ⟶ r))"
proof
  assume "q ⟶ r"
  show "(¬q ⟶ ¬p) ⟶ (p ⟶ r)"
  proof
    assume "¬q ⟶ ¬p"
    show "p ⟶ r"
    proof
      assume "p"
      hence "¬¬p" ..
```

```
      with ‘¬q  ¬p‘ have "¬¬q" by (rule mt)
      hence "q" by (rule notnotD)
      with ‘q  r‘ show "r" ..
    qed
  qed
qed

-- "La demostración automática es"
lemma ejemplo_11_5:
  "(q  r)  ((¬q  ¬p)  (p  r))"
by auto

subsection {* Reglas de la disyunción *}

text {*
  Las reglas de la introducción de la disyunción son
  • disjI1: P  P  Q
  • disjI2: Q  P  Q
  La regla de elimación de la disyunción es
  • disjE:  P  Q; P  R; Q  R  R
*}

text {*
  Ejemplo 12 (p. 11). Demostrar
     p  q  q  p
*}

-- "La demostración detallada es"
lemma ejemplo_12_1:
  assumes "p  q"
  shows "q  p"
proof -
  have "p  q" using assms by this
  moreover
  { assume 2: "p"
    have "q  p" using 2 by (rule disjI2) }
  moreover
  { assume 3: "q"
    have "q  p" using 3 by (rule disjI1) }
  ultimately show "q  p" by (rule disjE)
```

```
qed

text {*
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ů "moreover" para separar los bloques y
  ů "ultimately" para unir los resultados de los bloques. *}

-- "La demostración detallada con reglas implícitas es"
lemma ejemplo_12_2:
  assumes "p  q"
  shows "q  p"
proof -
  note ‘p  q‘
  moreover
  { assume "p"
    hence "q  p" .. }
  moreover
  { assume "q"
    hence "q  p" .. }
  ultimately show "q  p" ..
qed

text {*
  Nota sobre el lenguaje: En la demostración anterior se ha usado
  ů "note" para copiar un hecho. *}

-- "La demostración hacia atrás es"
lemma ejemplo_12_3:
  assumes 1: "p  q"
  shows "q  p"
using 1
proof (rule disjE)
  { assume 2: "p"
    show "q  p" using 2 by (rule disjI2) }
next
  { assume 3: "q"
    show "q  p" using 3 by (rule disjI1) }
qed

-- "La demostración hacia atrás con reglas implícitas es"
```

```
lemma ejemplo_12_4:
  assumes "p  q"
  shows "q  p"
using assms
proof
  { assume  "p"
    thus "q  p" .. }
next
  { assume "q"
    thus "q  p" .. }
qed

-- "La demostración automática es"
lemma ejemplo_12_5:
  assumes "p  q"
  shows "q  p"
using assms
by auto

text {*
  Ejemplo 13. (p. 12) Demostrar
     q  r  p  q  p  r
*}

-- "La demostración detallada es"
lemma ejemplo_13_1:
  assumes 1: "q  r"
  shows "p  q  p  r"
proof (rule impI)
  assume 2: "p  q"
  thus "p  r"
  proof (rule disjE)
    { assume 3: "p"
      show "p  r" using 3 by (rule disjI1) }
  next
    { assume 4: "q"
      have 5: "r" using 1 4 by (rule mp)
      show "p  r" using 5 by (rule disjI2) }
  qed
qed
```

```
-- "La demostración estructurada es"
lemma ejemplo_13_2:
  assumes "q  r"
  shows "p  q  p  r"
proof
  assume "p  q"
  thus "p  r"
  proof
    { assume "p"
      thus "p  r" .. }
  next
    { assume "q"
      have "r" using assms ‘q‘ ..
      thus "p  r" .. }
  qed
qed

-- "La demostración automática es"
lemma ejemplo_13_3:
  assumes "q  r"
  shows "p  q  p  r"
using assms
by auto

subsection {* Regla de copia *}

text {*
  Ejemplo 14 (p. 13). Demostrar
      p  (q  p)
*}

-- "La demostración detallada es"
lemma ejemplo_14_1:
  "p  (q  p)"
proof (rule impI)
  assume 1: "p"
  show "q  p"
  proof (rule impI)
    assume "q"
```

```
    show "p" using 1 by this
  qed
qed

-- "La demostración estructurada es"
lemma ejemplo_14_2:
  "p  (q  p)"
proof
  assume "p"
  thus "q  p" ..
qed

-- "La demostración automática es"
lemma ejemplo_14_3:
  "p  (q  p)"
by auto

subsection {* Reglas de la negación *}

text {*
  La regla de eliminación de lo falso es
  ů FalseE: False  P
  La regla de eliminación de la negación es
  ů notE: ňP; P  R
  La regla de introducción de la negación es
  ů notI: (P  False)  ňP
*}

text {*
  Ejemplo 15 (p. 15). Demostrar
     ňp  q  p  q
*}

-- "La demostración detallada es"
lemma ejemplo_15_1:
  assumes 1: "ňp  q"
  shows "p  q"
proof (rule impI)
  assume 2: "p"
  note 1
```

```
  thus "q"
  proof (rule disjE)
    { assume 3: "¬p"
      show "q" using 3 2 by (rule notE) }
  next
    { assume 4: "q"
      show "q" using 4 by this}
  qed
qed

-- "La demostración estructurada es"
lemma ejemplo_15_2:
  assumes "¬p ∨ q"
  shows "p → q"
proof
  assume "p"
  note ‹¬p ∨ q›
  thus "q"
  proof
    { assume "¬p"
      thus "q" using ‹p› .. }
  next
    { assume "q"
      thus "q" . }
  qed
qed

-- "La demostración automática es"
lemma ejemplo_15_3:
  assumes "¬p ∨ q"
  shows "p → q"
using assms
by auto

text {*
  Ejemplo 16 (p. 16). Demostrar
     p → q, p → ¬q ⊢ ¬p
*}

-- "La demostración detallada es"
```

```
lemma ejemplo_16_1:
  assumes 1: "p  q" and
          2: "p  ňq"
  shows "ňp"
proof (rule notI)
  assume 3: "p"
  have 4: "q" using 1 3 by (rule mp)
  have 5: "ňq" using 2 3 by (rule mp)
  show False using 5 4 by (rule notE)
qed

-- "La demostración estructurada es"
lemma ejemplo_16_2:
  assumes "p  q"
          "p  ňq"
  shows "ňp"
proof
  assume "p"
  have "q" using assms(1) ‘p‘ ..
  have "ňq" using assms(2) ‘p‘ ..
  thus False using ‘q‘ ..
qed

-- "La demostración automática es"
lemma ejemplo_16_3:
  assumes "p  q"
          "p  ňq"
  shows "ňp"
using assms
by auto

subsection {* Reglas del bicondicional *}

text {*
  La regla de introducción del bicondicional es
  ů iffI: P  Q; Q  P  P  Q
  Las reglas de eliminación del bicondicional son
  ů iffD1: Q  P; Q  P
  ů iffD2: P  Q; Q  P
*}
```

```
text {*
  Ejemplo 17 (p. 17) Demostrar
*}

-- "La demostración detallada es"
lemma ejemplo_17_1:
  "(p  q)  (q  p)"
proof (rule iffI)
  { assume 1: "p  q"
    have 2: "p" using 1 by (rule conjunct1)
    have 3: "q" using 1 by (rule conjunct2)
    show "q  p" using 3 2 by (rule conjI) }
next
  { assume 4: "q  p"
    have 5: "q" using 4 by (rule conjunct1)
    have 6: "p" using 4 by (rule conjunct2)
    show "p  q" using 6 5 by (rule conjI) }
qed

-- "La demostración estructurada es"
lemma ejemplo_17_2:
  "(p  q)  (q  p)"
proof
  { assume 1: "p  q"
    have "p" using 1 ..
    have "q" using 1 ..
    show "q  p" using `q` `p` .. }
next
  { assume 2: "q  p"
    have "q" using 2 ..
    have "p" using 2 ..
    show "p  q" using `p` `q`  .. }
qed

-- "La demostración automática es"
lemma ejemplo_17_3:
  "(p  q)  (q  p)"
by auto
```

```
text {*
  Ejemplo 18 (p. 18). Demostrar
     p ∧ q, p → q ⊢ p ∧ q
*}

-- "La demostración detallada es"
lemma ejemplo_18_1:
  assumes 1: "p ∧ q" and
          2: "p → q"
  shows "p ∧ q"
using 2
proof (rule disjE)
  { assume 3: "p"
    have 4: "q" using 1 3 by (rule iffD1)
    show "p ∧ q" using 3 4 by (rule conjI) }
next
  { assume 5: "q"
    have 6: "p" using 1 5 by (rule iffD2)
    show "p ∧ q" using 6 5 by (rule conjI) }
qed

-- "La demostración estructurada es"
lemma ejemplo_18_2:
  assumes "p ∧ q"
          "p → q"
  shows  "p ∧ q"
using assms(2)
proof
  { assume "p"
    with assms(1) have "q" ..
    with ‘p‘ show "p ∧ q" .. }
next
  { assume "q"
    with assms(1) have "p" ..
    thus "p ∧ q" using ‘q‘ .. }
qed

-- "La demostración automática es"
lemma ejemplo_18_3:
  assumes "p ∧ q"
```

```
          "p  q"
  shows "p  q"
using assms
by auto

subsection {* Reglas derivadas *}

subsubsection {* Regla del modus tollens *}

text {*
  Ejemplo 19 (p. 20) Demostrar la regla del modus tollens a partir de
  las reglas básicas.
*}

-- "La demostración detallada es"
lemma ejemplo_20_1:
  assumes 1: "F  G" and
          2: "¬G"
  shows "¬F"
proof (rule notI)
  assume 3: "F"
  have 4: "G" using 1 3 by (rule mp)
  show False using 2 4 by (rule notE)
qed

-- "La demostración estructurada es"
lemma ejemplo_20_2:
  assumes "F  G"
          "¬G"
  shows   "¬F"
proof
  assume "F"
  with assms(1) have "G" ..
  with assms(2) show False ..
qed

-- "La demostración automática es"
lemma ejemplo_20_3:
  assumes "F  G"
          "¬G"
```

```
  shows "ňF"
using assms
by auto

subsubsection {* Regla de la introducción de la doble negación *}

text {*
  Ejemplo 21 (p. 21) Demostrar la regla de introducción de la doble
  negación a partir de las reglas básicas.
*}

-- "La demostración detallada es"
lemma ejemplo_21_1:
  assumes 1: "F"
  shows "ňňF"
proof (rule notI)
  assume 2: "ňF"
  show False using 2 1 by (rule notE)
qed

-- "La demostración estructurada es"
lemma ejemplo_21_2:
  assumes "F"
  shows "ňňF"
proof
  assume "ňF"
  thus False using assms ..
qed

-- "La demostración automática es"
lemma ejemplo_21_3:
  assumes "F"
  shows "ňňF"
using assms
by auto

subsubsection {* Regla de reducción al absurdo *}

text {*
  La regla de reducción al absurdo en Isabelle se correponde con la
```

```
  regla clásica de contradicción
  ů ccontr: (ňP ⟹ False) ⟹ P
*}

subsubsection {* Ley del tercio excluso *}

text {*
  La ley del tercio excluso es
  ů excluded_middle: ňP ∨ P
*}

text {*
  Ejemplo 22 (p. 23). Demostrar la ley del tercio excluso a partir de
  las reglas básicas.
*}

-- "La demostración detallada es"
lemma ejemplo_22_1:
  "F ∨ ňF"
proof (rule ccontr)
  assume 1: "ň(F ∨ ňF)"
  thus False
  proof (rule notE)
    show "F ∨ ňF"
    proof (rule disjI2)
      show "ňF"
      proof (rule notI)
        assume 2: "F"
        hence 3: "F ∨ ňF" by (rule disjI1)
        show False using 1 3 by (rule notE)
      qed
    qed
  qed
qed

-- "La demostración estructurada es"
lemma ejemplo_22_2:
  "F ∨ ňF"
proof (rule ccontr)
  assume "ň(F ∨ ňF)"
```

```
  thus False
  proof (rule notE)
    show "F  ¬F"
    proof (rule disjI2)
      show "¬F"
      proof (rule notI)
        assume "F"
        hence "F  ¬F" ..
        with ‘¬(F  ¬F)‘show False ..
      qed
    qed
  qed
qed

-- "La demostración automática es"
lemma ejemplo_22_3:
  "F  ¬F"
using assms
by auto

text {*
  Ejemplo 23 (p. 24). Demostrar
     p  q  ¬p  q
*}

-- "La demostración detallada es"
lemma ejemplo_23_1:
  assumes 1: "p  q"
  shows "¬p  q"
proof -
  have "¬p  p" by (rule excluded_middle)
  thus "¬p  q"
  proof (rule disjE)
    { assume "¬p"
      thus "¬p  q" by (rule disjI1) }
  next
    { assume 2: "p"
      have "q" using 1 2 by (rule mp)
      thus "¬p  q" by (rule disjI2) }
  qed
```

```
qed

-- "La demostración estructurada es"
lemma ejemplo_23_2:
  assumes "p  q"
  shows "¬p  q"
proof -
  have "¬p  p" ..
  thus "¬p  q"
  proof
    { assume "¬p"
      thus "¬p  q" .. }
  next
    { assume "p"
      with assms have "q" ..
      thus "¬p  q" .. }
  qed
qed

-- "La demostración automática es"
lemma ejemplo_23_3:
  assumes "p  q"
  shows "¬p  q"
using assms
by auto

subsection {* Demostraciones por contradicción *}

text {*
  Ejemplo 24. Demostrar que
     ¬p, p  q  q
*}

-- "La demostración detallada es"
lemma ejemplo_24_1:
  assumes "¬p"
          "p  q"
  shows   "q"
using ‘p  q‘
proof (rule disjE)
```

```
  assume "p"
  with assms(1) show "q" by contradiction
next
  assume "q"
  thus "q" by assumption
qed

-- "La demostración estructurada es"
lemma ejemplo_24_2:
  assumes "¬p"
          "p ∨ q"
  shows "q"
using ‹p ∨ q›
proof
  assume "p"
  with assms(1) show "q" ..
next
  assume "q"
  thus "q" .
qed

-- "La demostración automática es"
lemma ejemplo_24_3:
  assumes "¬p"
          "p ∨ q"
  shows "q"
using assms
by auto

end
```

3.2. Ejercicios: Deducción natural en lógica proposicional

```
chapter {* T3R1: Deducción natural proposicional *}

theory T3R1
imports Main
begin

text {*
```

```
  ----------------------------------------------------------------------
  El objetivo de esta relación es lemas usando sólo las reglas básicas
  de deducción natural de la lógica proposicional.

  Los ejercicios son los de la asignatura de "Lógica informática" que se
  encuentran en http://goo.gl/yrPLn

  Las reglas básicas de la deducción natural son las siguientes:
  ů conjI:      P; Q  P  Q
  ů conjunct1:  P  Q  P
  ů conjunct2:  P  Q  Q
  ů notnotD:    ňň P  P
  ů notnotI:    P  ňň P
  ů mp:         P  Q; P  Q
  ů mt:         F  G; ňG  ňF
  ů impI:       (P  Q)  P  Q
  ů disjI1:     P  P  Q
  ů disjI2:     Q  P  Q
  ů disjE:      P  Q; P  R; Q  R  R
  ů FalseE:     False  P
  ů notE:       ňP; P  R
  ů notI:       (P  False)  ňP
  ů iffI:       P  Q; Q  P  P = Q
  ů iffD1:      Q = P; Q  P
  ů iffD2:      P = Q; Q  P
  ů ccontr:     (ňP  False)  P
  ů excluded_middle: ňP  P
  ----------------------------------------------------------------------
*}

text {*
  Se usarán las reglas notnotI y mt que demostramos a continuación.
  *}

lemma notnotI: "P  ňň P"
by auto

lemma mt: "F  G; ňG  ňF"
by auto
```

```
section {* Implicaciones *}

text {* ----------------------------------------------------------
  Ejercicio 1. Demostrar
       p  q, p  q
  ---------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_1_1:
  assumes 1: "p  q" and
          2: "p"
  shows "q"
proof -
   show "q" using 1 2 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejercicio_1_2:
  assumes "p  q"
          "p"
  shows "q"
proof -
   show "q" using assms ..
qed

-- "La demostración automática es"
lemma ejercicio_1_3:
  assumes "p  q"
          "p"
  shows "q"
using assms
by auto

text {* ----------------------------------------------------------
  Ejercicio 2. Demostrar
     p  q, q  r, p  r
  ---------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_2_1:
```

```
  assumes 1: "p  q" and
          2: "q  r" and
          3: "p"
  shows "r"
proof -
  have 4: "q" using 1 3 by (rule mp)
  show "r" using 2 4 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejercicio_2_2:
  assumes "p  q"
          "q  r"
          "p"
  shows "r"
proof -
  have "q" using assms(1,3) ..
  show "r" using assms(2) `q` ..
qed

-- "La demostración automática es"
lemma ejercicio_2_3:
  assumes "p  q"
          "q  r"
          "p"
  shows "r"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 3. Demostrar
     p  (q  r), p  q, p  r
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_3_1:
  assumes 1: "p  (q  r)" and
          2: "p  q" and
          3: "p"
  shows "r"
```

```
proof -
  have 4: "q" using 2 3 by (rule mp)
  have 5: "q  r" using 1 3 by (rule mp)
  show "r" using 5 4 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejercicio_3_2:
  assumes "p  (q  r)"
          "p  q"
          "p"
  shows   "r"
proof -
  have "q" using assms(2,3) ..
  have "q  r" using assms(1,3) ..
  thus "r" using `q` ..
qed

-- "La demostración automática es"
lemma ejercicio_3_3:
  assumes "p  (q  r)"
          "p  q"
          "p"
  shows   "r"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 4. Demostrar
     p  q, q  r  p  r
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_4_1:
  assumes 1: "p  q" and
          2: "q  r"
  shows "p  r"
proof (rule impI)
  assume 3: "p"
  have 4: "q" using 1 3 by (rule mp)
```

```
  show "r" using 2 4 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejercicio_4_2:
  assumes "p  q"
          "q  r"
  shows   "p  r"
proof
  assume "p"
  with assms(1) have "q" ..
  with assms(2) show "r" ..
qed

-- "La demostración automática es"
lemma ejercicio_4_3:
  assumes "p  q"
          "q  r"
  shows   "p  r"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 5. Demostrar
     p  (q  r)  q  (p  r)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_5_1:
  assumes 1: "p  (q  r)"
  shows   "q  (p  r)"
proof (rule impI)
  assume 2: "q"
  show "p  r"
  proof (rule impI)
    assume 3: "p"
    have "q  r" using 1 3 by (rule mp)
    thus "r" using 2 by (rule mp)
  qed
qed
```

```
-- "La demostración estructurada es"
lemma ejercicio_5_2:
  assumes "p  (q  r)"
  shows   "q  (p  r)"
proof
  assume "q"
  show "p  r"
  proof
    assume "p"
    with assms(1) have "q  r" ..
    thus "r" using `q` ..
  qed
qed

-- "La demostración automática es"
lemma ejercicio_5_3:
  assumes "p  (q  r)"
  shows   "q  (p  r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 6. Demostrar
     p  (q  r)  (p  q)  (p  r)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_6_1:
  assumes 1: "p  (q  r)"
  shows   "(p  q)  (p  r)"
proof (rule impI)
  assume 2: "p  q"
  show "p  r"
  proof (rule impI)
    assume 3: "p"
    have 4: "q" using 2 3 by (rule mp)
    have 5: "q  r" using 1 3 by (rule mp)
    show "r" using 5 4 by (rule mp)
  qed
```

```
qed

-- "La demostración estructurada es"
lemma ejercicio_6_2:
  assumes "p  (q  r)"
  shows   "(p  q)  (p  r)"
proof
  assume "p  q"
  show "p  r"
  proof
    assume "p"
    with ‘p  q‘ have "q" ..
    have "q  r" using assms(1) ‘p‘ ..
    thus "r" using ‘q‘ ..
  qed
qed

-- "La demostración automática es"
lemma ejercicio_6_3:
  assumes "p  (q  r)"
  shows   "(p  q)  (p  r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 7. Demostrar
     p  q  p
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_7_1:
  assumes 1: "p"
  shows      "q  p"
proof (rule impI)
  assume 2: "q"
  show "p" using 1 by this
qed

-- "La demostración estructurada es"
lemma ejercicio_7_2:
```

```
  assumes "p"
  shows   "q  p"
proof
  assume "q"
  show "p" using assms(1) .
qed

-- "La demostración automática es"
lemma ejercicio_7_3:
  assumes "p"
  shows   "q  p"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 8. Demostrar
      p  (q  p)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_8_1:
  "p  (q  p)"
proof (rule impI)
  assume 1: "p"
  show "q  p"
  proof (rule impI)
    assume 2: "q"
    show "p" using 1 by this
  qed
qed

-- "La demostración estructurada es"
lemma ejercicio_8_2:
  "p  (q  p)"
proof
  assume "p"
  show "q  p"
  proof
    assume "q"
    show "p" using `p` .
```

```
  qed
qed

-- "La demostración automática es"
lemma ejercicio_8_3:
  "p  (q  p)"
by auto

text {* ---------------------------------------------------------------
  Ejercicio 9. Demostrar
     p  q  (q  r)  (p  r)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_9_1:
  assumes 1: "p  q"
  shows      "(q  r)  (p  r)"
proof (rule impI)
  assume 2: "q  r"
  show "p  r"
  proof (rule impI)
    assume 3: "p"
    have 4: "q" using 1 3 by (rule mp)
    show "r" using 2 4 by (rule mp)
  qed
qed

-- "La demostración estructurada es"
lemma ejercicio_9_2:
  assumes "p  q"
  shows   "(q  r)  (p  r)"
proof
  assume "q  r"
  show "p  r"
  proof
    assume "p"
    with assms(1) have "q" ..
    with `q  r` show "r" ..
  qed
qed
```

```
-- "La demostración automática es"
lemma ejercicio_9_3:
  assumes "p ⟶ q"
  shows   "(q ⟶ r) ⟶ (p ⟶ r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 10. Demostrar
     p ⟶ (q ⟶ (r ⟶ s)) ⟶ r ⟶ (q ⟶ (p ⟶ s))
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_10_1:
  assumes 1: "p ⟶ (q ⟶ (r ⟶ s))"
  shows      "r ⟶ (q ⟶ (p ⟶ s))"
proof -
  { assume 2: "r"
    { assume 3: "q"
      { assume 4: "p"
        have 5: "q ⟶ (r ⟶ s)" using 1 4 by (rule mp)
        have 6: "r ⟶ s" using 5 3 by (rule mp)
        have 7: "s" using 6 2 by (rule mp) }
      hence 8: "p ⟶ s" by (rule impI) }
    hence 9: "q ⟶ (p ⟶ s)" by (rule impI) }
  thus 10: "r ⟶ (q ⟶ (p ⟶ s))" by (rule impI)
qed

-- "Una variante de la demostración anterior es"
lemma ejercicio_10_1b:
  assumes 1: "p ⟶ (q ⟶ (r ⟶ s))"
  shows      "r ⟶ (q ⟶ (p ⟶ s))"
proof (rule impI)
  assume 2: "r"
  show "q ⟶ (p ⟶ s)"
  proof (rule impI)
    assume 3: "q"
    show "p ⟶ s"
    proof (rule impI)
```

```
      assume 4: "p"
      have 5: "q  (r  s)" using 1 4 by (rule mp)
      have 6: "r  s" using 5 3 by (rule mp)
      show "s" using 6 2 by (rule mp)
    qed
  qed
qed

-- "La demostración estructurada es"
lemma ejercicio_10_2:
  assumes "p  (q  (r  s))"
  shows   "r  (q  (p  s))"
proof
  assume "r"
  show "q  (p  s)"
  proof
    assume "q"
    show "p  s"
    proof
      assume "p"
      with assms(1) have "q  (r  s)" ..
      hence "r  s" using `q` ..
      thus "s" using `r` ..
    qed
  qed
qed

-- "La demostración automática es"
lemma ejercicio_10_3:
  assumes "p  (q  (r  s))"
  shows   "r  (q  (p  s))"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 11. Demostrar
      (p  (q  r))  ((p  q)  (p  r))
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
```

```
lemma ejercicio_11_1:
  "(p  (q  r))  ((p  q)  (p  r))"
proof -
  { assume 1: "p  (q  r)"
    { assume 2: "p  q"
      { assume 3: "p"
        have 4: "q" using 2 3 by (rule mp)
        have 5: "q  r" using 1 3 by (rule mp)
        have 6: "r" using 5 4 by (rule mp) }
      hence 7: "p  r" by (rule impI) }
    hence 8: "(p  q)  (p  r)" by (rule impI) }
  thus "(p  (q  r))  ((p  q)  (p  r))" by (rule impI)
qed

-- "La demostración estructurada es"
lemma ejercicio_11_2:
  "(p  (q  r))  ((p  q)  (p  r))"
proof
  assume "p  (q  r)"
  { assume "p  q"
    { assume "p"
      with ‘p  q‘ have "q" ..
      have "q  r" using ‘p  (q  r)‘ ‘p‘ ..
      hence "r" using ‘q‘ .. }
    hence "p  r" .. }
  thus "(p  q)  (p  r)" ..
qed

-- "La demostración automática es"
lemma ejercicio_11_3:
  "(p  (q  r))  ((p  q)  (p  r))"
by auto

text {* ---------------------------------------------------------------
  Ejercicio 12. Demostrar
     (p  q)  r  p  (q  r)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_12_1:
```

```
  assumes 1: "(p  q)  r"
  shows      "p  (q  r)"
proof -
  { assume 2: "p"
    { assume 3: "q"
      { assume 4: "p"
        have 5: "q" using 3 by this }
      hence 6: "p  q" by (rule impI)
      have 7: "r" using 1 6 by (rule mp) }
    hence 8: "q  r" by (rule impI) }
  thus 9: "p  (q  r)" by (rule impI)
qed

-- "La demostración estructurada es"
lemma ejercicio_12_2:
  assumes "(p  q)  r"
  shows   "p  (q  r)"
proof
  assume "p"
  { assume "q"
    { assume "p"
      have "q" using `q` . }
    hence "p  q" ..
    with assms(1) have "r" .. }
  thus "q  r" ..
qed

-- "La demostración automática es"
lemma ejercicio_12_3:
  assumes "(p  q)  r"
  shows   "p  (q  r)"
using assms
by auto

section {* Conjunciones *}

text {* ---------------------------------------------------------------
  Ejercicio 13. Demostrar
     p, q   p  q
  --------------------------------------------------------------- *}
```

```
-- "La demostración detallada es"
lemma ejercicio_13_1:
  assumes 1: "p" and
          2: "q"
  shows "p  q"
proof -
  show "p  q" using assms by (rule conjI)
qed

-- "La demostración estructurada es"
lemma ejercicio_13_2:
  assumes "p"
          "q"
  shows "p  q"
proof
  show "p" using assms(1) .
next
  show "q" using assms(2) .
qed

-- "La demostración automática es"
lemma ejercicio_13_3:
  assumes "p"
          "q"
  shows "p  q"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 14. Demostrar
     p  q  p
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_14_1:
  assumes 1: "p  q"
  shows      "p"
proof -
  show "p" using 1 by (rule conjunct1)
```

```
qed

-- "La demostración estructurada es"
lemma ejercicio_14_2:
  assumes "p  q"
  shows   "p"
proof -
  show "p" using assms ..
qed

-- "La demostración automática es"
lemma ejercicio_14_3:
  assumes "p  q"
  shows   "p"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 15. Demostrar
     p  q  q
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_15_1:
  assumes 1: "p  q"
  shows      "q"
proof -
  show "q" using 1 by (rule conjunct2)
qed

-- "La demostración estructurada es"
lemma ejercicio_15_2:
  assumes "p  q"
  shows   "q"
proof -
  show "q" using assms ..
qed

-- "La demostración automática es"
lemma ejercicio_15_3:
```

```
  assumes "p ⟶ q"
  shows   "¬q ⟶ ¬p"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 16. Demostrar
     p ⟶ (q ⟶ r) ⊢ (p ∧ q) ⟶ r
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_16_1:
  assumes 1: "p ⟶ (q ⟶ r)"
  shows      "(p ∧ q) ⟶ r"
proof -
  have 2: "p" using 1 by (rule conjunct1)
  have 3: "q ⟶ r" using 1 by (rule conjunct2)
  have 4: "q" using 3 by (rule conjunct1)
  have 5: "p ∧ q" using 2 4 by (rule conjI)
  have 6: "r" using 3 by (rule conjunct2)
  show 7: "(p ∧ q) ⟶ r" using 5 6 by (rule conjI)
qed

-- "La demostración estructurada es"
lemma ejercicio_16_2:
  assumes "p ⟶ (q ⟶ r)"
  shows   "(p ∧ q) ⟶ r"
proof
  show "p ∧ q"
  proof
    show "p" using assms ..
  next
    have "q ⟶ r" using assms ..
    thus "q" ..
  qed
next
    have "q ⟶ r" using assms ..
    thus "r" ..
qed
```

```
-- "La demostración automática es"
lemma ejercicio_16_3:
  assumes "p  (q  r)"
  shows   "(p  q)  r"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 17. Demostrar
     (p  q)  r  p  (q  r)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_17_1:
  assumes 1: "(p  q)  r"
  shows      "p  (q  r)"
proof -
  have 2: "p  q" using 1 by (rule conjunct1)
  have 3: "p" using 2 by (rule conjunct1)
  have 4: "q" using 2 by (rule conjunct2)
  have 5: "r" using 1 by (rule conjunct2)
  have 6: "q  r" using 4 5 by (rule conjI)
  show 7: "p  (q  r)" using 3 6 by (rule conjI)
qed

-- "La demostración estructurada es"
lemma ejercicio_17_2:
  assumes "(p  q)  r"
  shows   "p  (q  r)"
proof
  have "p  q" using assms ..
  thus "p" ..
next
  show "q  r"
  proof
    have "p  q" using assms ..
    thus "q" ..
  next
    show "r" using assms ..
  qed
```

```
qed

-- "La demostración automática es"
lemma ejercicio_17_3:
  assumes "(p  q)  r"
  shows   "p  (q  r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 18. Demostrar
     p  q  p  q
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_18_1:
  assumes 1: "p  q"
  shows      "p  q"
proof (rule impI)
  assume 2: "p"
  show "q" using 1 by (rule conjunct2)
qed

-- "La demostración estructurada es"
lemma ejercicio_18_2:
  assumes "p  q"
  shows   "p  q"
proof
  assume "p"
  show "q" using assms ..
qed

-- "La demostración automática es"
lemma ejercicio_18_3:
  assumes "p  q"
  shows   "p  q"
using assms
by auto

text {* ---------------------------------------------------------------
```

```
  Ejercicio 19. Demostrar
     (p  q)  (p  r)  p  q  r
  ------------------------------------------------------------------ *}

-- "La demostración detallada es"
lemma ejercicio_19_1:
  assumes 1: "(p  q)  (p  r)"
  shows      "p  q  r"
proof (rule impI)
  assume 2: "p"
  have 3: "p  q" using 1 by (rule conjunct1)
  have 4: "q" using 3 2 by (rule mp)
  have 5: "p  r" using 1 by (rule conjunct2)
  have 6: "r" using 5 2 by (rule mp)
  show 7: "q  r" using 4 6 by (rule conjI)
qed

-- "La demostración estructurada es"
lemma ejercicio_19_2:
  assumes "(p  q)  (p  r)"
  shows   "p  q  r"
proof
  assume "p"
  show "q  r"
  proof
    have "p  q" using assms ..
    thus "q" using ‘p‘ ..
  next
    have "p  r" using assms ..
    thus "r" using ‘p‘ ..
  qed
qed

-- "La demostración automática es"
lemma ejercicio_19_3:
  assumes "(p  q)  (p  r)"
  shows   "p  q  r"
using assms
by auto
```

```
text {* ---------------------------------------------------------------
  Ejercicio 20. Demostrar
     p  q  r  (p  q)  (p  r)
  --------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_20_1:
  assumes 1: "p  q  r"
  shows      "(p  q)  (p  r)"
proof (rule conjI)
  show "p  q"
  proof (rule impI)
    assume 2: "p"
    have "q  r" using 1 2 by (rule mp)
    thus "q" by (rule conjunct1)
  qed
next
  show "p  r"
  proof (rule impI)
    assume 3: "p"
    have "q  r" using 1 3 by (rule mp)
    thus "r" by (rule conjunct2)
  qed
qed

-- "La demostración estructurada es"
lemma ejercicio_20_2:
  assumes "p  q  r"
  shows   "(p  q)  (p  r)"
proof
  show "p  q"
  proof
    assume "p"
    have "q  r" using assms `p` ..
    thus "q" ..
  qed
next
  show "p  r"
  proof
    assume "p"
```

```
    have "q ⟹ r" using assms ‹p› ..
    thus "r" ..
  qed
qed

-- "La demostración automática es"
lemma ejercicio_20_3:
  assumes "p ⟹ q ⟹ r"
  shows   "(p ⟹ q) ⟹ (p ⟹ r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 21. Demostrar
     p ⟹ (q ⟹ r) ⟹ p ⟹ q ⟹ r
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_21_1:
  assumes 1: "p ⟹ (q ⟹ r)"
  shows      "p ⟹ q ⟹ r"
proof (rule impI)
  assume 2: "p ⟹ q"
  have 3: "p" using 2 by (rule conjunct1)
  have 4: "q" using 2 by (rule conjunct2)
  have 5: "q ⟹ r" using 1 3 by (rule mp)
  show 6: "r" using 5 4 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejercicio_21_2:
  assumes "p ⟹ (q ⟹ r)"
  shows   "p ⟹ q ⟹ r"
proof
  assume "p ⟹ q"
  hence "p" ..
  have "q" using ‹p ⟹ q› ..
  have "q ⟹ r" using assms(1) ‹p› ..
  thus "r" using ‹q› ..
qed
```

```
-- "La demostración automática es"
lemma ejercicio_21_3:
  assumes "p  (q  r)"
  shows   "p  q  r"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 22. Demostrar
     p  q  r  p  (q  r)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_22_1:
  assumes 1: "p  q  r"
  shows      "p  (q  r)"
proof (rule impI)
  assume 2: "p"
  { assume 3: "q"
    have 4: "p  q" using 2 3 by (rule conjI)
    have "r" using 1 4 by (rule mp) }
  thus "q  r" by (rule impI)
qed

-- "La demostración estructurada es"
lemma ejercicio_22_2:
  assumes "p  q  r"
  shows   "p  (q  r)"
proof
  assume "p"
  show "q  r"
  proof
    assume "q"
    with `p` have "p  q" ..
    with  assms(1) show "r" ..
  qed
qed

-- "La demostración automática es"
```

```
lemma ejercicio_22_3:
  assumes "p  q  r"
  shows   "p  (q  r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 23. Demostrar
     (p  q)  r  p  q  r
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_23_1:
  assumes 1: "(p  q)  r"
  shows      "p  q  r"
proof (rule impI)
  assume 2: "p  q"
  have 3: "p  q"
  proof (rule impI)
    assume 4: "p"
    show "q" using 2 by (rule conjunct2)
  qed
  show "r" using 1 3 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejercicio_23_2:
  assumes "(p  q)  r"
  shows   "p  q  r"
proof
  assume "p  q"
  have "p  q"
  proof
    assume "p"
    show "q" using `p  q` ..
  qed
  with assms show "r" ..
qed

-- "La demostración automática es"
```

```
lemma ejercicio_23_3:
  assumes "(p ∧ q) ∨ r"
  shows   "p ∨ q ∨ r"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 24. Demostrar
     p ∨ (q ∨ r), (p ∧ q) ∨ r
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_24_1:
  assumes 1: "p ∨ (q ∨ r)"
  shows      "(p ∧ q) ∨ r"
proof (rule impI)
  assume 2: "p ∧ q"
  have 3: "p" using 1 by (rule conjunct1)
  have 4: "q" using 2 3 by (rule mp)
  have 5: "q ∨ r" using 1 by (rule conjunct2)
  thus "r" using 4 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejercicio_24_2:
  assumes "p ∨ (q ∨ r)"
  shows   "(p ∧ q) ∨ r"
proof
  assume "p ∧ q"
  have "p" using assms ..
  with ‘p ∧ q‘ have "q" ..
  have "q ∨ r" using assms ..
  thus "r" using ‘q‘ ..
qed

-- "La demostración automática es"
lemma ejercicio_24_3:
  assumes "p ∨ (q ∨ r)"
  shows   "(p ∧ q) ∨ r"
using assms
```

```
by auto

section {* Disyunciones *}

text {* ----------------------------------------------------------------
  Ejercicio 25. Demostrar
     p  p  q
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_25_1:
  assumes 1: "p"
  shows      "p  q"
proof -
  show "p  q" using 1 by (rule disjI1)
qed

-- "La demostración estructurada es"
lemma ejercicio_25_2:
  assumes "p"
  shows   "p  q"
proof -
  show "p  q" using assms ..
qed

-- "La demostración estructurada es"
lemma ejercicio_25_3:
  assumes "p"
  shows   "p  q"
using assms
by auto

text {* ----------------------------------------------------------------
  Ejercicio 26. Demostrar
     q  p  q
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_26_1:
  assumes 1: "q"
```

```
  shows      "p  q"
proof -
  show "p  q" using 1 by (rule disjI2)
qed

-- "La demostración estructurada es"
lemma ejercicio_26_2:
  assumes "q"
  shows   "p  q"
proof -
  show "p  q" using assms ..
qed

-- "La demostración estructurada es"
lemma ejercicio_26_3:
  assumes "q"
  shows   "p  q"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 27. Demostrar
     p  q  q  p
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_27_1:
  assumes 1: "p  q"
  shows      "q  p"
using 1
proof (rule disjE)
  assume "p"
  thus "q  p" by (rule disjI2)
next
  assume "q"
  thus "q  p" by (rule disjI1)
qed

-- "La demostración estructurada es"
lemma ejercicio_27_2:
```

```
  assumes "p ⟶ q"
  shows   "¬q ⟶ ¬p"
using assms
proof
  assume "¬p"
  thus "¬q ⟶ ¬p" ..
next
  assume "q"
  thus "¬q ⟶ ¬p" ..
qed

-- "La demostración estructurada es"
lemma ejercicio_27_3:
  assumes "p ⟶ q"
  shows   "¬q ⟶ ¬p"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 28. Demostrar
     q ∨ r ⟶ p ∨ q ⟶ p ∨ r
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_28_1:
  assumes 1: "q ∨ r"
  shows      "p ∨ q ⟶ p ∨ r"
proof (rule impI)
  assume "p ∨ q"
  thus "p ∨ r"
  proof (rule disjE)
    assume "p"
    thus "p ∨ r" by (rule disjI1)
  next
    assume "q"
    have "r" using assms `q` by (rule mp)
    thus "p ∨ r" by (rule disjI2)
  qed
qed
```

```
-- "La demostración estructurada es"
lemma ejercicio_28_2:
  assumes "q  r"
  shows   "p  q  p  r"
proof
  assume "p  q"
  thus "p  r"
  proof
    assume "p"
    thus "p  r" ..
  next
    assume "q"
    have "r" using assms `q` ..
    thus "p  r" ..
  qed
qed

-- "La demostración automática es"
lemma ejercicio_28_3:
  assumes "q  r"
  shows   "p  q  p  r"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 29. Demostrar
     p  p  p
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_29_1:
  assumes 1: "p  p"
  shows      "p"
using 1
proof (rule disjE)
  assume "p"
  thus "p" by this
next
  assume "p"
  thus "p" by this
```

```
qed

-- "La demostración estructurada es"
lemma ejercicio_29_2:
  assumes "p  p"
  shows   "p"
using assms
proof
  assume "p"
  thus "p" .
next
  assume "p"
  thus "p" .
qed

-- "La demostración estructurada es"
lemma ejercicio_29_3:
  assumes "p  p"
  shows   "p"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 30. Demostrar
     p  p  p
  ------------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_30_1:
  assumes 1: "p"
  shows      "p  p"
proof -
  show "p  p" using 1 by (rule disjI1)
qed

-- "La demostración estructurada es"
lemma ejercicio_30_2:
  assumes "p"
  shows   "p  p"
proof -
```

```
  show "p  p" using assms ..
qed

-- "La demostración estructurada es"
lemma ejercicio_30_3:
  assumes "p"
  shows   "p  p"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 31. Demostrar
     p  (q  r)  (p  q)  r
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_31_1:
  assumes 1: "p  (q  r)"
  shows      "(p  q)  r"
using 1
proof (rule disjE)
  assume "p"
  hence "p  q" by (rule disjI1)
  thus "(p  q)  r" by (rule disjI1)
next
  assume "q  r"
  thus "(p  q)  r"
  proof (rule disjE)
    assume "q"
    hence "p  q" by (rule disjI2)
    thus "(p  q)  r" by (rule disjI1)
  next
    assume "r"
    thus "(p  q)  r" by (rule disjI2)
  qed
qed

-- "La demostración estructurada es"
lemma ejercicio_31_2:
  assumes "p  (q  r)"
```

```
  shows   "(p  q)  r"
using assms
proof
  assume "p"
  hence "p  q" ..
  thus "(p  q)  r" ..
next
  assume "q  r"
  thus "(p  q)  r"
  proof
    assume "q"
    hence "p  q" ..
    thus "(p  q)  r" ..
  next
    assume "r"
    thus "(p  q)  r" ..
  qed
qed

-- "La demostración estructurada es"
lemma ejercicio_31_3:
  assumes "p  (q  r)"
  shows   "(p  q)  r"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 32. Demostrar
     (p  q)  r  p  (q  r)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_32_1:
  assumes 1: "(p  q)  r"
  shows      "p  (q  r)"
using 1
proof (rule disjE)
  assume "p  q"
  thus "p  (q  r)"
  proof (rule disjE)
```

```
    assume "p"
    thus "p ∨ (q ∨ r)" by (rule disjI1)
  next
    assume "q"
    hence "q ∨ r" by (rule disjI1)
    thus "p ∨ (q ∨ r)" by (rule disjI2)
  qed
next
  assume "r"
  hence "q ∨ r" by (rule disjI2)
  thus "p ∨ (q ∨ r)" by (rule disjI2)
qed

-- "La demostración estructurada es"
lemma ejercicio_32_2:
  assumes "(p ∨ q) ∨ r"
  shows   "p ∨ (q ∨ r)"
using assms
proof
  assume "p ∨ q"
  thus "p ∨ (q ∨ r)"
  proof
    assume "p"
    thus "p ∨ (q ∨ r)" ..
  next
    assume "q"
    hence "q ∨ r" ..
    thus "p ∨ (q ∨ r)" ..
  qed
next
  assume "r"
  hence "q ∨ r" ..
  thus "p ∨ (q ∨ r)" ..
qed

-- "La demostración estructurada es"
lemma ejercicio_32_3:
  assumes "(p ∨ q) ∨ r"
  shows   "p ∨ (q ∨ r)"
using assms
```

```
by auto

text {* ---------------------------------------------------------------
  Ejercicio 33. Demostrar
     p  (q  r)  (p  q)  (p  r)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_33_1:
  assumes 1: "p  (q  r)"
  shows      "(p  q)  (p  r)"
proof -
  have 2: "p" using 1 by (rule conjunct1)
  have 3: "q  r" using 1 by (rule conjunct2)
  thus "(p  q)  (p  r)"
  proof (rule disjE)
    assume 4: "q"
    have "p  q" using 2 4 by (rule conjI)
    thus "(p  q)  (p  r)" by (rule disjI1)
  next
    assume 5: "r"
    have "q  r" using 1 by (rule conjunct2)
    thus "(p  q)  (p  r)"
    proof (rule disjE)
      assume "q"
      have "p  q" using `p` `q` by (rule conjI)
      thus "(p  q)  (p  r)" by (rule disjI1)
    next
      assume "r"
      have "p  r" using `p` `r` by (rule conjI)
      thus "(p  q)  (p  r)" by (rule disjI2)
    qed
  qed
qed

-- "La demostración estructurada es"
lemma ejercicio_33_2:
  assumes "p  (q  r)"
  shows   "(p  q)  (p  r)"
proof -
```

```
  have "p" using assms ..
  have "q  r" using assms ..
  thus "(p  q)  (p  r)"
  proof
    assume "q"
    with ‘p‘ have "p  q" ..
    thus "(p  q)  (p  r)" ..
  next
    assume "r"
    hence "q  r" ..
    thus "(p  q)  (p  r)"
    proof
      assume "q"
      with ‘p‘ have "p  q" ..
      thus "(p  q)  (p  r)" ..
    next
      assume "r"
      with ‘p‘ have "p  r" ..
      thus "(p  q)  (p  r)" ..
    qed
  qed
qed

-- "La demostración automática es"
lemma ejercicio_33_3:
  assumes "p  (q  r)"
  shows   "(p  q)  (p  r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 34. Demostrar
     (p  q)  (p  r)  p  (q  r)
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_34_1:
  assumes "(p  q)  (p  r)"
  shows   "p  (q  r)"
using assms
```

```
proof (rule disjE)
  assume 1: "p  q"
  show "p  (q  r)"
  proof (rule conjI)
    show "p" using 1 by (rule conjunct1)
  next
    have "q" using 1 by (rule conjunct2)
    thus "q  r" by (rule disjI1)
  qed
next
  assume 2: "p  r"
  hence 3: "p" by (rule conjunct1)
  have "r" using 2 by (rule conjunct2)
  hence 4: "q  r" by (rule disjI2)
  show "p  (q  r)" using 3 4 by (rule conjI)
qed

-- "La demostración estructurada es"
lemma ejercicio_34_2:
  assumes "(p  q)  (p  r)"
  shows   "p  (q  r)"
using assms
proof
  assume "p  q"
  show "p  (q  r)"
  proof
    show "p" using ‘p  q‘ ..
  next
    have "q" using ‘p  q‘ ..
    thus "q  r" ..
  qed
next
  assume "p  r"
  hence "p" ..
  have "r" using ‘p  r‘ ..
  hence "q  r" ..
  with ‘p‘ show "p  (q  r)" ..
qed

-- "La demostración estructurada es"
```

```
lemma ejercicio_34_3:
  assumes "(p ⟶ q) ⟶ (p ⟶ r)"
  shows   "p ⟶ (q ⟶ r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 35. Demostrar
     p ⟶ (q ⟶ r) ⊢ (p ⟶ q) ⟶ (p ⟶ r)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_35_1:
  assumes "p ⟶ (q ⟶ r)"
  shows   "(p ⟶ q) ⟶ (p ⟶ r)"
using assms
proof
  assume "p"
  show "(p ⟶ q) ⟶ (p ⟶ r)"
  proof
    show "p ⟶ q" using ‘p‘ ..
  next
    show "p ⟶ r" using ‘p‘ ..
  qed
next
  assume "q ⟶ r"
  show "(p ⟶ q) ⟶ (p ⟶ r)"
  proof
    have "q" using ‘q ⟶ r‘ ..
    thus "p ⟶ q" ..
  next
    have "r" using ‘q ⟶ r‘ ..
    thus "p ⟶ r" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_35_2:
  assumes "p ⟶ (q ⟶ r)"
  shows   "(p ⟶ q) ⟶ (p ⟶ r)"
```

```
using assms
proof (rule disjE)
  assume "p"
  show "(p ∧ q) ∨ (p ∧ r)"
  proof (rule conjI)
    show "p ∨ q" using ‹p› by (rule disjI1)
  next
    show "p ∨ r" using ‹p› by (rule disjI1)
  qed
next
  assume "q ∧ r"
  show "(p ∧ q) ∨ (p ∧ r)"
  proof (rule conjI)
    have "q" using ‹q ∧ r› ..
    thus "p ∨ q" by (rule disjI2)
  next
    have "r" using ‹q ∧ r› by (rule conjunct2)
    thus "p ∨ r" by (rule disjI2)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_35_3:
  assumes "p ∨ (q ∧ r)"
  shows   "(p ∨ q) ∧ (p ∨ r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 36. Demostrar
     (p ∨ q) ∧ (p → r) ⊢ p ∨ (q ∧ r)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_36_1:
  assumes "(p ∨ q) ∧ (p → r)"
  shows   "p ∨ (q ∧ r)"
proof -
  have "p ∨ q" using assms ..
  thus "p ∨ (q ∧ r)"
```

```
  proof
    assume "p"
    thus "p  (q  r)" ..
  next
    assume "q"
    have "p  r" using assms ..
    thus "p  (q  r)"
    proof
      assume "p"
      thus "p  (q  r)" ..
    next
      assume "r"
      with ‘q‘ have "q  r" ..
      thus "p  (q  r)" ..
    qed
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_36_2:
  assumes "(p  q)  (p  r)"
  shows   "p  (q  r)"
proof -
  have "p  q" using assms by (rule conjunct1)
  thus "p  (q  r)"
  proof (rule disjE)
    assume "p"
    thus "p  (q  r)" by (rule disjI1)
  next
    assume "q"
    have "p  r" using assms by (rule conjunct2)
    thus "p  (q  r)"
    proof (rule disjE)
      assume "p"
      thus "p  (q  r)" by (rule disjI1)
    next
      assume "r"
      have "q  r" using ‘q‘ ‘r‘ by (rule conjI)
      thus "p  (q  r)" by (rule disjI2)
    qed
```

```
  qed
qed

-- "La demostración automática es"
lemma ejercicio_36_3:
  assumes "(p  q)  (p  r)"
  shows   "p  (q  r)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 37. Demostrar
     (p  r)  (q  r)  p  q  r
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_37_1:
  assumes "(p  r)  (q  r)"
  shows   "p  q  r"
proof
  assume "p  q"
  thus "r"
  proof
    assume "p"
    have "p  r" using assms ..
    thus "r" using `p` ..
  next
    assume "q"
    have "q  r" using assms ..
    thus "r" using `q` ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_37_2:
  assumes "(p  r)  (q  r)"
  shows   "p  q  r"
proof (rule impI)
  assume "p  q"
  thus "r"
```

```
  proof (rule disjE)
    assume "p"
    have "p  r" using assms by (rule conjunct1)
    thus "r" using `p` by (rule mp)
  next
    assume "q"
    have "q  r" using assms by (rule conjunct2)
    thus "r" using `q` by (rule mp)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_37_3:
  assumes "(p  r)  (q  r)"
  shows   "p  q  r"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 38. Demostrar
     p  q  r  (p  r)  (q  r)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_38_1:
  assumes "p  q  r"
  shows   "(p  r)  (q  r)"
proof
  show "p  r"
  proof
    assume "p"
    hence "p  q" ..
    with assms show "r" ..
  qed
next
  show "q  r"
  proof
    assume "q"
    hence "p  q" ..
    with assms show "r" ..
```

```
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_38_2:
  assumes "p  q  r"
  shows   "(p  r)  (q  r)"
proof (rule conjI)
  show "p  r"
  proof (rule impI)
    assume "p"
    hence "p  q" by (rule disjI1)
    show "r" using assms ‘p  q‘ by (rule mp)
  qed
next
  show "q  r"
  proof (rule impI)
    assume "q"
    hence "p  q" by (rule disjI2)
    show "r" using assms ‘p  q‘ by (rule mp)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_38_3:
  assumes "p  q  r"
  shows   "(p  r)  (q  r)"
using assms
by auto

section {* Negación *}

text {* ---------------------------------------------------------------------
  Ejercicio 39. Demostrar
     p  ňňp
  --------------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_39_1:
  assumes "p"
```

```
  shows   "¬¬p"
proof -
  show "¬¬p" using assms by (rule notnotI)
qed

-- "La demostración automática es"
lemma ejercicio_39_2:
  assumes "p"
  shows   "¬¬p"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 40. Demostrar
     ¬p  p  q
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_40_1:
  assumes "¬p"
  shows   "p → q"
proof
  assume "p"
  with assms(1) show "q" ..
qed

-- "La demostración detallada es"
lemma ejercicio_40_2:
  assumes "¬p"
  shows   "p → q"
proof (rule impI)
  assume "p"
  show "q" using assms(1) `p` by (rule notE)
qed

-- "La demostración automática es"
lemma ejercicio_40_3:
  assumes "¬p"
  shows   "p → q"
using assms
```

```
by auto

text {* ---------------------------------------------------------------
  Ejercicio 41. Demostrar
     p  q  ¬q  ¬p
  --------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_41_1:
  assumes "p  q"
  shows   "¬q  ¬p"
proof
  assume "¬q"
  with assms(1) show "¬p" by (rule mt)
qed

-- "La demostración detallada es"
lemma ejercicio_41_2:
  assumes "p  q"
  shows   "¬q  ¬p"
proof (rule impI)
  assume "¬q"
  show "¬p" using assms(1) `¬q` by (rule mt)
qed

-- "La demostración automática es"
lemma ejercicio_41_3:
  assumes "p  q"
  shows   "¬q  ¬p"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 42. Demostrar
     p  q, ¬q  p
  --------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_42_1:
  assumes "p  q"
```

```
          "¬q"
  shows   "p"
using assms(1)
proof
  assume "p"
  thus "p" .
next
  assume "q"
  with assms(2) show "p" ..
qed

-- "La demostración detallada es"
lemma ejercicio_42_2:
  assumes "p ∨ q"
          "¬q"
  shows   "p"
using assms(1)
proof (rule disjE)
  assume "p"
  thus "p" by this
next
  assume "q"
  show "p" using assms(2) `q` by (rule notE)
qed

-- "La demostración automática es"
lemma ejercicio_42_3:
  assumes "p ∨ q"
          "¬q"
  shows   "p"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 42. Demostrar
     p ∨ q, ¬p ⊢ q
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_43_1:
```

```
  assumes "p  q"
          "¬p"
  shows   "q"
using assms(1)
proof
  assume "p"
  with assms(2) show "q" ..
next
  assume "q"
  thus "q" .
qed

-- "La demostración detallada es"
lemma ejercicio_43_2:
  assumes "p  q"
          "¬p"
  shows   "q"
using assms(1)
proof (rule disjE)
  assume "p"
  show "q" using assms(2) `p` by (rule notE)
next
  assume "q"
  thus "q" by this
qed

-- "La demostración automática es"
lemma ejercicio_43_3:
  assumes "p  q"
          "¬p"
  shows   "q"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 44. Demostrar
     p  q  ¬(¬p  ¬q)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
```

```
lemma ejercicio_44_1:
  assumes "p  q"
  shows   "ň(ňp  ňq)"
proof
  assume "ňp  ňq"
  note ‘p  q‘
  thus "False"
  proof
    assume "p"
    have "ňp" using ‘ňp  ňq‘ ..
    thus "False" using ‘p‘ ..
  next
    assume "q"
    have "ňq" using ‘ňp  ňq‘ ..
    thus "False" using ‘q‘ ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_44_2:
  assumes "p  q"
  shows   "ň(ňp  ňq)"
proof (rule notI)
  assume "ňp  ňq"
  note ‘p  q‘
  thus "False"
  proof
    assume "p"
    have "ňp" using ‘ňp  ňq‘ by (rule conjunct1)
    thus "False" using ‘p‘ by (rule notE)
  next
    assume "q"
    have "ňq" using ‘ňp  ňq‘ by (rule conjunct2)
    thus "False" using ‘q‘ by (rule notE)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_44_3:
  assumes "p  q"
```

```
  shows   "ň(ňp  ňq)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 45. Demostrar
     p  q  ň(ňp  ňq)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_45_1:
  assumes "p  q"
  shows   "ň(ňp  ňq)"
proof
  assume "ňp  ňq"
  thus "False"
  proof
    assume "ňp"
    have "p" using assms ..
    with ‘ňp‘ show "False" ..
  next
    assume "ňq"
    have "q" using assms ..
    with ‘ňq‘ show "False" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_45_2:
  assumes "p  q"
  shows   "ň(ňp  ňq)"
proof (rule notI)
  assume "ňp  ňq"
  thus "False"
  proof
    assume "ňp"
    have "p" using assms by (rule conjunct1)
    show "False" using ‘ňp‘ ‘p‘ by (rule notE)
  next
    assume "ňq"
```

```
    have "q" using assms by (rule conjunct2)
    show "False" using ‘ňq‘ ‘q‘ by (rule notE)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_45_3:
  assumes "p  q"
  shows   "ň(ňp  ňq)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 46. Demostrar
     ň(p  q)  ňp  ňq
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_46_1:
  assumes "ň(p  q)"
  shows   "ňp  ňq"
proof
  show "ňp"
  proof
    assume "p"
    hence "p  q" ..
    with assms show "False" ..
  qed
next
  show "ňq"
  proof
    assume "q"
    hence "p  q" ..
    with assms show "False" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_46_2:
  assumes "ň(p  q)"
```

```
  shows   "¬p ∨ ¬q"
proof (rule conjI)
  show "¬p"
  proof (rule notI)
    assume "p"
    hence "p ∨ q" by (rule disjI1)
    show "False" using assms ‹p ∨ q› by (rule notE)
  qed
next
  show "¬q"
  proof (rule notI)
    assume "q"
    hence "p ∨ q" by (rule disjI2)
    show "False" using assms ‹p ∨ q› by (rule notE)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_46_3:
  assumes "¬(p ∨ q)"
  shows   "¬p ∧ ¬q"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 47. Demostrar
     ¬p ∨ ¬q ⊢ ¬(p ∧ q)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_47_1:
  assumes "¬p ∨ ¬q"
  shows   "¬(p ∧ q)"
proof
  assume "p ∧ q"
  thus False
  proof
    assume "p"
    have "¬p" using assms ..
    thus False using ‹p› ..
```

```
  next
    assume "q"
    have "¬q" using assms ..
    thus False using ‹q› ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_47_2:
  assumes "¬p ∨ ¬q"
  shows   "¬(p ∧ q)"
proof (rule notI)
  assume "p ∧ q"
  thus False
  proof (rule disjE)
    assume "p"
    have "¬p" using assms by (rule conjunct1)
    thus False using ‹p› by (rule notE)
  next
    assume "q"
    have "¬q" using assms by (rule conjunct2)
    thus False using ‹q› by (rule notE)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_47_3:
  assumes "¬p ∨ ¬q"
  shows   "¬(p ∧ q)"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 48. Demostrar
     ¬p ∨ ¬q ⊢ ¬(p ∧ q)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_48_1:
  assumes "¬p ∨ ¬q"
```

```
  shows   "¬(p ∧ q)"
proof
  assume "p ∧ q"
  note ‹¬p ∨ ¬ q›
  thus False
  proof
    assume "¬p"
    have "p" using ‹p ∧ q› ..
    with ‹¬p› show False ..
  next
    assume "¬q"
    have "q" using ‹p ∧ q› ..
    with ‹¬q› show False ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_48_2:
  assumes "¬p ∨ ¬q"
  shows   "¬(p ∧ q)"
proof (rule notI)
  assume "p ∧ q"
  note ‹¬p ∨ ¬ q›
  thus False
  proof (rule disjE)
    assume "¬p"
    have "p" using ‹p ∧ q› by (rule conjunct1)
    show False using ‹¬p› ‹p› by (rule notE)
  next
    assume "¬q"
    have "q" using ‹p ∧ q› by (rule conjunct2)
    show False using ‹¬q› ‹q› by (rule notE)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_48_3:
  assumes "¬p ∨ ¬q"
  shows   "¬(p ∧ q)"
using assms
```

```
by auto

text {* ----------------------------------------------------------------
  Ejercicio 49. Demostrar
      ň(p  ňp)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_49_1:
  "ň(p  ňp)"
proof
  assume "p  ňp"
  hence "p" ..
  have "ňp" using `p  ňp` ..
  thus False using `p` ..
qed

-- "La demostración detallada es"
lemma ejercicio_49_2:
  "ň(p  ňp)"
proof (rule notI)
  assume "p  ňp"
  hence "p" by (rule conjunct1)
  have "ňp" using `p  ňp` by (rule conjunct2)
  thus False using `p` by (rule notE)
qed

-- "La demostración automática es"
lemma ejercicio_49_3:
  "ň(p  ňp)"
by auto

text {* ----------------------------------------------------------------
  Ejercicio 50. Demostrar
     p  ňp  q
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_50_1:
  assumes "p  ňp"
```

```
  shows   "q"
proof -
  have "p" using assms ..
  have "¬p" using assms ..
  thus "q" using ‘p‘ ..
qed

-- "La demostración detallada es"
lemma ejercicio_50_2:
  assumes "p  ¬p"
  shows   "q"
proof -
  have "p" using assms by (rule conjunct1)
  have "¬p" using assms by (rule conjunct2)
  thus "q" using ‘p‘ by (rule notE)
qed

-- "La demostración automática es"
lemma ejercicio_50_3:
  assumes "p  ¬p"
  shows   "q"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 51. Demostrar
     ¬¬p  p
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma ejercicio_51_1:
  assumes "¬¬p"
  shows   "p"
using assms
by (rule notnotD)

-- "La demostración automática es"
lemma ejercicio_51_2:
  assumes "¬¬p"
  shows   "p"
```

```
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 52. Demostrar
      p  ¬p
  --------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_52_1:
  "p  ¬p"
proof -
  have "¬¬p  ¬p" ..
  thus "p  ¬p" by simp
qed

-- "La demostración detallada es"
lemma ejercicio_52_2:
  "p  ¬p"
proof -
  have "¬¬p  ¬p" by (rule excluded_middle)
  thus "p  ¬p" by simp
qed

-- "La demostración automática es"
lemma ejercicio_52_3:
  "p  ¬p"
by auto

text {* ---------------------------------------------------------------
  Ejercicio 53. Demostrar
      ((p  q)  p)  p
  --------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_53_1:
  "((p  q)  p)  p"
proof
  assume "(p  q)  p"
  show "p"
```

```
  proof (rule ccontr)
    assume "¬p"
    have "¬(p ∧ q)" using ‘(p ∧ q) ∨ p‘ ‘¬p‘ by (rule mt)
    have "p ∧ q"
    proof
      assume "p"
      with ‘¬p‘ show "q" ..
    qed
    show False using ‘¬(p ∧ q)‘ ‘p ∧ q‘ ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_53_2:
  "((p ∧ q) ∨ p) ∨ p"
proof (rule impI)
  assume "(p ∧ q) ∨ p"
  show "p"
  proof (rule ccontr)
    assume "¬p"
    have "¬(p ∧ q)" using ‘(p ∧ q) ∨ p‘ ‘¬p‘ by (rule mt)
    have "p ∧ q"
    proof (rule impI)
      assume "p"
      show "q" using ‘¬p‘ ‘p‘ by (rule notE)
    qed
    show False using ‘¬(p ∧ q)‘ ‘p ∧ q‘ by (rule notE)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_53_3:
  "((p ∧ q) ∨ p) ∨ p"
by auto

text {* ---------------------------------------------------------------
  Ejercicio 54. Demostrar
     ¬q → ¬p → p → q
  ---------------------------------------------------------------- *}
```

```
-- "La demostración estructurada es"
lemma ejercicio_54_1:
  assumes "¬q  ¬p"
  shows   "p  q"
proof
  assume "p"
  show "q"
  proof (rule ccontr)
    assume "¬q"
    with assms have "¬p" ..
    thus False using ‘p‘ ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_54_2:
  assumes "¬q  ¬p"
  shows   "p  q"
proof (rule impI)
  assume "p"
  show "q"
  proof (rule ccontr)
    assume "¬q"
    have "¬p" using assms ‘¬q‘ by (rule mp)
    thus False using ‘p‘ by (rule notE)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_54_3:
  assumes "¬q  ¬p"
  shows   "p  q"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 55. Demostrar
     ¬(¬p  ¬q)  p  q
  ---------------------------------------------------------------- *}
```

```
-- "La demostración estructurada es"
lemma ejercicio_55_1:
  assumes "¬(¬p ∨ ¬q)"
  shows   "p ∧ q"
proof -
  have "¬p ∨ p" ..
  thus "p ∧ q"
  proof
    assume "¬p"
    have "¬q ∨ q" ..
    thus "p ∧ q"
    proof
      assume "¬q"
      with ‹¬p› have "¬p ∨ ¬q" ..
      with assms show "p ∧ q" ..
    next
      assume "q"
      thus "p ∧ q" ..
    qed
  next
    assume "p"
    thus "p ∧ q" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_55_2:
  assumes "¬(¬p ∨ ¬q)"
  shows   "p ∧ q"
proof -
  have "¬p ∨ p" by (rule excluded_middle)
  thus "p ∧ q"
  proof
    assume "¬p"
    have "¬q ∨ q" by (rule excluded_middle)
    thus "p ∧ q"
    proof
      assume "¬q"
      have "¬p ∨ ¬q" using ‹¬p› ‹¬q› by (rule conjI)
      show "p ∧ q" using assms ‹¬p ∨ ¬q› by (rule notE)
```

```
    next
      assume "q"
      thus "p ∨ q" by (rule disjI2)
    qed
  next
    assume "p"
    thus "p ∨ q" by (rule disjI1)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_55_3:
  assumes "¬(¬p ∧ ¬q)"
  shows   "p ∨ q"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 56. Demostrar
     ¬(¬p ∧ ¬q) ⊢ p ∨ q
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_56_1:
  assumes "¬(¬p ∧ ¬q)"
  shows   "p ∨ q"
proof
  show "p"
  proof (rule ccontr)
    assume "¬p"
    hence "¬p ∧ ¬q" ..
    with assms show False ..
  qed
next
  show "q"
  proof (rule ccontr)
    assume "¬q"
    hence "¬p ∧ ¬q" ..
    with assms show False ..
  qed
```

```
qed

-- "La demostración detallada es"
lemma ejercicio_56_2:
  assumes "ň(ňp  ňq)"
  shows   "p  q"
proof (rule conjI)
  show "p"
  proof (rule ccontr)
    assume "ňp"
    hence "ňp  ňq" by (rule disjI1)
    show False using assms ‘ňp  ňq‘ by (rule notE)
  qed
next
  show "q"
  proof (rule ccontr)
    assume "ňq"
    hence "ňp  ňq" by (rule disjI2)
    show False using assms ‘ňp  ňq‘ by (rule notE)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_56_3:
  assumes "ň(ňp  ňq)"
  shows   "p  q"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 57. Demostrar
     ň(p  q)  ňp  ňq
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_57_1:
  assumes "ň(p  q)"
  shows   "ňp  ňq"
proof -
  have "ňp  p" ..
```

```
  thus "¬p ∨ ¬q"
  proof
    assume "¬p"
    thus "¬p ∨ ¬q" ..
  next
    assume "p"
    have "¬q ∨ q" ..
    thus "¬p ∨ ¬q"
    proof
      assume "¬q"
      thus "¬p ∨ ¬q" ..
    next
      assume "q"
      with ‹p› have "p ∧ q" ..
      with assms show "¬p ∨ ¬q" ..
    qed
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_57_2:
  assumes "¬(p ∧ q)"
  shows   "¬p ∨ ¬q"
proof -
  have "¬p ∨ p" by (rule excluded_middle)
  thus "¬p ∨ ¬q"
  proof (rule disjE)
    assume "¬p"
    thus "¬p ∨ ¬q" by (rule disjI1)
  next
    assume "p"
    have "¬q ∨ q" by (rule excluded_middle)
    thus "¬p ∨ ¬q"
    proof
      assume "¬q"
      thus "¬p ∨ ¬q" by (rule disjI2)
    next
      assume "q"
      have "p ∧ q" using ‹p› ‹q› by (rule conjI)
      show "¬p ∨ ¬q" using assms ‹p ∧ q› by (rule notE)
```

```
    qed
  qed
qed

-- "La demostración automática es"
lemma ejercicio_57_3:
  assumes "ň(p  q)"
  shows   "ňp  ňq"
using assms
by auto

text {* ---------------------------------------------------------------
  Ejercicio 58. Demostrar
      (p  q)  (q  p)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma ejercicio_58_1:
  "(p  q)  (q  p)"
proof -
  have "ňp  p" ..
  thus "(p  q)  (q  p)"
  proof
    assume "ňp"
    have "p  q"
    proof
      assume "p"
      with ‘ňp‘ show "q" ..
    qed
    thus "(p  q)  (q  p)" ..
  next
    assume "p"
    have "q  p"
    proof
      assume "q"
      show "p" using ‘p‘ .
    qed
    thus "(p  q)  (q  p)" ..
  qed
qed
```

```
-- "La demostración detallada es"
lemma ejercicio_58_2:
  "(p  q)  (q  p)"
proof -
  have "¬p  p" by (rule excluded_middle)
  thus "(p  q)  (q  p)"
  proof
    assume "¬p"
    have "p  q"
    proof (rule impI)
      assume "p"
      show "q" using ‹¬p› ‹p› by (rule notE)
    qed
    thus "(p  q)  (q  p)" by (rule disjI1)
  next
    assume "p"
    have "q  p"
    proof
      assume "q"
      show "p" using ‹p› by this
    qed
    thus "(p  q)  (q  p)" by (rule disjI2)
  qed
qed

-- "La demostración automática es"
lemma ejercicio_58_3:
  "(p  q)  (q  p)"
by auto

end
```

3.3. Ejercicios: Argumentación lógica proposicional

```
chapter {* T3R2: Argumentación proposicional *}

theory T3R2
imports Main
begin
```

```
text {*
  ----------------------------------------------------------------------
  El objetivo de esta es relación formalizar y demostrar la corrección
  de los argumentos usando sólo las reglas básicas de deducción natural
  de la lógica proposicional (sin usar el método auto).

  Las reglas básicas de la deducción natural son las siguientes:
  ů conjI:      P; Q  P  Q
  ů conjunct1:  P  Q  P
  ů conjunct2:  P  Q  Q
  ů notnotD:    ňň P  P
  ů notnotI:    P  ňň P
  ů mp:         P  Q; P  Q
  ů mt:         F  G; ňG  ňF
  ů impI:       (P  Q)  P  Q
  ů disjI1:     P  P  Q
  ů disjI2:     Q  P  Q
  ů disjE:      P  Q; P  R; Q  R  R
  ů FalseE:     False  P
  ů notE:       ňP; P  R
  ů notI:       (P  False)  ňP
  ů iffI:       P  Q; Q  P  P = Q
  ů iffD1:      Q = P; Q  P
  ů iffD2:      P = Q; Q  P
  ů ccontr:     (ňP  False)  P
  ----------------------------------------------------------------------
*}

text {*
  Se usarán las reglas notnotI y mt que demostramos a continuación.
  *}

lemma notnotI: "P  ňň P"
by auto

lemma mt: "F  G; ňG  ňF"
by auto

text {* ---------------------------------------------------------------
```

```
  Ejercicio 1. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Si no está el mañana ni el ayer escrito, entonces no está el mañana
     escrito.
  Usar M: El mañana está escrito.
       A: El ayer está escrito.
  ---------------------------------------------------------------- *}

-- "La demostración automática es:"
lemma ejercicio_1_1:
  "¬M ∧ ¬A ⟷ ¬M"
by auto

-- "La demostración estructurada es:"
lemma ejercicio_1_2:
  assumes "¬M ∧ ¬A"
  shows   "¬M"
using assms ..

-- "La demostración detallada es:"
lemma ejercicio_1_3:
  assumes "¬M ∧ ¬A"
  shows   "¬M"
using assms by (rule conjunct1)

text {* ----------------------------------------------------------------
  Ejercicio 2. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Siempre que un número x es divisible por 10, acaba en 0. El número
     x no acaba en 0. Por lo tanto, x no es divisible por 10.
  Usar D para "el número es divisible por 10" y
       C para "el número acaba en cero".
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_2_1:
  "C → D; ¬D ⟹ ¬C"
by auto

-- "La demostración estructurada es"
```

```
lemma ejercicio_2_2:
  assumes "C  D"
          "¬D"
  shows   "¬C"
proof
  assume "C"
  with assms(1) have "D" ..
  with assms(2) show False ..
qed

-- "La demostración detallada es"
lemma ejercicio_2_3:
  assumes "C  D"
          "¬D"
  shows   "¬C"
proof
  assume "C"
  with assms(1) have "D" by (rule mp)
  with assms(2) show False by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 3. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Si no hay control de nacimientos, entonces la población crece
     ilimitadamente; pero si la población crece ilimitadamente,
     aumentará el índice de pobreza. Por consiguiente, si no hay control
     de nacimientos, aumentará el índice de pobreza.
  Usar N: Hay control de nacimientos.
       P: La población crece ilimitadamente,
       I: Aumentará el índice de pobreza.
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_3_1:
  "¬N  P; P  I  ¬N  I"
by auto

-- "La demostración estructurada es"
lemma ejercicio_3_2:
```

```
  assumes "ňN  P"
          "P  I"
  shows   "ňN  I"
proof
  assume "ňN"
  with assms(1) have "P" ..
  with assms(2) show "I" ..
qed

-- "La demostración detallada es"
lemma ejercicio_3_3:
  assumes "ňN  P"
          "P  I"
  shows   "ňN  I"
proof (rule impI)
  assume "ňN"
  with assms(1) have "P" by (rule mp)
  with assms(2) show "I" by (rule mp)
qed

text {* ---------------------------------------------------------------
  Ejercicio 4. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Si te llamé por teléfono, entonces recibiste mi llamada y no es
     cierto que no te avisé del peligro que corrías. Por consiguiente,
     como te llamé, es cierto que te avisé del peligro que corrías.
  Usar T: Te llamé por teléfono.
       R: Recibiste mi llamada.
       P: Te avisé del peligro que corrías.
  ---------------------------------------------------------------- *}

-- "La demostración automática es:"
lemma ejercicio_4_1:
  "T  R  ňňA  T  A"
by auto

-- "La demostración estructurada es:"
lemma ejercicio_4_2:
  assumes "T  R  ňňA"
  shows   "T  A"
```

```
proof
  assume "T"
  with assms have "R  ¬¬A" ..
  hence "¬¬A" ..
  thus "A" by (rule notnotD)
qed

-- "La demostración detallada es:"
lemma ejercicio_4_3:
  assumes "T  R  ¬¬A"
  shows   "T  A"
proof (rule impI)
  assume "T"
  with assms(1) have "R  ¬¬A" by (rule mp)
  hence "¬¬A" by (rule conjunct2)
  thus "A" by (rule notnotD)
qed

text {* ---------------------------------------------------------------
  Ejercicio 5. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Si la válvula está abierta o la monitorización está preparada,
     entonces se envía una señal de reconocimiento y un mensaje de
     funcionamiento al controlador del ordenador. Si se envía un mensaje
     de funcionamiento al controlador del ordenador o el sistema está en
     estado normal, entonces se aceptan las órdenes del operador. Por lo
     tanto, si la válvula está abierta, entonces se aceptan las órdenes
     del operador.
  Usar A : La válvula está abierta.
       P : La monitorización está preparada.
       R : Envía una señal de reconocimiento.
       F : Envía un mensaje de funcionamiento.
       N : El sistema está en estado normal.
       O : Se aceptan órdenes del operador.
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_5_1:
  "A  P  R  F; F  N  Or  A  Or"
by auto
```

```
-- "La demostración estructurada es"
lemma ejercicio_5_2:
  assumes "A  P  R  F"
          "F  N  Or"
  shows   "A  Or"
proof
  assume "A"
  hence "A  P" ..
  with assms(1) have "R  F" ..
  hence "F" ..
  hence "F  N" ..
  with assms(2) show "Or" ..
qed

-- "La demostración detallada es"
lemma ejercicio_5_3:
  assumes "A  P  R  F"
          "F  N  Or"
  shows   "A  Or"
proof (rule impI)
  assume "A"
  hence "A  P" by (rule disjI1)
  with assms(1) have "R  F" by (rule mp)
  hence "F" by (rule conjunct2)
  hence "F  N" by (rule disjI1)
  with assms(2) show "Or" by (rule mp)
qed

text {* ------------------------------------------------------------
  Ejercicio 6. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Cuando tanto la temperatura como la presión atmosférica permanecen
     contantes, no llueve. La temperatura permanece constante. Por lo
     tanto, en caso de que llueva, la presión atmosférica no permanece
     constante.
  Usar T para "La temperatura permanece constante",
       P para "La presión atmosférica permanece constante" y
       L para "Llueve".
  ------------------------------------------------------------ *}
```

```
-- "La demostración automática es"
lemma ejercicio_6_1:
  "T  P  ňL; T  L  ňP"
by auto

-- "La demostración estructurada es"
lemma ejercicio_6_2:
  assumes "T  P  ňL"
          "T"
  shows   "L  ňP"
proof
  assume "L"
  show "ňP"
  proof
    assume "P"
    with ‘T‘ have "T  P" ..
    with assms(1) have "ňL" ..
    thus "False" using ‘L‘ ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_6_3:
  assumes "T  P  ňL"
          "T"
  shows   "L  ňP"
proof (rule impI)
  assume "L"
  show "ňP"
  proof
    assume "P"
    with ‘T‘ have "T  P" by (rule conjI)
    with assms(1) have "ňL" by (rule mp)
    thus "False" using ‘L‘ by (rule notE)
  qed
qed

text {* ----------------------------------------------------------
  Ejercicio 7. Formalizar, y demostrar la corrección, del siguiente
```

```
  argumento
     Si el general era leal, hubiera obedecido las órdenes, y si era
     inteligente las hubiera comprendido. O el general desobedeció las
     órdenes o no las comprendió. Luego, el general era desleal o no era
     inteligente.
  Usar L:  El general es leal.
       Ob: El general obedece las órdenes.
       I:  El general es inteligente.
       C:  El general comprende las órdenes.
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_7_1:
  "(L  Ob)  (I  C); ¬Ob  ¬C  ¬L  ¬I"
by auto

-- "La demostración estructurada es"
lemma ejercicio_7_2:
  assumes "(L  Ob)  (I  C)"
          "¬Ob  ¬C"
  shows   "¬L  ¬I"
using assms(2)
proof
  assume "¬Ob"
  have "L  Ob" using assms(1) ..
  hence "¬L" using ‹¬Ob› by (rule mt)
  thus "¬L  ¬I" ..
next
  assume "¬C"
  have "I  C" using assms(1) ..
  hence "¬I" using ‹¬C› by (rule mt)
  thus "¬L  ¬I" ..
qed

-- "La demostración detallada es"
lemma ejercicio_7_3:
  assumes "(L  Ob)  (I  C)"
          "¬Ob  ¬C"
  shows   "¬L  ¬I"
using assms(2)
```

```
proof (rule disjE)
  assume "¬Ob"
  have "L ∨ Ob" using assms(1) by (rule conjunct1)
  hence "¬L" using ‹¬Ob› by (rule mt)
  thus "¬L ∨ ¬I" by (rule disjI1)
next
  assume "¬C"
  have "I ∨ C" using assms(1) by (rule conjunct2)
  hence "¬I" using ‹¬C› by (rule mt)
  thus "¬L ∨ ¬I" by (rule disjI2)
qed

text ‹* ----------------------------------------------------------------
  Ejercicio 8. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Me matan si no trabajo y si trabajo me matan. Me matan siempre me
     matan.
  Usar M: Me matan.
       T: Trabajo.
  ---------------------------------------------------------------- *›

-- "La demostración automática es"
lemma ejercicio_8_1:
  "(¬T → M) ∧ (T → M) → M"
by auto

-- "La demostración estructurada es"
lemma ejercicio_8_2:
  assumes "(¬T → M) ∧ (T → M)"
  shows   "M"
proof -
  have "¬T ∨ T" ..
  thus "M"
  proof
    assume "¬T"
    have "¬T → M" using assms ..
    thus "M" using ‹¬T› ..
  next
    assume "T"
    have "T → M" using assms ..
```

```
    thus "M" using ‘T‘ ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_8_3:
  assumes "(¬T ∨ M) ∧ (T ∨ M)"
  shows   "M"
proof -
  have "¬T ∨ T" by (rule excluded_middle)
  thus "M"
  proof (rule disjE)
    assume "¬T"
    have "¬T ∨ M" using assms by (rule conjunct1)
    thus "M" using ‘¬T‘ by (rule mp)
  next
    assume "T"
    have "T ∨ M" using assms by (rule conjunct2)
    thus "M" using ‘T‘ by (rule mp)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 9. Formalizar, y demostrar la corrección, del siguiente
  argumento
     En cierto experimento, cuando hemos empleado un fármaco A, el
     paciente ha mejorado considerablemente en el caso, y sólo en el
     caso, en que no se haya empleado también un fármaco B. Además, o se
     ha empleado el fármaco A o se ha empleado el fármaco B. En
     consecuencia, podemos afirmar que si no hemos empleado el fármaco
     B, el paciente ha mejorado considerablemente.
  Usar A: Hemos empleado el fármaco A.
       B: Hemos empleado el fármaco B.
       M: El paciente ha mejorado notablemente.
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_9_1:
  assumes "A ∨ (M ∨ ¬B)"
          "A ∨ B"
```

```
  shows   "¬B  M"
using assms
by auto

-- "La demostración estructurada es"
lemma ejercicio_9_2:
  assumes "A  (M  ¬B)"
          "A  B"
  shows   "¬B  M"
proof
  assume "¬B"
  note ‘A  B‘
  hence "A"
  proof
    assume "A"
    thus "A" .
  next
    assume "B"
    with ‘¬B‘ show "A" ..
  qed
  have "M  ¬B" using assms(1) ‘A‘ ..
  thus "M" using ‘¬B‘ ..
qed

-- "La demostración detallada es"
lemma ejercicio_9_3:
  assumes "A  (M  ¬B)"
          "A  B"
  shows   "¬B  M"
proof (rule impI)
  assume "¬B"
  note ‘A  B‘
  hence "A"
  proof (rule disjE)
    assume "A"
    thus "A" by this
  next
    assume "B"
    with ‘¬B‘ show "A" by (rule notE)
  qed
```

```
  have "M  ŏB" using assms(1) `A` by (rule mp)
  thus "M" using `ŏB` by (rule iffD2)
qed

text {* ---------------------------------------------------------------
  Ejercicio 10. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Si trabajo gano dinero, pero si no trabajo gozo de la vida. Sin
     embargo, si trabajo no gozo de la vida, mientras que si no trabajo
     no gano dinero. Por lo tanto, gozo de la vida si y sólo si no gano
  dinero.
  Usar p: Trabajo
       q: Gano dinero.
       r: Gozo de la vida.
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_61_1:
  "(p  q)  (ŏp  r); (p  ŏr)  (ŏp  ŏq)  r  ŏq"
by auto

-- "La demostración estructurada es"
lemma ejercicio_61_2:
  assumes "(p  q)  (ŏp  r)"
          "(p  ŏr)  (ŏp  ŏq)"
  shows   "r  ŏq"
proof
  assume "r"
  hence "ŏŏr" by (rule notnotI)
  have "p  ŏr" using assms(2) ..
  hence "ŏp" using `ŏŏr` by (rule mt)
  have "ŏp  ŏq" using assms(2) ..
  thus "ŏq" using `ŏp` ..
next
  assume "ŏq"
  have "p  q" using assms(1) ..
  hence "ŏp" using `ŏq` by (rule mt)
  have "ŏp  r" using assms(1) ..
  thus "r" using  `ŏp` ..
qed
```

```
-- "La demostración detallada es"
lemma ejercicio_61_3:
  assumes "(p  q)  (ňp  r)"
          "(p  ňr)  (ňp  ňq)"
  shows   "r  ňq"
proof (rule iffI)
  assume "r"
  hence "ňňr" by (rule notnotI)
  have "p  ňr" using assms(2) by (rule conjunct1)
  hence "ňp" using ‘ňňr‘ by (rule mt)
  have "ňp  ňq" using assms(2) by (rule conjunct2)
  thus "ňq" using ‘ňp‘ by (rule mp)
next
  assume "ňq"
  have "p  q" using assms(1) by (rule conjunct1)
  hence "ňp" using ‘ňq‘ by (rule mt)
  have "ňp  r" using assms(1) by (rule conjunct2)
  thus "r" using  ‘ňp‘ by (rule mp)
qed

text {* ---------------------------------------------------------------
  Ejercicio 11. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Si Dios fuera capaz de evitar el mal y quisiera hacerlo, lo
     haría. Si Dios fuera incapaz de evitar el mal, no sería
     omnipotente; si no quisiera evitar el mal sería malévolo. Dios no
     evita el mal. Si Dios existe, es omnipotente y no es
     malévolo. Luego, Dios no existe.
  Usar C:  Dios es capaz de evitar el mal.
       Q:  Dios quiere evitar el mal.
       Om: Dios es omnipotente.
       M:  Dios es malévolo.
       P:  Dios evita el mal.
       E:  Dios existe.
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_11_1:
  "C  Q  P; (ňC  ňOm)  (ňQ  M); ňP; E  Om  ňM  ňE"
```

```
by auto

-- "La demostración estructurada es"
lemma ejercicio_11_2:
  assumes "C  Q  P"
          "(ňC  ňOm)  (ňQ  M)"
          "ňP"
          "E  Om  ňM"
  shows   "ňE"
proof
  assume "E"
  have "Om  ňM" using assms(4) ‘E‘ ..
  hence "Om" ..
  hence "ňňOm" by (rule notnotI)
  have "ňC  ňOm" using assms(2) ..
  hence "ňňC" using ‘ňňOm‘ by (rule mt)
  hence "C" by (rule notnotD)
  have "ňM" using ‘Om  ňM‘ ..
  have "ňQ  M" using assms(2) ..
  hence "ňňQ" using ‘ňM‘ by (rule mt)
  hence "Q" by (rule notnotD)
  with ‘C‘ have "C  Q" ..
  with assms(1) have "P" ..
  with assms(3) show False ..
qed

-- "La demostración detallada es"
lemma ejercicio_11_3:
  assumes "C  Q  P"
          "(ňC  ňOm)  (ňQ  M)"
          "ňP"
          "E  Om  ňM"
  shows   "ňE"
proof (rule notI)
  assume "E"
  with assms(4) have "Om  ňM" by (rule mp)
  hence "Om" by (rule conjunct1)
  hence "ňňOm" by (rule notnotI)
  have "ňC  ňOm" using assms(2) by (rule conjunct1)
  hence "ňňC" using ‘ňňOm‘ by (rule mt)
```

```
  hence "C" by (rule notnotD)
  have "¬M" using ‘Om  ¬M‘ by (rule conjunct2)
  have "¬Q  M" using assms(2) by (rule conjunct2)
  hence "¬¬Q" using ‘¬M‘ by (rule mt)
  hence "Q" by (rule notnotD)
  with ‘C‘ have "C  Q" by (rule conjI)
  with assms(1) have "P" by (rule mp)
  with assms(3) show False by (rule notE)
qed

end
```

3.4. Ejercicios: Eliminación de conectivas

```
chapter {* T3R3: Eliminación de conectivas *}

theory T3R3
imports Main
begin

text {*
  ----------------------------------------------------------------------
  El objetivo de esta es relación es demostrar cómo a partir de las
  conectivas False,  y  pueden definirse las restantes.
  ----------------------------------------------------------------------
*}


text {* ----------------------------------------------------------------
  Ejercicio 1. Definir  usando  y ; es decir, sustituir en
      (A  B) = indefinida
  la indefinida por una fórmula que sólo usa las conectivas  y  y
  demostrar la equivalencia.
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma bicondicional: "
  (A  B) = ((A  B)  (B  A))"
by auto
```

```
-- "La demostración estructurada es"
lemma bicondicional_2: "
  (A  B) = ((A  B)  (B  A))"
proof
  assume "A  B"
  show "(A  B)  (B  A)"
  proof
    show "A  B"
    proof
      assume "A"
      with ‘A  B‘ show "B" ..
    qed
  next
    show "B  A"
    proof
      assume "B"
      with ‘A  B‘ show "A" ..
    qed
  qed
next
  assume 1: "(A  B)  (B  A)"
  show "A  B"
  proof
    assume "A"
    have "A  B"using 1  ..
    thus "B" using ‘A‘ ..
  next
    assume "B"
    have "B  A"using 1  ..
    thus "A" using ‘B‘ ..
  qed
qed

-- "La demostración detallada es"
lemma bicondicional_3: "
  (A  B) = ((A  B)  (B  A))"
proof (rule iffI)
  assume "A  B"
  show "(A  B)  (B  A)"
  proof (rule conjI)
```

```
    show "A  B"
    proof (rule impI)
      assume "A"
      with `A  B` show "B" by (rule iffD1)
    qed
  next
    show "B  A"
    proof (rule impI)
      assume "B"
      with `A  B` show "A" by (rule iffD2)
    qed
  qed
next
  assume 1: "(A  B)  (B  A)"
  show "A  B"
  proof (rule iffI)
    assume "A"
    have "A  B"using 1  by (rule conjunct1)
    thus "B" using `A` by (rule mp)
  next
    assume "B"
    have "B  A"using 1  by (rule conjunct2)
    thus "A" using `B` by (rule mp)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 2. Definir ň usando  y False; es decir, sustituir en
      (ňA) = indefinida
  la indefinida por una fórmula que sólo usa las conectivas  y False
  y demostrar la equivalencia.
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma negacion:
  "(ňA) = (A  False)"
by auto

-- "La demostración estructurada es"
lemma negacion_2:
```

```
  "(¬A) = (A  False)"
proof
  assume "¬A"
  show "A  False"
  proof
    assume "A"
    with ‘¬A‘ show False ..
  qed
next
  assume "A  False"
  show "¬A"
  proof
    assume "A"
    with ‘A  False‘ show False ..
  qed
qed

-- "La demostración detallada es"
lemma negacion_3:
  "(¬A) = (A  False)"
proof (rule iffI)
  assume "¬A"
  show "A  False"
  proof (rule impI)
    assume "A"
    with ‘¬A‘ show False by (rule notE)
  qed
next
  assume "A  False"
  show "¬A"
  proof (rule notI)
    assume "A"
    with ‘A  False‘ show False by (rule mp)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 3. Definir  usando  y False; es decir, sustituir en
      (A  B) = indefinida
  la indefinida por una fórmula que sólo usa las conectivas  y False
```

```
  y demostrar la equivalencia.
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma disyuncion:
  "(A  B) = ((A  False)  B)"
by auto

-- "La demostración estructurada es"
lemma disyuncion_2:
  "(A  B) = ((A  False)  B)"
proof
  assume "A  B"
  show "(A  False)  B"
  proof
    assume "A  False"
    note ‘A  B‘
    thus "B"
    proof
      assume "A"
      with ‘A  False‘ have False ..
      thus "B" ..
    next
      assume "B"
      thus "B" .
    qed
  qed
next
  assume "(A  False)  B"
  show "A  B"
  proof -
    have "¬A  A" ..
    thus "A  B"
    proof
      assume "¬A"
      have "A  False"
      proof
        assume "A"
        with ‘¬A‘ show False ..
      qed
```

```
      with ‘(A  False)  B‘ have "B" ..
      thus "A  B" ..
    next
      assume "A"
      thus "A  B" ..
    qed
  qed
qed

-- "La demostración detallada es"
lemma disyuncion_3:
  "(A  B) = ((A  False)  B)"
proof (rule iffI)
  assume "A  B"
  show "(A  False)  B"
  proof (rule impI)
    assume "A  False"
    note ‘A  B‘
    thus "B"
    proof (rule disjE)
      assume "A"
      with ‘A  False‘ have False by (rule mp)
      thus "B" by (rule FalseE)
    next
      assume "B"
      thus "B" by this
    qed
  qed
next
  assume "(A  False)  B"
  show "A  B"
  proof -
    have "¬A  A" by (rule excluded_middle)
    thus "A  B"
    proof (rule disjE)
      assume "¬A"
      have "A  False"
      proof (rule impI)
        assume "A"
        with ‘¬A‘ show False by (rule notE)
```

```
      qed
      with ‘(A  False)  B‘ have "B" by (rule mp)
      thus "A  B" by (rule disjI2)
    next
      assume "A"
      thus "A  B" by (rule disjI1)
    qed
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 4. Encontrar una fórmula equivalente a
     (A  (B  C))  A
  que sólo use las conectivas False,  y  y demostrar la
  equivalencia.
  ---------------------------------------------------------------- *}

-- "Se puede buscar la fórmula con"
lemma ejercicio_4a:
  "(A  (B  C))  A"
apply (simp only: bicondicional disyuncion)
oops

-- "La fórmula obtenida es"
-- "(((A  False)  B  C)  A)  (A  (A  False)  B  C)"

-- "La demostración de la equivalencia es"
lemma ejercicio_4b:
  "((A  (B  C))  A)
   ((((A  False)  B  C)  A)  (A  (A  False)  B  C))"
by (auto simp add: bicondicional disyuncion)

end
```

Capítulo 4

Deducción natural en lógica de primer orden

4.1. Tema: Deducción natural en lógica de primer orden

```
chapter {* Tema 4: Deducción natural en lógica de primer orden *}

theory T4
imports Main
begin

text {*
  El objetivo de este tema es presentar la deducción natural en
  lógica de primer orden con Isabelle/HOL. La presentación se
  basa en los ejemplos de tema 2 del curso LMF que se encuentra
  en http://goo.gl/uJj8d (que a su vez se basa en el libro de
  Huth y Ryan "Logic in Computer Science" http://goo.gl/qsVpY ).

  La página al lado de cada ejemplo indica la página de las
  transparencias de LMF donde se encuentra la demostración. *}

section {* Reglas del cuantificador universal *}

text {*
  Las reglas del cuantificador universal son
  ů allE:    x. P x; P a  R  R
  ů allI:    (x. P x)  x. P x
  *}
```

```
text {*
  Ejemplo 1 (p. 10). Demostrar que
     P(c), x. (P(x)  ¬Q(x))  ¬Q(c)
*}

-- "La demostración detallada es"
lemma ejemplo_1a:
  assumes 1: "P(c)" and
          2: "x. (P(x)  ¬Q(x))"
  shows "¬Q(c)"
proof -
  have 3: "P(c)  ¬Q(c)" using 2 by (rule allE)
  show 4: "¬Q(c)" using 3 1 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejemplo_1b:
  assumes "P(c)"
          "x. (P(x)  ¬Q(x))"
  shows "¬Q(c)"
proof -
  have "P(c)  ¬Q(c)" using assms(2) ..
  thus "¬Q(c)" using assms(1) ..
qed

-- "La demostración automática es"
lemma ejemplo_1c:
  assumes "P(c)"
          "x. (P(x)  ¬Q(x))"
  shows "¬Q(c)"
using assms
by auto

text {*
  Ejemplo 2 (p. 11). Demostrar que
     x. (P x  ¬(Q x)), x. P x  x. ¬(Q x)
*}

-- "La demostración detallada es"
lemma ejemplo_2a:
```

```
  assumes 1: "∀x. (P x ⟶ ¬(Q x))" and
          2: "∀x. P x"
  shows "∀x. ¬(Q x)"
proof -
  { fix a
    have 3: "P a ⟶ ¬(Q a)" using 1 by (rule allE)
    have 4: "P a" using 2 by (rule allE)
    have 5: "¬(Q a)" using 3 4 by (rule mp) }
  thus "∀x. ¬(Q x)" by (rule allI)
qed

-- "La demostración detallada hacia atrás es"
lemma ejemplo_2b:
  assumes 1: "∀x. (P x ⟶ ¬(Q x))" and
          2: "∀x. P x"
  shows "∀x. ¬(Q x)"
proof (rule allI)
  fix a
  have 3: "P a ⟶ ¬(Q a)" using 1 by (rule allE)
  have 4: "P a" using 2 by (rule allE)
  show 5: "¬(Q a)" using 3 4 by (rule mp)
qed

-- "La demostración estructurada es"
lemma ejemplo_2c:
  assumes "∀x. (P x ⟶ ¬(Q x))"
          "∀x. P x"
  shows "∀x. ¬(Q x)"
proof
  fix a
  have "P a" using assms(2) ..
  have "P a ⟶ ¬(Q a)" using assms(1) ..
  thus "¬(Q a)" using `P a` ..
qed

-- "La demostración automática es"
lemma ejemplo_2d:
  assumes "∀x. (P x ⟶ ¬(Q x))"
          "∀x. P x"
  shows   "∀x. ¬(Q x)"
```

```
using assms
by auto

section {* Reglas del cuantificador existencial *}

text {*
  Las reglas del cuantificador existencial son
  å exI:     P a  x. P x
  å exE:     x. P x; x. P x  Q  Q

  En la regla exE la nueva variable se introduce mediante la declaración
  "obtain ... where ... by (rule exE)"
  *}

text {*
  Ejemplo  (p. 12). Demostrar que
     x. P x  x. P x
*}

-- "La demostración detallada es"
lemma ejemplo_3a:
  assumes "x. P x"
  shows "x. P x"
proof -
  fix a
  have "P a" using assms by (rule allE)
  thus "x. P x" by (rule exI)
qed

-- "La demostración estructurada es"
lemma ejemplo_3b:
  assumes "x. P x"
  shows "x. P x"
proof -
  fix a
  have "P a" using assms ..
  thus "x. P x" ..
qed

-- "La demostración estructurada se puede simplificar"
```

```
lemma ejemplo_3c:
  assumes "∃x. P x"
  shows "∃x. P x"
proof (rule exI)
  fix a
  show "P a" using assms ..
qed

-- "La demostración estructurada se puede simplificar aún más"
lemma ejemplo_3d:
  assumes "∃x. P x"
  shows "∃x. P x"
proof
  fix a
  show "P a" using assms ..
qed

-- "La demostración automática es"
lemma ejemplo_3e:
  assumes "∃x. P x"
  shows "∃x. P x"
using assms
by auto

text {*
  Ejemplo 4 (p. 13). Demostrar
     ∃x. (P x ∧ Q x), ∀x. P x → ∀x. Q x
*}

-- "La demostración detallada es"
lemma ejemplo_4a:
  assumes 1: "∃x. (P x ∧ Q x)" and
          2: "∀x. P x"
  shows "∀x. Q x"
proof -
  obtain a where 3: "P a" using 2 by (rule exE)
  have 4: "P a ∧ Q a" using 1 by (rule allE)
  have 5: "Q a" using 4 3 by (rule mp)
  thus 6: "∀x. Q x" by (rule exI)
qed
```

```
-- "La demostración estructurada es"
lemma ejemplo_4b:
  assumes "∀x. (P x ∧ Q x)"
          "∀x. P x"
  shows "∀x. Q x"
proof -
  obtain a where "P a" using assms(2) ..
  have "P a ∧ Q a" using assms(1) ..
  hence "Q a" using ‘P a‘ ..
  thus "∀x. Q x" ..
qed

-- "La demostración automática es"
lemma ejemplo_4c:
  assumes "∀x. (P x ∧ Q x)"
          "∀x. P x"
  shows "∀x. Q x"
using assms
by auto

section {* Demostración de equivalencias *}

text {*
  Ejemplo 5.1 (p. 15). Demostrar
     ¬∀x. P x ⟷ ∃x. ¬(P x) *}

-- "La demostración detallada es"
lemma ejemplo_5_1a:
  assumes "¬(∀x. P(x))"
  shows   "∃x. ¬P(x)"
proof (rule ccontr)
  assume "¬(∃x. ¬P(x))"
  have "∀x. P(x)"
  proof (rule allI)
    fix a
    show "P(a)"
    proof (rule ccontr)
      assume "¬P(a)"
      hence "∃x. ¬P(x)" by (rule exI)
```

```
      with ‘ň(x. ňP(x))‘ show False by (rule notE)
    qed
  qed
  with assms show False by (rule notE)
qed

-- "La demostración estructurada es"
lemma ejemplo_5_1b:
  assumes "ň(x. P(x))"
  shows   "x. ňP(x)"
proof (rule ccontr)
  assume "ň(x. ňP(x))"
  have "x. P(x)"
  proof
    fix a
    show "P(a)"
    proof (rule ccontr)
      assume "ňP(a)"
      hence "x. ňP(x)" ..
      with ‘ň(x. ňP(x))‘ show False ..
    qed
  qed
  with assms show False ..
qed

-- "La demostración automática es"
lemma ejemplo_5_1c:
  assumes "ň(x. P(x))"
  shows   "x. ňP(x)"
using assms
by auto

text {*
  Ejemplo 5.2 (p. 16). Demostrar
     x. ň(P x)   ňx. P x *}

-- "La demostración detallada es"
lemma ejemplo_5_2a:
  assumes "x. ňP(x)"
  shows   "ň(x. P(x))"
```

```
proof (rule notI)
  assume "∀x. P(x)"
  obtain a where "¬P(a)" using assms by (rule exE)
  have "P(a)" using ‘∀x. P(x)‘ by (rule allE)
  with ‘¬P(a)‘ show False by (rule notE)
qed

-- "La demostración estructurada es"
lemma ejemplo_5_2b:
  assumes "∀x. ¬P(x)"
  shows   "¬(∀x. P(x))"
proof
  assume "∀x. P(x)"
  obtain a where "¬P(a)" using assms ..
  have "P(a)" using ‘∀x. P(x)‘ ..
  with ‘¬P(a)‘ show False ..
qed

-- "La demostración automática es"
lemma ejemplo_5_2c:
  assumes "∀x. ¬P(x)"
  shows   "¬(∀x. P(x))"
using assms
by auto

text {*
  Ejemplo 5.3 (p. 17). Demostrar
      ¬∀x. P x  ⟷ ∃x. ¬(P x) *}

-- "La demostración detallada es"
lemma ejemplo_5_3a:
  "(¬(∀x. P(x))) ⟷ (∃x. ¬P(x))"
proof (rule iffI)
  assume "¬(∀x. P(x))"
  thus "∃x. ¬P(x)" by (rule ejemplo_5_1a)
next
  assume "∃x. ¬P(x)"
  thus "¬(∀x. P(x))" by (rule ejemplo_5_2a)
qed
```

```
-- "La demostración automática es"
lemma ejemplo_5_3b:
  "(ň(x. P(x)))  (x. ňP(x))"
by auto

text {*
  Ejemplo 6.1 (p. 18). Demostrar
     x. P(x)  Q(x)   (x. P(x))  (x. Q(x)) *}

-- "La demostración detallada es"
lemma ejemplo_6_1a:
  assumes "x. P(x)  Q(x)"
  shows   "(x. P(x))  (x. Q(x))"
proof (rule conjI)
  show "x. P(x)"
  proof (rule allI)
    fix a
    have "P(a)  Q(a)" using assms by (rule allE)
    thus "P(a)" by (rule conjunct1)
  qed
next
  show "x. Q(x)"
  proof (rule allI)
    fix a
    have "P(a)  Q(a)" using assms by (rule allE)
    thus "Q(a)" by (rule conjunct2)
  qed
qed

-- "La demostración estructurada es"
lemma ejemplo_6_1b:
  assumes "x. P(x)  Q(x)"
  shows   "(x. P(x))  (x. Q(x))"
proof
  show "x. P(x)"
  proof
    fix a
    have "P(a)  Q(a)" using assms ..
    thus "P(a)" ..
  qed
```

```
next
  show "∀x. Q(x)"
  proof
    fix a
    have "P(a) ⟶ Q(a)" using assms ..
    thus "Q(a)" ..
  qed
qed

-- "La demostración automática es"
lemma ejemplo_6_1c:
  assumes "∀x. P(x) ⟶ Q(x)"
  shows   "(∀x. P(x)) ⟶ (∀x. Q(x))"
using assms
by auto

text {*
  Ejemplo 6.2 (p. 19). Demostrar
     (∀x. P(x)) ⟶ (∀x. Q(x)) ⟷ ∀x. P(x) ⟶ Q(x)  *}

-- "La demostración detallada es"
lemma ejemplo_6_2a:
  assumes "(∀x. P(x)) ⟶ (∀x. Q(x))"
  shows   "∀x. P(x) ⟶ Q(x)"
proof (rule allI)
  fix a
  have "∀x. P(x)" using assms by (rule conjunct1)
  hence "P(a)" by (rule allE)
  have "∀x. Q(x)" using assms by (rule conjunct2)
  hence "Q(a)" by (rule allE)
  with ‘P(a)‘ show "P(a) ⟶ Q(a)" by (rule conjI)
qed

-- "La demostración estructurada es"
lemma ejemplo_6_2b:
  assumes "(∀x. P(x)) ⟶ (∀x. Q(x))"
  shows   "∀x. P(x) ⟶ Q(x)"
proof
  fix a
  have "∀x. P(x)" using assms ..
```

```
  hence "P(a)" by (rule allE)
  have "x. Q(x)" using assms ..
  hence "Q(a)" ..
  with ‘P(a)‘ show "P(a)  Q(a)" ..
qed

-- "La demostración automática es"
lemma ejemplo_6_2c:
  assumes "(x. P(x))  (x. Q(x))"
  shows   "x. P(x)  Q(x)"
using assms
by auto

text {*
  Ejemplo 6.3 (p. 20). Demostrar
      x. P(x)  Q(x)  (x. P(x))  (x. Q(x)) *}

-- "La demostración detallada es"
lemma ejemplo_6_3a:
  "(x. P(x)  Q(x))  ((x. P(x))  (x. Q(x)))"
proof (rule iffI)
  assume "x. P(x)  Q(x)"
  thus "(x. P(x))  (x. Q(x))" by (rule ejemplo_6_1a)
next
  assume "(x. P(x))  (x. Q(x))"
  thus "x. P(x)  Q(x)" by (rule ejemplo_6_2a)
qed

text {*
  Ejemplo 7.1 (p. 21). Demostrar
     (x. P(x))  (x. Q(x))  x. P(x)  Q(x)  *}

-- "La demostración detallada es"
lemma ejemplo_7_1a:
  assumes "(x. P(x))  (x. Q(x))"
  shows   "x. P(x)  Q(x)"
using assms
proof (rule disjE)
  assume "x. P(x)"
  then obtain a where "P(a)" by (rule exE)
```

```
  hence "P(a)  Q(a)" by (rule disjI1)
  thus "x. P(x)  Q(x)" by (rule exI)
next
  assume "x. Q(x)"
  then obtain a where "Q(a)" by (rule exE)
  hence "P(a)  Q(a)" by (rule disjI2)
  thus "x. P(x)  Q(x)" by (rule exI)
qed

-- "La demostración estructurada es"
lemma ejemplo_7_1b:
  assumes "(x. P(x))  (x. Q(x))"
  shows   "x. P(x)  Q(x)"
using assms
proof
  assume "x. P(x)"
  then obtain a where "P(a)" ..
  hence "P(a)  Q(a)" ..
  thus "x. P(x)  Q(x)" ..
next
  assume "x. Q(x)"
  then obtain a where "Q(a)" ..
  hence "P(a)  Q(a)" ..
  thus "x. P(x)  Q(x)" ..
qed

-- "La demostración automática es"
lemma ejemplo_7_1c:
  assumes "(x. P(x))  (x. Q(x))"
  shows   "x. P(x)  Q(x)"
using assms
by auto

text {*
  Ejemplo 7.2 (p. 22). Demostrar
     x. P(x)  Q(x)  (x. P(x))  (x. Q(x))  *}

-- "La demostración detallada es"
lemma ejemplo_7_2a:
  assumes "x. P(x)  Q(x)"
```

```
  shows   "(x. P(x))  (x. Q(x))"
proof -
  obtain a where "P(a)  Q(a)" using assms by (rule exE)
  thus "(x. P(x))  (x. Q(x))"
  proof (rule disjE)
    assume "P(a)"
    hence "x. P(x)" by (rule exI)
    thus "(x. P(x))  (x. Q(x))" by (rule disjI1)
  next
    assume "Q(a)"
    hence "x. Q(x)" by (rule exI)
    thus "(x. P(x))  (x. Q(x))" by (rule disjI2)
  qed
qed

-- "La demostración estructurada es"
lemma ejercicio_7_2b:
  assumes "x. P(x)  Q(x)"
  shows   "(x. P(x))  (x. Q(x))"
proof -
  obtain a where "P(a)  Q(a)" using assms ..
  thus "(x. P(x))  (x. Q(x))"
  proof
    assume "P(a)"
    hence "x. P(x)" ..
    thus "(x. P(x))  (x. Q(x))" ..
  next
    assume "Q(a)"
    hence "x. Q(x)" ..
    thus "(x. P(x))  (x. Q(x))" ..
  qed
qed

-- "La demostración automática es"
lemma ejercicio_7_2c:
  assumes "x. P(x)  Q(x)"
  shows   "(x. P(x))  (x. Q(x))"
using assms
by auto
```

```
text {*
  Ejemplo 7.3 (p. 23). Demostrar
      ((x. P(x))  (x. Q(x)))  (x. P(x)  Q(x))  *}

-- "La demostración detallada es"
lemma ejemplo_7_3a:
  "((x. P(x))  (x. Q(x)))  (x. P(x)  Q(x))"
proof (rule iffI)
  assume "(x. P(x))  (x. Q(x))"
  thus "x. P(x)  Q(x)" by (rule ejemplo_7_1a)
next
  assume "x. P(x)  Q(x)"
  thus "(x. P(x))  (x. Q(x))" by (rule ejemplo_7_2a)
qed

-- "La demostración automática es"
lemma ejemplo_7_3b:
  "((x. P(x))  (x. Q(x)))  (x. P(x)  Q(x))"
using assms
by auto

text {*
  Ejemplo 8.1 (p. 24). Demostrar
     x y. P(x,y)  y x. P(x,y)  *}

-- "La demostración detallada es"
lemma ejemplo_8_1a:
  assumes "x y. P(x,y)"
  shows   "y x. P(x,y)"
proof -
  obtain a where "y. P(a,y)" using assms by (rule exE)
  then obtain b where "P(a,b)" by (rule exE)
  hence "x. P(x,b)" by (rule exI)
  thus "y x. P(x,y)" by (rule exI)
qed

-- "La demostración estructurada es"
lemma ejemplo_8_1b:
  assumes "x y. P(x,y)"
  shows   "y x. P(x,y)"
```

```
proof -
  obtain a where "y. P(a,y)" using assms ..
  then obtain b where "P(a,b)" ..
  hence "x. P(x,b)" ..
  thus "y x. P(x,y)" ..
qed

-- "La demostración automática es"
lemma ejemplo_8_1c:
  assumes "x y. P(x,y)"
  shows   "y x. P(x,y)"
using assms
by auto

text {*
  Ejemplo 8.2. Demostrar
     y x. P(x,y)  x y. P(x,y)  *}

-- "La demostración detallada es"
lemma ejemplo_8_2a:
  assumes "y x. P(x,y)"
  shows   "x y. P(x,y)"
proof -
  obtain b where "x. P(x,b)" using assms by (rule exE)
  then obtain a where "P(a,b)" by (rule exE)
  hence "y. P(a,y)" by (rule exI)
  thus "x y. P(x,y)" by (rule exI)
qed

-- "La demostración estructurada es"
lemma ejemplo_8_2b:
  assumes "y x. P(x,y)"
  shows   "x y. P(x,y)"
proof -
  obtain b where "x. P(x,b)" using assms ..
  then obtain a where "P(a,b)" ..
  hence "y. P(a,y)" ..
  thus "x y. P(x,y)" ..
qed
```

```
-- "La demostración estructurada es"
lemma ejemplo_8_2c:
  assumes "∀y x. P(x,y)"
  shows   "∀x y. P(x,y)"
using assms
by auto

text {*
  Ejemplo 8.3 (p. 25). Demostrar
      (∀x y. P(x,y)) → (∀y x. P(x,y)) *}

-- "La demostración detallada es"
lemma ejemplo_8_3a:
  "(∀x y. P(x,y)) → (∀y x. P(x,y))"
proof (rule impI)
  assume "∀x y. P(x,y)"
  thus "∀y x. P(x,y)" by (rule ejemplo_8_1a)
next
  assume "∀y x. P(x,y)"
  thus "∀x y. P(x,y)" by (rule ejemplo_8_2a)
qed

-- "La demostración automática es"
lemma ejemplo_8_3b:
  "(∀x y. P(x,y)) → (∀y x. P(x,y))"
by auto

section {* Reglas de la igualdad *}

text {*
  Las reglas básicas de la igualdad son:
  • refl:  t = t
  • subst: s = t; P s → P t
*}

text {*
  Ejemplo 9 (p. 27). Demostrar
     x+1 = 1+x, x+1 > 1 → x+1 > 0 → 1+x > 1 → 1+x > 0
*}
```

```
-- "La demostración detallada es"
lemma ejemplo_9a:
  assumes "x+1 = 1+x"
          "x+1 > 1  x+1 > 0"
  shows   "1+x > 1  1+x > 0"
proof -
  show "1+x > 1  1+x > 0" using assms by (rule subst)
qed

-- "La demostración estructurada es"
lemma ejemplo_9b:
  assumes "x+1 = 1+x"
          "x+1 > 1  x+1 > 0"
  shows   "1+x > 1  1+x > 0"
using assms
by (rule subst)

-- "La demostración automática es"
lemma ejemplo_9c:
  assumes "x+1 = 1+x"
          "x+1 > 1  x+1 > 0"
  shows   "1+x > 1  1+x > 0"
using assms
by auto

text {*
  Ejemplo 10 (p. 27). Demostrar
     x = y, y = z  x = z
*}

-- "La demostración detallada es"
lemma ejemplo_10a:
  assumes "x = y"
          "y = z"
  shows   "x = z"
proof -
  show "x = z" using assms(2,1) by (rule subst)
qed

-- "La demostración estructurada es"
```

```
lemma ejemplo_10b:
  assumes "x = y"
          "y = z"
  shows   "x = z"
using assms(2,1)
by (rule subst)

-- "La demostración automática es"
lemma ejemplo_10c:
  assumes "x = y"
          "y = z"
  shows   "x = z"
using assms
by auto

text {*
  Ejemplo 11 (p. 28). Demostrar
     s = t  t = s
*}

-- "La demostración detallada es"
lemma ejemplo_11a:
  assumes "s = t"
  shows   "t = s"
proof -
  have "s = s" by (rule refl)
  with assms show "t = s" by (rule subst)
qed

-- "La demostración automática es"
lemma ejemlo_11b:
  assumes "s = t"
  shows   "t = s"
using assms
by auto

text {*
  Algunas reglas derivadas de la igualdad son:
  ů trans:      r = s; s = t  r = t
  ů sym:        s = t  t = s
```

```
  ů not_sym:     t  s  s  t
  ů ssubst:      t = s; P s  P t
  ů box_equals: a = b; a = c; b = d  c = d
  ů arg_cong:    x = y  f x = f y
  ů fun_cong:    f = g  f x = g x
  ů cong:        f = g; x = y  f x = g y
*}

-- "La demostración estructurada de not_sym es"
lemma not_sym_1:
  assumes "t  s"
  shows   "s  t"
proof
  assume "s = t"
  hence "t = s" ..
  show False using assms(1) `t = s` ..
qed

-- "La demostración detallada de not_sym es"
lemma not_sym_2:
  assumes "t  s"
  shows   "s  t"
proof (rule notI)
  assume "s = t"
  hence "t = s" by (rule sym)
  show False using assms(1) `t = s` by (rule notE)
qed

-- "La demostración automática de not_sym es"
lemma not_sym_3:
  assumes "t  s"
  shows   "s  t"
using assms
by auto

-- "La demostración estructurada de ssubst es"
lemma sssubs_1:
  assumes "t = s"
          "P s"
  shows   "P t"
```

```
proof -
  have "s = t" using assms(1) ..
  thus "P t" using assms(2) by (rule subst)
qed

-- "La demostración detallada de ssubst es"
lemma sssubs_2:
  assumes "t = s"
          "P s"
  shows   "P t"
proof -
  have "s = t" using assms(1) by (rule sym)
  thus "P t" using assms(2) by (rule subst)
qed

-- "La demostración automática de ssubst es"
lemma ssubst_3:
  assumes "t = s"
          "P s"
  shows   "P t"
using assms
by auto

-- "La demostración detallada de box_equals es"
lemma box_equals_1:
  assumes "a = b"
          "a = c"
          "b = d"
  shows   "c = d"
proof -
  have "c = b" using assms(2,1) by (rule subst)
  with assms(3) show "c = d" by (rule subst)
qed

-- "La demostración semiautomática de box_equals es"
lemma box_equals_2:
  assumes "a = b"
          "a = c"
          "b = d"
  shows   "c = d"
```

```
using assms
by (rule box_equals)

-- "La demostración automática de box_equals es"
lemma box_equals_3:
  assumes "a = b"
          "a = c"
          "b = d"
  shows   "c = d"
using assms
by auto

-- "La demostración detallada de arg_cong es"
lemma arg_cong_1:
  assumes "x = y"
  shows   "f x = f y"
proof -
  have "f x = f x" by (rule refl)
  with assms(1) show "f x = f y" by (rule subst)
qed

-- "La demostración semiautomática de arg_cong es"
lemma arg_cong_2:
  assumes "x = y"
  shows   "f x = f y"
using assms
by (rule arg_cong)

-- "La demostración automática de arg_cong es"
lemma arg_cong_3:
  assumes "x = y"
  shows   "f x = f y"
using assms
by auto

-- "La demostración detallada de fun_cong es"
lemma fun_cong_1:
  assumes "f = g"
  shows   "f x = g x"
proof -
```

```
  have "f x = f x" by (rule refl)
  with assms(1) show "f x = g x" by (erule_tac P = "h. f x = h x" in subst)
qed

-- "La demostración semiautomática de fun_cong es"
lemma fun_cong_2:
  assumes "f = g"
  shows   "f x = g x"
using assms
by (rule fun_cong)

-- "La demostración automática de fun_cong es"
lemma fun_cong_3:
  assumes "f = g"
  shows   "f x = g x"
using assms
by auto

-- "La demostración detallada de cong es"
lemma cong_1:
  assumes "f = g"
          "x = y"
  shows   "f x = g y"
proof -
  have "f x = f x" by (rule refl)
  with assms(2) have "f x = f y" by (rule subst)
  with assms(1) show "f x = g y" by (erule_tac P = "h. f x = h y" in subst)
qed

-- "La demostración semiautomática de cong es"
lemma cong_2:
  assumes "f = g"
          "x = y"
  shows   "f x = g y"
using assms
by (rule cong)

-- "La demostración automática de cong es"
lemma cong_3:
  assumes "f = g"
```

```
          "x = y"
  shows   "f x = g y"
using assms
by auto

section {* Ejemplo de razonamiento sobre programa *}

text {*
  Definición. El número natural x divide al número natural y si existe
  un natural k tal que kx = y.
*}

definition divide :: "nat  nat  bool" where
  "divide x y  k. k*x = y"

text {*
  La definición de divide se añade a las reglas de simplificación.
*}

declare divide_def[simp]

text {*
  Ejemplo 8 [Transitividad de la divisibilidad]. Sean a, b y c números
  naturales. Si b es divisible por a y c es divisible por b, entonces c
  es divisible por a.
*}

-- "La demostración estructurada es"
lemma ejemplo_8_1:
  assumes "divide a b"
          "divide b c"
  shows   "divide a c"
proof -
  obtain m where "m*a = b" using assms(1) by auto
  obtain n where "n*b = c" using assms(2) by auto
  hence "m*n*a = c" using `m*a = b` by auto
  hence "k. k*a = c" by (rule exI)
  thus "divide a c" by simp
qed
```

```
-- "La demostración puede simplificarse"
lemma ejemplo_8_2:
  assumes "divide a b"
          "divide b c"
  shows   "divide a c"
proof simp
  obtain m where "m*a = b" using assms(1) by auto
  obtain n where "n*b = c" using assms(2) by auto
  hence "m*n*a = c" using `m*a = b` by auto
  thus "k. k*a = c" ..
qed

-- "La demostración automática es"
lemma ejemplo_8_3:
  assumes "divide a b"
          "divide b c"
  shows   "divide a c"
using assms
by auto

section {* Razonamiento ecuacional *}

text {*
  El razonamiento ecuacional se realiza usando la combinación de "also"
  (además) y "finally" (finalmente).
*}

text {*
  Ejemplo 9 [Razonamiento ecuacional]. Si a=b, b=c y c=d, entonces a=d.
*}

-- "La demostración detallada es"
lemma ejemplo_9_1:
  assumes "a = b"
          "b = c"
          "c = d"
  shows   "a = d"
proof -
  have "a = b" by (rule assms(1))
  also have " = c" by (rule assms(2))
```

```
  also have " = d" by (rule assms(3))
  finally show "a = d" .
qed

-- "La demostración automática es"
lemma  ejemplo_9_2:
  assumes "a = b"
          "b = c"
          "c = d"
  shows   "a = d"
using assms
by metis

end
```

4.2. Ejercicios: Deducción natural en lógica de primer orden

```
chapter {* T4R1: Deducción natural de primer orden *}

theory T4R1
imports Main
begin

text {*
  Demostrar o refutar los siguientes lemas usando sólo las reglas
  básicas de deducción natural de la lógica proposicional, de los
  cuantificadores y de la igualdad:
  ů conjI:      P; Q  P  Q
  ů conjunct1:  P  Q  P
  ů conjunct2:  P  Q  Q
  ů notnotD:    ňň P  P
  ů mp:         P  Q; P  Q
  ů impI:       (P  Q)  P  Q
  ů disjI1:     P  P  Q
  ů disjI2:     Q  P  Q
  ů disjE:      P  Q; P  R; Q  R  R
  ů FalseE:     False  P
  ů notE:       ňP; P  R
```

```
  ů notI:        (P  False)  ňP
  ů iffI:        P  Q; Q  P  P = Q
  ů iffD1:       Q = P; Q  P
  ů iffD2:       P = Q; Q  P
  ů ccontr:      (ňP  False)  P

  ů allI:        x. P x; P x  R  R
  ů allE:        (x. P x)  x. P x
  ů exI:         P x  x. P x
  ů exE:         x. P x; x. P x  Q  Q

  ů refl:        t = t
  ů subst:       s = t; P s  P t
  ů trans:       r = s; s = t  r = t
  ů sym:         s = t  t = s
  ů not_sym:     t  s  s  t
  ů ssubst:      t = s; P s  P t
  ů box_equals:  a = b; a = c; b = d  a: = d
  ů arg_cong:    x = y  f x = f y
  ů fun_cong:    f = g  f x = g x
  ů cong:        f = g; x = y  f x = g y
*}

text {*
  Se usarán las reglas notnotI y mt que demostramos a continuación.
  *}

lemma notnotI: "P  ňň P"
by auto

lemma mt: "F  G; ňG  ňF"
by auto

text {* ----------------------------------------------------------------
  Ejercicio 1. Demostrar
       x. P x  Q x  (x. P x)  (x. Q x)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_1a:
```

```
  "∀x. P x ∧ Q x ⟷ (∀x. P x) ∧ (∀x. Q x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_1b:
  assumes "∀x. P x ∧ Q x"
  shows   "(∀x. P x) ∧ (∀x. Q x)"
proof
  assume "∀x. P x"
  show "∀x. Q x"
  proof
    fix a
    have "P a" using ‹∀x. P x› ..
    have "P a ∧ Q a" using assms(1) ..
    thus "Q a" using ‹P a› ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_1c:
  assumes "∀x. P x ∧ Q x"
  shows   "(∀x. P x) ∧ (∀x. Q x)"
proof (rule impI)
  assume "∀x. P x"
  show "∀x. Q x"
  proof (rule allI)
    fix a
    have "P a" using ‹∀x. P x› by (rule allE)
    have "P a ∧ Q a" using assms(1) by (rule allE)
    thus "Q a" using ‹P a› by (rule mp)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 2. Demostrar
       ∀x. ¬¬(P x) ⟷ ¬¬(∀x. P x)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_2a: "∀x. ¬¬(P x) ⟷ ¬¬(∀x. P x)"
```

```
by auto

-- "La demostración estructurada es"
lemma ejercicio_2b:
  assumes "∀x. ň(P x)"
  shows   "ň(∃x. P x)"
proof
  assume "∃x. P x"
  obtain a where "ň(P a)" using assms(1) ..
  have "P a" using ‘∃x. P x‘ ..
  with ‘ň(P a)‘ show False ..
qed

-- "La demostración detallada es"
lemma ejercicio_2c:
  assumes "∀x. ň(P x)"
  shows   "ň(∃x. P x)"
proof (rule notI)
  assume "∃x. P x"
  obtain a where "ň(P a)" using assms(1) by (rule exE)
  have "P a" using ‘∃x. P x‘ by (rule allE)
  with ‘ň(P a)‘ show False by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 3. Demostrar
       ∀x. P x ⟶ ∃y. P y
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_3a: "∀x. P x ⟶ ∃y. P y"
by auto

-- "La demostración estructurada es"
lemma ejercicio_3b:
  assumes "∀x. P x"
  shows   "∃y. P y"
proof
  fix a
  show "P a" using assms ..
```

```
qed

-- "La demostración estructurada es"
lemma ejercicio_3c:
  assumes "∀x. P x"
  shows   "∀y. P y"
proof (rule allI)
  fix a
  show "P a" using assms by (rule allE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 4. Demostrar
       ∀x. P x → Q x → (∀x. ¬(Q x)) → (∀x. ¬ (P x))
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_4a:
  "∀x. P x → Q x → (∀x. ¬(Q x)) → (∀x. ¬ (P x))"
by auto

-- "La demostración estructurada es"
lemma ejercicio_4b:
  assumes "∀x. P x → Q x"
  shows   "(∀x. ¬(Q x)) → (∀x. ¬ (P x))"
proof
  assume "∀x. ¬(Q x)"
  show "∀x. ¬(P x)"
  proof
    fix a
    show "¬(P a)"
    proof
      assume "P a"
      have "P a → Q a" using assms ..
      hence "Q a" using ‘P a‘ ..
      have "¬(Q a)" using ‘∀x. ¬(Q x)‘ ..
      thus False using ‘Q a‘ ..
    qed
  qed
qed
```

```
-- "La demostración detallada es"
lemma ejercicio_4c:
  assumes "∀x. P x ∨ Q x"
  shows   "(∀x. ¬(Q x)) → (∃x. ¬ (P x))"
proof (rule impI)
  assume "∀x. ¬(Q x)"
  show "∃x. ¬(P x)"
  proof (rule allI)
    fix a
    show "¬(P a)"
    proof
      assume "P a"
      have "P a ∨ Q a" using assms by (rule allE)
      hence "Q a" using ‘P a‘ by (rule mp)
      have "¬(Q a)" using ‘∀x. ¬(Q x)‘ by (rule allE)
      thus False using ‘Q a‘ by (rule notE)
    qed
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 5. Demostrar
       ∀x. P x ∨ ¬(Q x) ⊢ ¬(∀x. P x ∧ Q x)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_5a:
  "∀x. P x ∨ ¬(Q x) ⊢ ¬(∀x. P x ∧ Q x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_5b:
  assumes "∀x. P x ∨ ¬(Q x)"
  shows   "¬(∀x. P x ∧ Q x)"
proof
  assume "∀x. P x ∧ Q x"
  then obtain a where "P a ∧ Q a" ..
  hence "P a" ..
  have "P a ∨ ¬(Q a)" using assms ..
```

```
  hence "¬(Q a)" using ‘P a‘ ..
  have "Q a" using ‘P a  Q a‘ ..
  with ‘¬(Q a)‘ show False ..
qed

-- "La demostración estructurada es"
lemma ejercicio_5c:
  assumes "∀x. P x  ¬(Q x)"
  shows   "¬(∃x. P x  Q x)"
proof (rule notI)
  assume "∃x. P x  Q x"
  then obtain a where "P a  Q a" by (rule exE)
  hence "P a" by (rule conjunct1)
  have "P a  ¬(Q a)" using assms by (rule allE)
  hence "¬(Q a)" using ‘P a‘ by (rule mp)
  have "Q a" using ‘P a  Q a‘ by (rule conjunct2)
  with ‘¬(Q a)‘ show False by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 6. Demostrar
       ∀x y. P x y  ∀u v. P u v
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_6a:
  "∀x y. P x y  ∀u v. P u v"
by auto

-- "La demostración estructurada es"
lemma ejercicio_6b:
  assumes "∀x y. P x y"
  shows   "∀u v. P u v"
proof
  fix a
  show "∀v. P a v"
  proof
    fix b
    have "∀y. P a y" using assms ..
    thus "P a b" ..
```

```
  qed
qed

-- "La demostración estructurada simplificada es"
lemma ejercicio_6b2:
  assumes "x y. P x y"
  shows   "u v. P u v"
proof (rule allI)+
  fix a b
  have "y. P a y" using assms ..
  thus "P a b" ..
qed

-- "La demostración detallada es"
lemma ejercicio_6c:
  assumes "x y. P x y"
  shows   "u v. P u v"
proof (rule allI)+
  fix a b
  have "y. P a y" using assms by (rule allE)
  thus "P a b" by (rule allE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 7. Demostrar
       x y. P x y  u v. P u v
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_7a:
  "x y. P x y  u v. P u v"
by auto

-- "La demostración estructurada es"
lemma ejercicio_7b:
  assumes "x y. P x y"
  shows   "u v. P u v"
proof -
  obtain a where "y. P a y" using assms ..
  then obtain b where "P a b" ..
```

```
  hence "∀v. P a v" ..
  thus "∀u v. P u v" ..
qed

text {* ---------------------------------------------------------------
  Ejercicio 8. Demostrar
       ∀x. ∀y. P x y → ∀y. ∀x. P x y
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_8a:
  "∀x. ∀y. P x y → ∀y. ∀x. P x y"
by auto

-- "La demostración estructurada es"
lemma ejercicio_8b:
  assumes "∀x. ∀y. P x y"
  shows   "∀y. ∀x. P x y"
proof
  fix b
  obtain a where "∀y. P a y" using assms ..
  hence "P a b" ..
  thus "∀x. P x b" ..
qed

-- "La demostración detallada es"
lemma ejercicio_8c:
  assumes "∀x. ∀y. P x y"
  shows   "∀y. ∀x. P x y"
proof (rule allI)
  fix b
  obtain a where "∀y. P a y" using assms by (rule exE)
  hence "P a b" by (rule allE)
  thus "∀x. P x b" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 9. Demostrar
       ∀x. P a → Q x → P a ∧ (∀x. Q x)
  --------------------------------------------------------------- *}
```

```
-- "La demostración automática es"
lemma ejercicio_9a:
  "∃x. P a ⟶ Q x ⟹ P a ⟶ (∃x. Q x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_9b:
  assumes "∃x. P a ⟶ Q x"
  shows   "P a ⟶ (∃x. Q x)"
proof
  assume "P a"
  obtain b where "P a ⟶ Q b" using assms ..
  hence "Q b" using ‹P a› ..
  thus "∃x. Q x" ..
qed

-- "La demostración detallada es"
lemma ejercicio_9c:
  assumes "∃x. P a ⟶ Q x"
  shows   "P a ⟶ (∃x. Q x)"
proof (rule impI)
  assume "P a"
  obtain b where "P a ⟶ Q b" using assms by (rule exE)
  hence "Q b" using ‹P a› by (rule mp)
  thus "∃x. Q x" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 10. Demostrar
       P a ⟶ (∃x. Q x) ⟹ ∃x. P a ⟶ Q x
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_10a:
  "P a ⟶ (∃x. Q x) ⟹ ∃x. P a ⟶ Q x"
by auto

-- "La demostración estructurada es"
lemma ejercicio_10b:
```

```
  fixes P Q :: "'b ⇒ bool"
  assumes "P a ∨ (∀x. Q x)"
  shows   "∀x. P a ∨ Q x"
proof -
  have "¬(P a) ∨ P a" ..
  thus "∀x. P a ∨ Q x"
  proof
    assume "¬(P a)"
    have "P a ∨ Q a"
    proof
      assume "P a"
      with ‹¬(P a)› show "Q a" ..
    qed
    thus "∀x. P a ∨ Q x" ..
  next
    assume "P a"
    with assms have "∀x. Q x" by (rule mp)
    then obtain b where "Q b" ..
    have "P a ∨ Q b"
    proof
      assume "P a"
      note ‹Q b›
      thus "Q b" .
    qed
    thus "∀x. P a ∨ Q x" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_10c:
  fixes P Q :: "'b ⇒ bool"
  assumes "P a ∨ (∀x. Q x)"
  shows   "∀x. P a ∨ Q x"
proof -
  have "¬(P a) ∨ P a" by (rule excluded_middle)
  thus "∀x. P a ∨ Q x"
  proof (rule disjE)
    assume "¬(P a)"
    have "P a ∨ Q a"
    proof (rule impI)
```

```
      assume "P a"
      with ‘¬(P a)‘ show "Q a" by (rule notE)
    qed
    thus "∃x. P a → Q x" by (rule exI)
  next
    assume "P a"
    with assms have "∃x. Q x" by (rule mp)
    then obtain b where "Q b" by (rule exE)
    have "P a → Q b"
    proof (rule impI)
      assume "P a"
      note ‘Q b‘
      thus "Q b" by this
    qed
    thus "∃x. P a → Q x" by (rule exI)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 11. Demostrar
       ∀(x. P x) → Q a ⊢ ∃x. P x → Q a
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_11a:
  "(∀x. P x) → Q a ⊢ ∃x. P x → Q a"
by auto

-- "La demostración estructurada es"
lemma ejercicio_11b:
  assumes "(∀x. P x) → Q a"
  shows   "∃x. P x → Q a"
proof
  fix b
  show "P b → Q a"
  proof
    assume "P b"
    hence "∀x. P x" ..
    with assms show "Q a" ..
  qed
```

```
qed

-- "La demostración detallada es"
lemma ejercicio_11c:
  assumes "(x. P x)  Q a"
  shows   "x. P x  Q a"
proof (rule allI)
  fix b
  show "P b  Q a"
  proof (rule impI)
    assume "P b"
    hence "x. P x" by (rule exI)
    with assms show "Q a" by (rule mp)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 12. Demostrar
       x. P x  Q a   x. P x  Q a
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_12a:
  "x. P x  Q a  x. P x  Q a"
by auto

-- "La demostración estructurada es"
lemma ejercicio_12b:
  assumes "x. P x  Q a"
  shows   "x. P x  Q a"
proof -
  have "P b  Q a" using assms ..
  thus "x. P x  Q a" ..
qed

-- "La demostración detallada es"
lemma ejercicio_12c:
  assumes "x. P x  Q a"
  shows   "x. P x  Q a"
proof -
```

```
  have "P b  Q a" using assms by (rule allE)
  thus "∃x. P x  Q a" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 13. Demostrar
       (∀x. P x)  (∀x. Q x)  ∀x. P x  Q x
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_13a:
  "(∀x. P x)  (∀x. Q x)  ∀x. P x  Q x"
by auto

-- "La demostración estructurada es"
lemma ejercicio_13b:
  assumes "(∀x. P x)  (∀x. Q x)"
  shows   "∀x. P x  Q x"
proof
  fix a
  note assms
  thus "P a  Q a"
  proof
    assume "∀x. P x"
    hence "P a" ..
    thus "P a  Q a" ..
  next
    assume "∀x. Q x"
    hence "Q a" ..
    thus "P a  Q a" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_13c:
  assumes "(∀x. P x)  (∀x. Q x)"
  shows   "∀x. P x  Q x"
proof (rule  allI)
  fix a
  note assms
```

```
  thus "P a ∨ Q a"
  proof (rule disjE)
    assume "∀x. P x"
    hence "P a" by (rule allE)
    thus "P a ∨ Q a" by (rule disjI1)
  next
    assume "∀x. Q x"
    hence "Q a" by (rule allE)
    thus "P a ∨ Q a" by (rule disjI2)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 14. Demostrar
       ∀x. P x ∨ Q x ⊢ (∀x. P x) ∨ (∀x. Q x)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_14a:
  "∀x. P x ∨ Q x ⟹ (∀x. P x) ∨ (∀x. Q x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_14b:
  assumes "∀x. P x ∨ Q x"
  shows   "(∀x. P x) ∨ (∀x. Q x)"
proof
  obtain a where "P a ∨ Q a" using assms ..
  hence "P a" ..
  thus "∀x. P x" ..
next
  obtain a where "P a ∨ Q a" using assms ..
  hence "Q a" ..
  thus "∀x. Q x" ..
qed

-- "La demostración detallada es"
lemma ejercicio_14c:
  assumes "∀x. P x ∨ Q x"
  shows   "(∀x. P x) ∨ (∀x. Q x)"
```

```
proof (rule conjI)
  obtain a where "P a  Q a" using assms by (rule exE)
  hence "P a" by (rule conjunct1)
  thus "x. P x" by (rule exI)
next
  obtain a where "P a  Q a" using assms by (rule exE)
  hence "Q a" by (rule conjunct2)
  thus "x. Q x" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 15. Demostrar
       x y. P y  Q x  (y. P y)  (x. Q x)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_15a:
  "x y. P y  Q x  (y. P y)  (x. Q x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_15b:
  assumes "x y. P y  Q x"
  shows   "(y. P y)  (x. Q x)"
proof
  assume "y. P y"
  then obtain b where "P b" ..
  show "x. Q x"
  proof
    fix a
    have "y. P y  Q a" using assms ..
    hence "P b  Q a" ..
    thus "Q a" using `P b` ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_15c:
  assumes "x y. P y  Q x"
  shows   "(y. P y)  (x. Q x)"
```

```
proof (rule impI)
  assume "∀y. P y"
  then obtain b where "P b" by (rule exE)
  show "∃x. Q x"
  proof (rule allI)
    fix a
    have "∀y. P y ⟶ Q a" using assms by (rule allE)
    hence "P b ⟶ Q a" by (rule allE)
    thus "Q a" using ‹P b› by (rule mp)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 16. Demostrar
       ¬∃x. ¬∀y. P x y ⟷ ∀x. ∀y. P x y
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_16a:
  "¬∃x. ¬∀y. P x y ⟷ ∀x. ∀y. P x y"
by auto

-- "La demostración estructurada es"
lemma ejercicio_16b:
  assumes "¬∃x. ¬∀y. P x y"
  shows   "∀x. ∀y. P x y"
proof (rule ccontr)
  assume "¬∃x. ∀y. P x y"
  have "∀x. ¬∀y. P x y"
  proof
    fix a
    show "¬∀y. P a y"
    proof
      assume "∀y. P a y"
      hence "∃x. ∀y. P x y" ..
      with ‹¬∃x. ∀y. P x y› show False ..
    qed
  qed
  with assms show False ..
qed
```

```
-- "La demostración detallada es"
lemma ejercicio_16c:
  assumes "ň(x. ň(P x))"
  shows   "x. P x"
proof (rule ccontr)
  assume "ň(x. P x)"
  have "x. ň(P x)"
  proof (rule allI)
    fix a
    show "ň(P a)"
    proof
      assume "P a"
      hence "x. P x" by (rule exI)
      with ‘ň(x. P x)‘ show False by (rule notE)
    qed
  qed
  with assms show False by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 17. Demostrar
       x. ň(P x)  ň(x. P x)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_17a:
  "x. ň(P x)  ň(x. P x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_17b:
  assumes "x. ň(P x)"
  shows   "ň(x. P x)"
proof
  assume "x. P x"
  then obtain a where "P a" ..
  have "ň(P a)" using assms ..
  thus False using ‘P a‘ ..
qed
```

```
-- "La demostración detallada es"
lemma ejercicio_17c:
  assumes "x. ň(P x)"
  shows   "ň(x. P x)"
proof (rule notI)
  assume "x. P x"
  then obtain a where "P a" by (rule exE)
  have "ň(P a)" using assms by (rule allE)
  thus False using ‘P a‘ by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 18. Demostrar
       x. P x  ň(x. ň(P x))
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_18a:
  "x. P x  ň(x. ň(P x))"
by auto

-- "La demostración estructurada es"
lemma ejercicio_18b:
  assumes "x. P x"
  shows   "ň(x. ň(P x))"
proof
  assume "x. ň(P x)"
  obtain a where "P a" using assms ..
  have "ň(P a)" using ‘x. ň(P x)‘ ..
  thus False using ‘P a‘ ..
qed

-- "La demostración detallada es"
lemma ejercicio_18c:
  assumes "x. P x"
  shows   "ň(x. ň(P x))"
proof (rule notI)
  assume "x. ň(P x)"
  obtain a where "P a" using assms by (rule exE)
```

```
  have "ň(P a)" using ‘x. ň(P x)‘ by (rule allE)
  thus False using ‘P a‘ by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 19. Demostrar
       P a  (x. Q x)  x. P a  Q x
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_19a:
  "P a  (x. Q x)  x. P a  Q x"
by auto

-- "La demostración estructurada es"
lemma ejercicio_19b:
  assumes "P a  (x. Q x)"
  shows   "x. P a  Q x"
proof
  fix b
  show "P a  Q b"
  proof
    assume "P a"
    with assms have "x. Q x" ..
    thus "Q b" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_19c:
  assumes "P a  (x. Q x)"
  shows   "x. P a  Q x"
proof (rule allI)
  fix b
  show "P a  Q b"
  proof (rule impI)
    assume "P a"
    with assms have "x. Q x" by (rule mp)
    thus "Q b" by (rule allE)
  qed
```

```
qed

text {* ---------------------------------------------------------------
  Ejercicio 20. Demostrar
       {∀x y z. R x y ∧ R y z → R x z,
        ∀x. ¬(R x x)}
        ∀x y. R x y → ¬(R y x)
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_20a:
  "∀x y z. R x y ∧ R y z → R x z ∧ ∀x. ¬(R x x) → ∀x y. R x y → ¬(R y x)"
by metis

-- "La demostración estructurada es"
lemma ejercicio_20b:
  assumes "∀x y z. R x y ∧ R y z → R x z"
          "∀x. ¬(R x x)"
  shows   "∀x y. R x y → ¬(R y x)"
proof (rule allI)+
  fix a b
  show "R a b → ¬(R b a)"
  proof
    assume "R a b"
    show "¬(R b a)"
    proof
      assume "R b a"
      show False
      proof -
        have "R a b ∧ R b a" using ‹R a b› ‹R b a› ..
        have "∀y z. R a y ∧ R y z → R a z" using assms(1) ..
        hence "∀z. R a b ∧ R b z → R a z" ..
        hence "R a b ∧ R b a → R a a" ..
        hence "R a a" using ‹R a b ∧ R b a› ..
        have "¬(R a a)" using assms(2) ..
        thus False using ‹R a a› ..
      qed
    qed
  qed
qed
```

```
-- "La demostración detallada es"
lemma ejercicio_20c:
  assumes "∀x y z. R x y ∧ R y z → R x z"
          "∀x. ¬(R x x)"
  shows   "∀x y. R x y → ¬(R y x)"
proof (rule allI)+
  fix a b
  show "R a b → ¬(R b a)"
  proof (rule impI)
    assume "R a b"
    show "¬(R b a)"
    proof (rule notI)
      assume "R b a"
      show False
      proof -
        have "R a b ∧ R b a" using ‘R a b‘ ‘R b a‘ by (rule conjI)
        have "∀y z. R a y ∧ R y z → R a z" using assms(1) by (rule allE)
        hence "∀z. R a b ∧ R b z → R a z" by (rule allE)
        hence "R a b ∧ R b a → R a a" by (rule allE)
        hence "R a a" using ‘R a b ∧ R b a‘ by (rule mp)
        have "¬(R a a)" using assms(2) by (rule allE)
        thus False using ‘R a a‘ by (rule notE)
      qed
    qed
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 21. Demostrar
     {∀x. P x ∨ Q x, ∀x. ¬(Q x), ∃x. R x ∧ ¬(P x)} ⊢ ∃x. ¬(R x)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_21a:
  "∀x. P x ∨ Q x; ∀x. ¬(Q x); ∃x. R x ∧ ¬(P x) ⟹ ∃x. ¬(R x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_21b:
```

```
  assumes "∀x. P x ∨ Q x"
          "∀x. ¬(Q x)"
          "∀x. R x → ¬(P x)"
  shows   "∀x. ¬(R x)"
proof -
  obtain a where "¬(Q a)" using assms(2) ..
  have "P a ∨ Q a" using assms(1) ..
  hence "P a"
  proof
    assume "P a"
    thus "P a" .
  next
    assume "Q a"
    with ‹¬(Q a)› show "P a" ..
  qed
  hence "¬¬(P a)" by (rule notnotI)
  have "R a → ¬(P a)" using assms(3) ..
  hence "¬(R a)" using ‹¬¬(P a)› by (rule mt)
  thus "∀x. ¬(R x)" ..
qed

-- "La demostración detallada es"
lemma ejercicio_21c:
  assumes "∀x. P x ∨ Q x"
          "∀x. ¬(Q x)"
          "∀x. R x → ¬(P x)"
  shows   "∀x. ¬(R x)"
proof -
  obtain a where "¬(Q a)" using assms(2) by (rule exE)
  have "P a ∨ Q a" using assms(1) by (rule allE)
  hence "P a"
  proof (rule disjE)
    assume "P a"
    thus "P a" by this
  next
    assume "Q a"
    with ‹¬(Q a)› show "P a" by (rule notE)
  qed
  hence "¬¬(P a)" by (rule notnotI)
  have "R a → ¬(P a)" using assms(3) by (rule allE)
```

```
  hence "ň(R a)" using ‘ňň(P a)‘ by (rule mt)
  thus "x. ň(R x)" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 22. Demostrar
     {x. P x  Q x  R x, ň(x. P x  R x)}  x. P x  Q x
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_22a:
  "x. P x  Q x  R x; ň(x. P x  R x)  x. P x  Q x"
by auto

-- "La demostración estructurada es"
lemma ejercicio_22b:
  assumes "x. P x  Q x  R x"
          "ň(x. P x  R x)"
  shows   "x. P x  Q x"
proof
  fix a
  show "P a  Q a"
  proof
    assume "P a"
    have "P a  Q a  R a" using assms(1) ..
    hence "Q a  R a" using ‘P a‘ ..
    thus "Q a"
    proof
      assume "Q a"
      thus "Q a" .
    next
      assume "R a"
      with ‘P a‘ have "P a  R a" ..
      hence "x. P x  R x" ..
      with assms(2) show "Q a" ..
    qed
  qed
qed

-- "La demostración detallada es"
```

```
lemma ejercicio_22c:
  assumes "∀x. P x ⟶ Q x ∨ R x"
          "¬(∀x. P x ⟶ R x)"
  shows   "∃x. P x ∧ Q x"
proof (rule allI)
  fix a
  show "P a ⟶ Q a"
  proof (rule impI)
    assume "P a"
    have "P a ⟶ Q a ∨ R a" using assms(1) by (rule allE)
    hence "Q a ∨ R a" using ‹P a› by (rule mp)
    thus "Q a"
    proof (rule disjE)
      assume "Q a"
      thus "Q a" by this
    next
      assume "R a"
      with ‹P a› have "P a ∧ R a" by (rule conjI)
      hence "∃x. P x ∧ R x" by (rule exI)
      with assms(2) show "Q a" by (rule notE)
    qed
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 23. Demostrar
     ∀x y. R x y ⟶ R y x ⟶ ∀x y. R x y
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_23a:
  "∀x y. R x y ⟶ R y x ⟶ ∀x y. R x y"
by auto

-- "La demostración estructurada es"
lemma ejercicio_23b:
  assumes "∀x y. R x y ⟶ R y x"
  shows   "∀x y. R x y"
proof -
  obtain a where "∀y. R a y ⟶ R y a" using assms ..
```

```
  then obtain b where "R a b  R b a" ..
  thus "x y. R x y"
  proof
    assume "R a b"
    hence "y. R a y" ..
    thus " x y. R x y" ..
  next
    assume "R b a"
    hence "y. R b y" ..
    thus " x y. R x y" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_23c:
  assumes "x y. R x y  R y x"
  shows   "x y. R x y"
proof -
  obtain a where "y. R a y  R y a" using assms by (rule exE)
  then obtain b where "R a b  R b a" by (rule exE)
  thus "x y. R x y"
  proof (rule disjE)
    assume "R a b"
    hence "y. R a y" by (rule exI)
    thus " x y. R x y" by (rule exI)
  next
    assume "R b a"
    hence "y. R b y" by (rule exI)
    thus " x y. R x y" by (rule exI)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 24. Demostrar
       (x. y. P x y)  (y. x. P x y)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_24a: "(x. y. P x y)  (y. x. P x y)"
by auto
```

```
-- "La demostración estructurada es"
lemma ejercicio_24b: "(∀x. ∀y. P x y) → (∀y. ∀x. P x y)"
proof
  assume "∀x. ∀y. P x y"
  then obtain a where "∀y. P a y" ..
  show "∀y. ∀x. P x y"
  proof
    fix b
    have "P a b" using ‹∀y. P a y› ..
    thus "∀x. P x b" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_24c: "(∀x. ∀y. P x y) → (∀y. ∀x. P x y)"
proof (rule impI)
  assume "∀x. ∀y. P x y"
  then obtain a where "∀y. P a y" by (rule exE)
  show "∀y. ∀x. P x y"
  proof (rule allI)
    fix b
    have "P a b" using ‹∀y. P a y› by (rule allE)
    thus "∀x. P x b" by (rule exI)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 25. Demostrar
       (∀x. P x → Q) → ((∃x. P x) → Q)
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_25a: "(∀x. P x → Q) → ((∃x. P x) → Q)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_25b: "(∀x. P x → Q) → ((∃x. P x) → Q)"
proof
  assume "∀x. P x → Q"
```

```
  show "(x. P x)  Q"
  proof
    assume "x. P x"
    then obtain a where "P a" ..
    have "P a  Q" using `x. P x  Q` ..
    thus "Q" using `P a` ..
  qed
next
  assume "(x. P x)  Q"
  show "x. P x  Q"
  proof
    fix b
    show "P b  Q"
    proof
      assume "P b"
      hence "x. P x" ..
      with `(x. P x)  Q` show "Q" ..
    qed
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_25c: "(x. P x  Q)  ((x. P x)  Q)"
proof (rule iffI)
  assume "x. P x  Q"
  show "(x. P x)  Q"
  proof (rule impI)
    assume "x. P x"
    then obtain a where "P a" by (rule exE)
    have "P a  Q" using `x. P x  Q` by (rule allE)
    thus "Q" using `P a` by (rule mp)
  qed
next
  assume "(x. P x)  Q"
  show "x. P x  Q"
  proof (rule allI)
    fix b
    show "P b  Q"
    proof (rule impI)
      assume "P b"
```

```
      hence "∃x. P x" by (rule exI)
      with ‘(∃x. P x) ⟶ Q‘ show "Q" by (rule mp)
    qed
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 26. Demostrar
       ((∃x. P x) ∧ (∃x. Q x)) ⟷ (∃x. P x ∧ Q x)
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_26a: "((∃x. P x) ∧ (∃x. Q x)) ⟷ (∃x. P x ∧ Q x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_26b: "((∃x. P x) ∧ (∃x. Q x)) ⟷ (∃x. P x ∧ Q x)"
proof
  assume "(∃x. P x) ∧ (∃x. Q x)"
  show "∃x. P x ∧ Q x"
  proof
    fix a
    have "∃x. P x" using ‘(∃x. P x) ∧ (∃x. Q x)‘ ..
    have "∃x. Q x" using ‘(∃x. P x) ∧ (∃x. Q x)‘ ..
    hence "Q a" ..
    have "P a" using ‘∃x. P x‘ ..
    thus "P a ∧ Q a" using ‘Q a‘ ..
  qed
next
  assume "∃x. P x ∧ Q x"
  have "∃x. P x"
  proof
    fix a
    have "P a ∧ Q a" using ‘∃x. P x ∧ Q x‘ ..
    thus "P a" ..
  qed
  moreover have "∃x. Q x"
  proof
    fix a
    have "P a ∧ Q a" using ‘∃x. P x ∧ Q x‘ ..
```

```
    thus "Q a" ..
  qed
  ultimately show "(x. P x)  (x. Q x)" ..
qed

-- "La demostración detallada es"
lemma ejercicio_26c: "((x. P x)  (x. Q x)) = (x. P x  Q x)"
proof (rule iffI)
  assume "(x. P x)  (x. Q x)"
  show "x. P x  Q x"
  proof (rule allI)
    fix a
    have "x. P x" using `(x. P x)  (x. Q x)` by (rule conjunct1)
    have "x. Q x" using `(x. P x)  (x. Q x)` by (rule conjunct2)
    hence "Q a" by (rule allE)
    have "P a" using `x. P x` by (rule allE)
    thus "P a  Q a" using `Q a` by (rule conjI)
  qed
next
  assume "x. P x  Q x"
  have "x. P x"
  proof (rule allI)
    fix a
    have "P a  Q a" using `x. P x  Q x` by (rule allE)
    thus "P a" by (rule conjunct1)
  qed
  moreover have "x. Q x"
  proof (rule allI)
    fix a
    have "P a  Q a" using `x. P x  Q x` by (rule allE)
    thus "Q a" by (rule conjunct2)
  qed
  ultimately show "(x. P x)  (x. Q x)" by (rule conjI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 27. Demostrar o refutar
       ((x. P x)  (x. Q x))  (x. P x  Q x)
  ---------------------------------------------------------------- *}
```

```
lemma ejercicio_27: "((∀x. P x) ∨ (∀x. Q x)) → (∀x. P x ∨ Q x)"
oops

(*
Auto Quickcheck found a counterexample:
P = {a\<^isub>1}
Q = {a\<^isub>2}
*)

text {* ---------------------------------------------------------------
  Ejercicio 28. Demostrar o refutar
       ((∀x. P x) ∨ (∀x. Q x)) → (∀x. P x ∨ Q x)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_28a:
  "((∀x. P x) ∨ (∀x. Q x)) → (∀x. P x ∨ Q x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_28b:
  "((∀x. P x) ∨ (∀x. Q x)) → (∀x. P x ∨ Q x)"
proof
  assume "(∀x. P x) ∨ (∀x. Q x)"
  thus "∀x. P x ∨ Q x"
  proof
    assume "∀x. P x"
    then obtain a where "P a" ..
    hence "P a ∨ Q a" ..
    thus "∀x. P x ∨ Q x" ..
  next
    assume "∀x. Q x"
    then obtain a where "Q a" ..
    hence "P a ∨ Q a" ..
    thus "∀x. P x ∨ Q x" ..
  qed
next
  assume "∀x. P x ∨ Q x"
  then obtain a where "P a ∨ Q a" ..
  thus "(∀x. P x) ∨ (∀x. Q x)"
```

```
  proof
    assume "P a"
    hence "∃x. P x" ..
    thus "(∃x. P x) ∨ (∃x. Q x)" ..
  next
    assume "Q a"
    hence "∃x. Q x" ..
    thus "(∃x. P x) ∨ (∃x. Q x)" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_28c:
  "((∃x. P x) ∨ (∃x. Q x)) ↔ (∃x. P x ∨ Q x)"
proof (rule iffI)
  assume "(∃x. P x) ∨ (∃x. Q x)"
  thus "∃x. P x ∨ Q x"
  proof (rule disjE)
    assume "∃x. P x"
    then obtain a where "P a" by (rule exE)
    hence "P a ∨ Q a" by (rule disjI1)
    thus "∃x. P x ∨ Q x" by (rule exI)
  next
    assume "∃x. Q x"
    then obtain a where "Q a" by (rule exE)
    hence "P a ∨ Q a" by (rule disjI2)
    thus "∃x. P x ∨ Q x" by (rule exI)
  qed
next
  assume "∃x. P x ∨ Q x"
  then obtain a where "P a ∨ Q a" by (rule exE)
  thus "(∃x. P x) ∨ (∃x. Q x)"
  proof (rule disjE)
    assume "P a"
    hence "∃x. P x" by (rule exI)
    thus "(∃x. P x) ∨ (∃x. Q x)" by (rule disjI1)
  next
    assume "Q a"
    hence "∃x. Q x" by (rule exI)
    thus "(∃x. P x) ∨ (∃x. Q x)" by (rule disjI2)
```

```
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 29. Demostrar o refutar
       (x. y. P x y)  (y. x. P x y)
  ---------------------------------------------------------------- *}

lemma ejercicio_29:
  "(x. y. P x y)  (y. x. P x y)"
quickcheck
(*
Quickcheck found a counterexample:

P = (x. undefined)(a\<^isub> := {b}, b := {a})
*)
oops

text {* ---------------------------------------------------------------
  Ejercicio 30. Demostrar o refutar
       (ň(x. P x))  (x. ňP x)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_30a:
  "(ň(x. P x))  (x. ňP x)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_30b:
  "(ň(x. P x))  (x. ňP x)"
proof
  assume "ň(x. P x)"
  show "x. ňP x"
  proof (rule ccontr)
    assume "ň(x. ňP x)"
    have "x. P x"
    proof
      fix a
      show "P a"
```

```
      proof (rule ccontr)
        assume "ňP a"
        hence "x. ňP x" ..
        with ‘ň(x. ňP x)‘ show False ..
      qed
    qed
    with ‘ň(x. P x)‘ show False ..
  qed
next
  assume "x. ňP x"
  then obtain a where "ňP a" ..
  show "ň(x. P x)"
  proof
    assume "x. P x"
    hence "P a" ..
    with ‘ňP a‘ show False ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_30c:
  "(ň(x. P x))  (x. ňP x)"
proof (rule iffI)
  assume "ň(x. P x)"
  show "x. ňP x"
  proof (rule ccontr)
    assume "ň(x. ňP x)"
    have "x. P x"
    proof (rule allI)
      fix a
      show "P a"
      proof (rule ccontr)
        assume "ňP a"
        hence "x. ňP x" by (rule exI)
        with ‘ň(x. ňP x)‘ show False by (rule notE)
      qed
    qed
    with ‘ň(x. P x)‘ show False by (rule notE)
  qed
next
```

```
  assume "x. ňP x"
  then obtain a where "ňP a" by (rule exE)
  show "ň(x. P x)"
  proof (rule notI)
    assume "x. P x"
    hence "P a" by (rule allE)
    show False using ‘ňP a‘ ‘P a‘ by (rule notE)
  qed
qed

section {* Ejercicios sobre igualdad *}

text {* ----------------------------------------------------------------
  Ejercicio 31. Demostrar o refutar
       P a  x. x = a  P x
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_31a:
  "P a  x. x = a  P x"
by auto

-- "La demostración estructurada es"
lemma ejercicio_31b:
  assumes "P a"
  shows   "x. x = a  P x"
proof
  fix b
  show "b = a  P b"
  proof
    assume "b = a"
    thus "P b" using assms by (rule ssubst)
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_31c:
  assumes "P a"
  shows   "x. x = a  P x"
proof (rule allI)
```

```
  fix b
  show "b = a  P b"
  proof (rule impI)
    assume "b = a"
    thus "P b" using assms by (rule ssubst)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 32. Demostrar o refutar
       x y. R x y  R y x; ¬(x. R x x)  x y. x  y
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_32a:
  fixes R :: "'c  'c  bool"
  assumes "x y. R x y  R y x"
          "¬(x. R x x)"
  shows   "(x::'c) y. x  y"
using assms
by metis

-- "La demostración estructurada es"
lemma ejercicio_32b:
  fixes R :: "'c  'c  bool"
  assumes "x y. R x y  R y x"
          "¬(x. R x x)"
  shows   "(x::'c) y. x  y"
proof -
  obtain a where "y. R a y  R y a" using assms(1) ..
  then obtain b where "R a b  R b a" ..
  hence "a  b"
  proof
    assume "R a b"
    show "a  b"
    proof
      assume "a = b"
      hence "R b b" using `R a b` by (rule subst)
      hence "x. R x x" ..
      with assms(2) show False ..
```

```
    qed
  next
    assume "R b a"
    show "a  b"
    proof
      assume "a = b"
      hence "R a a" using ‘R b a‘ by (rule ssubst)
      hence "x. R x x" ..
      with assms(2) show False ..
    qed
  qed
  hence "y. a  y" ..
  thus "(x::'c) y. x  y" ..
qed

-- "La demostración detallada es"
lemma ejercicio_32c:
  fixes R :: "'c  'c  bool"
  assumes "x y. R x y  R y x"
          "ň(x. R x x)"
  shows   "(x::'c) y. x  y"
proof -
  obtain a where "y. R a y  R y a" using assms(1) by (rule exE)
  then obtain b where "R a b  R b a" by (rule exE)
  hence "a  b"
  proof (rule disjE)
    assume "R a b"
    show "a  b"
    proof (rule notI)
      assume "a = b"
      hence "R b b" using ‘R a b‘ by (rule subst)
      hence "x. R x x" by (rule exI)
      with assms(2) show False by (rule notE)
    qed
  next
    assume "R b a"
    show "a  b"
    proof (rule notI)
      assume "a = b"
      hence "R a a" using ‘R b a‘ by (rule ssubst)
```

```
      hence "x. R x x" by (rule exI)
      with assms(2) show False by (rule notE)
    qed
  qed
  hence "y. a  y" by (rule exI)
  thus "(x::'c) y. x  y" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 33. Demostrar o refutar
     {x. P a x x,
      x y z. P x y z  P (f x) y (f z)}
      P (f a) a (f a)
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_33a:
  "x. P a x x; x y z. P x y z  P (f x) y (f z)  P (f a) a (f a)"
by auto

-- "La demostración estructura es"
lemma ejercicio_33b:
  assumes "x. P a x x"
          "x y z. P x y z  P (f x) y (f z)"
  shows   "P (f a) a (f a)"
proof -
  have "P a a a" using assms(1) ..
  have "y z. P a y z  P (f a) y (f z)" using assms(2) ..
  hence "z. P a a z  P (f a) a (f z)" ..
  hence "P a a a  P (f a) a (f a)" ..
  thus "P (f a) a (f a)" using `P a a a` ..
qed

-- "La demostración detallada es"
lemma ejercicio_33c:
  assumes "x. P a x x"
          "x y z. P x y z  P (f x) y (f z)"
  shows   "P (f a) a (f a)"
proof -
  have "P a a a" using assms(1) by (rule allE)
```

```
  have "y z. P a y z  P (f a) y (f z)" using assms(2) by (rule allE)
  hence "z. P a a z  P (f a) a (f z)" by (rule allE)
  hence "P a a a  P (f a) a (f a)" by (rule allE)
  thus "P (f a) a (f a)" using `P a a a` by (rule mp)
qed

text {* ---------------------------------------------------------------
  Ejercicio 34. Demostrar o refutar
     {x. P a x x,
      x y z. P x y z  P (f x) y (f z)
      z. P (f a) z (f (f a))
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_34a:
  "x. P a x x; x y z. P x y z  P (f x) y (f z)
    z. P (f a) z (f (f a))"
by metis

-- "La demostración estructura es"
lemma ejercicio_34b:
  assumes "x. P a x x"
          "x y z. P x y z  P (f x) y (f z)"
  shows   "z. P (f a) z (f (f a))"
proof -
  have "P a (f a) (f a)" using assms(1) ..
  have "y z. P a y z  P (f a) y (f z)" using assms(2) ..
  hence "z. P a (f a) z  P (f a) (f a) (f z)" ..
  hence "P a (f a) (f a)  P (f a) (f a) (f (f a))" ..
  hence "P (f a) (f a) (f (f a))" using `P a (f a) (f a)` ..
  thus "z. P (f a) z (f (f a))" ..
qed

-- "La demostración detallada es"
lemma ejercicio_34c:
  assumes "x. P a x x"
          "x y z. P x y z  P (f x) y (f z)"
  shows   "z. P (f a) z (f (f a))"
proof -
  have "P a (f a) (f a)" using assms(1) by (rule allE)
```

```
  have "y z. P a y z  P (f a) y (f z)" using assms(2) by (rule allE)
  hence "z. P a (f a) z  P (f a) (f a) (f z)" by (rule allE)
  hence "P a (f a) (f a)  P (f a) (f a) (f (f a))" by (rule allE)
  hence "P (f a) (f a) (f (f a))" using `P a (f a) (f a)` by (rule mp)
  thus "z. P (f a) z (f (f a))" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 35. Demostrar o refutar
     {y. Q a y,
      x y. Q x y  Q (s x) (s y)}
      z. Qa z  Q z (s (s a))
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_35a:
  "y. Q a y; x y. Q x y  Q (s x) (s y)  z. Q a z  Q z (s (s a))"
by auto

-- "La demostración estructura es"
lemma ejercicio_35b:
  assumes "y. Q a y"
          "x y. Q x y  Q (s x) (s y)"
  shows   "z. Q a z  Q z (s (s a))"
proof -
  have "Q a (s a)" using assms(1) ..
  have "y. Q a y  Q (s a) (s y)" using assms(2) ..
  hence "Q a (s a)  Q (s a) (s (s a))" ..
  hence "Q (s a) (s (s a))" using `Q a (s a)` ..
  with `Q a (s a)` have "Q a (s a)  Q (s a) (s (s a))" ..
  thus "z. Q a z  Q z (s (s a))" ..
qed

-- "La demostración detallada es"
lemma ejercicio_35c:
  assumes "y. Q a y"
          "x y. Q x y  Q (s x) (s y)"
  shows   "z. Q a z  Q z (s (s a))"
proof -
  have "Q a (s a)" using assms(1) by (rule allE)
```

```
  have "∃y. Q a y ∧ Q (s a) (s y)" using assms(2) by (rule allE)
  hence "Q a (s a) → Q (s a) (s (s a))" by (rule allE)
  hence "Q (s a) (s (s a))" using ‘Q a (s a)‘ by (rule mp)
  with ‘Q a (s a)‘ have "Q a (s a) ∧ Q (s a) (s (s a))" by (rule conjI)
  thus "∃z. Q a z ∧ Q z (s (s a))" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 36. Demostrar o refutar
     {x = f x, odd (f x)} ⊢ odd x
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_36a:
  "x = f x; odd (f x) ⟹ odd x"
by auto

-- "La demostración semiautomática es"
lemma ejercicio_36b:
  "x = f x; odd (f x) ⟹ odd x"
by (rule ssubst)

-- "La demostración estructurada es"
lemma ejercicio_36c:
  assumes "x = f x"
          "odd (f x)"
  shows   "odd x"
proof -
  show "odd x" using assms by (rule ssubst)
qed

text {* ---------------------------------------------------------------
  Ejercicio 37. Demostrar o refutar
     {x = f x, triple (f x) (f x) x} ⊢ triple x x x
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_37a:
  "x = f x; triple (f x) (f x) x ⟹ triple x x x"
by auto
```

```
-- "La demostración semiautomática es"
lemma ejercicio_37b:
  "x = f x; triple (f x) (f x) x  triple x x x"
by (rule ssubst)

-- "La demostración estructurada es"
lemma ejercicio_37c:
  assumes "x = f x"
          "triple (f x) (f x) x"
  shows   "triple x x x"
proof -
  show "triple x x x" using assms by (rule ssubst)
qed

end
```

4.3. Ejercicios: Argumentación lógica de primer orden

```
chapter {* T4R2: Argumentación en lógica de primer orden *}

theory T4R2
imports Main
begin

text {*
  ----------------------------------------------------------------------
  El objetivo de esta relación es formalizar y decidir la corrección
  de los argumentos. En el caso de que sea correcto, demostrarlo usando
  sólo las reglas básicas de deducción natural de la lógica de primer
  orden (sin usar el método auto). En el caso de que sea incorrecto,
  calcular un contraejemplo con QuickCheck.

  Las reglas básicas de la deducción natural son las siguientes:
  ů conjI:      P; Q  P  Q
  ů conjunct1:  P  Q  P
  ů conjunct2:  P  Q  Q
  ů notnotD:    ňň P  P
  ů notnotI:    P  ňň P
  ů mp:         P  Q; P  Q
```

```
  ů mt:          F  G; ňG  ňF
  ů impI:        (P  Q)  P  Q
  ů disjI1:      P  P  Q
  ů disjI2:      Q  P  Q
  ů disjE:       P  Q; P  R; Q  R  R
  ů FalseE:      False  P
  ů notE:        ňP; P  R
  ů notI:        (P  False)  ňP
  ů iffI:        P  Q; Q  P  P = Q
  ů iffD1:       Q = P; Q  P
  ů iffD2:       P = Q; Q  P
  ů ccontr:      (ňP  False)  P
  ů excluded_middle: ňP  P

  ů allI:        x. P x; P x  R  R
  ů allE:        (x. P x)  x. P x
  ů exI:         P x  x. P x
  ů exE:         x. P x; x. P x  Q  Q

  ----------------------------------------------------------------------
*}

text {*
  Se usarán las reglas notnotI y mt que demostramos a continuación.
  *}

lemma notnotI: "P  ňň P"
by auto

lemma mt: "F  G; ňG  ňF"
by auto

lemma no_ex: "ň(x. P(x))  x. ňP(x)"
by auto

lemma no_para_todo: "ň(x. P(x))  x. ňP(x)"
by auto

text {* ----------------------------------------------------------------
  Ejercicio 1. Formalizar, y decidir la corrección, del siguiente
```

```
  argumento
     Sócrates es un hombre.
     Los hombres son mortales.
     Luego, Sócrates es mortal.
  Usar s    para Sócrates
       H(x) para x es un hombre
       M(x) para x es mortal
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_1a:
  "H(s); x. H(x)  M(x)  M(s)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_1b:
  assumes "H(s)"
          "x. H(x)  M(x)"
  shows   "M(s)"
proof -
  have "H(s)  M(s)" using assms(2) ..
  thus "M(s)" using assms(1) ..
qed

-- "La demostración detallada es"
lemma ejercicio_1c:
  assumes "H(s)"
          "x. H(x)  M(x)"
  shows   "M(s)"
proof -
  have "H(s)  M(s)" using assms(2) by (rule allE)
  thus "M(s)" using assms(1) by (rule mp)
qed

text {* ----------------------------------------------------------------
  Ejercicio 2. Formalizar, y decidir la corrección, del siguiente
  argumento
     Hay estudiantes inteligentes y hay estudiantes trabajadores. Por
     tanto, hay estudiantes inteligentes y trabajadores.
  Usar I(x) para x es inteligente
```

```
       T(x) para x es trabajador
  ---------------------------------------------------------------- *}

-- "La refutación automática es"
lemma ejercicio_2a:
  "(x. I(x))  (x. T(x))  x. I(x)  T(x)"
quickcheck
oops

text {*
El argumento es incorrecto como muestra el siguiente contraejemplo:
   I = {a\<^isub>1}
   T = {a\<^isub>2}
*}

text {* ----------------------------------------------------------------
  Ejercicio 3. Formalizar, y decidir la corrección, del siguiente
  argumento
     Todos los participantes son vencedores. Hay como máximo un
     vencedor. Hay como máximo un participante. Por lo tanto, hay
     exactamente un participante.
  Usar P(x) para x es un participante
       V(x) para x es un vencedor
  ---------------------------------------------------------------- *}

-- "La refutación automática es"
lemma ejercicio_3a:
  "x. P(x)  V(x);
    x y. V(x)  V(y)  x=y;
    x y. P(x)  P(y)  x=y
    x. P(x)  (y. P(y)  x=y)"
quickcheck
oops

text {*
El argumento es incorrecto como muestra el siguiente contraejemplo:
   V = {}
   P = {}
*}
```

```
text {* ---------------------------------------------------------------
  Ejercicio 4. Formalizar, y decidir la corrección, del siguiente
  argumento
     Todo aquel que entre en el país y no sea un VIP será cacheado por
     un aduanero. Hay un contrabandista que entra en el país y que solo
     podrá ser cacheado por contrabandistas. Ningún contrabandista es un
     VIP. Por tanto, algún aduanero es contrabandista.
  Usar A(x)    para x es aduanero
       Ca(x,y) para x cachea a y
       Co(x)   para x es contrabandista
       E(x)    para x entra en el pais
       V(x)    para x es un VIP
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_4a:
  "∀x. E(x) ∧ ¬V(x) ⟶ (∃y. A(y) ∧ Ca(y,x));
    ∃x. Co(x) ∧ E(x) ∧ (∀y. Ca(y,x) ⟶ Co(y));
    ¬(∀x. Co(x) ⟶ V(x))
    ∃x. A(x) ∧ Co(x)"
by metis

-- "La demostración estructurada es"
lemma ejercicio_4b:
  assumes "∀x. E(x) ∧ ¬V(x) ⟶ (∃y. A(y) ∧ Ca(y,x))"
          "∃x. Co(x) ∧ E(x) ∧ (∀y. Ca(y,x) ⟶ Co(y))"
          "¬(∀x. Co(x) ⟶ V(x))"
  shows   "∃x. A(x) ∧ Co(x)"
proof -
  obtain a where a: "Co(a) ∧ E(a) ∧ (∀y. Ca(y,a) ⟶ Co(y))"
    using assms(2) ..
  have "∃y. A(y) ∧ Ca(y,a)"
  proof -
    have "E(a) ∧ ¬V(a) ⟶ (∃y. A(y) ∧ Ca(y,a))" using assms(1) ..
    moreover
    have "E(a) ∧ ¬V(a)"
    proof
      have "E(a) ∧ (∀y. Ca(y,a) ⟶ Co(y))" using a ..
      thus "E(a)" ..
    next
```

```
      have "x. ň(Co(x)  V(x))" using assms(3) by (rule no_ex)
      hence "ň(Co(a)  V(a))" ..
      have "Co(a)" using a ..
      show "ňV(a)"
      proof
        assume "V(a)"
        with ‘Co(a)‘ have "Co(a)  V(a)" ..
        with ‘ň(Co(a)  V(a))‘ show False ..
      qed
    qed
    ultimately show "y. A(y)  Ca(y,a)" ..
  qed
  then obtain b where "A(b)  Ca(b,a)" ..
  hence "A(b)" ..
  moreover
  have "Co(b)"
    proof -
      have "E(a)  (y. Ca(y,a)  Co(y))" using a ..
      hence "y. Ca(y,a)  Co(y)" ..
      hence "Ca(b,a)  Co(b)" ..
      have "Ca(b,a)" using ‘A(b)  Ca(b,a)‘ ..
      with ‘Ca(b,a)  Co(b)‘ show "Co(b)" ..
    qed
  ultimately have "A(b)  Co(b)" ..
  thus "x. A(x)  Co(x)" ..
qed

-- "La demostración detallada es"
lemma ejercicio_4c:
  assumes "x. E(x)  ňV(x)  (y. A(y)  Ca(y,x))"
          "x. Co(x)  E(x)  (y. Ca(y,x)  Co(y))"
          "ň(x. Co(x)  V(x))"
  shows   "x. A(x)  Co(x)"
proof -
  obtain a where a: "Co(a)  E(a)  (y. Ca(y,a)  Co(y))"
    using assms(2) by (rule exE)
  have "y. A(y)  Ca(y,a)"
  proof -
    have "E(a)  ňV(a)  (y. A(y)  Ca(y,a))" using assms(1) by (rule allE)
    moreover
```

```
    have "E(a)  ňV(a)"
    proof
      have "E(a)  (y. Ca(y,a)  Co(y))" using a by (rule conjunct2)
      thus "E(a)" by (rule conjunct1)
    next
      have "x. ň(Co(x)  V(x))" using assms(3) by (rule no_ex)
      hence "ň(Co(a)  V(a))" by (rule allE)
      have "Co(a)" using a by (rule conjunct1)
      show "ňV(a)"
      proof (rule notI)
        assume "V(a)"
        with ‘Co(a)‘ have "Co(a)  V(a)" by (rule conjI)
        with ‘ň(Co(a)  V(a))‘ show False by (rule notE)
      qed
    qed
    ultimately show "y. A(y)  Ca(y,a)" by (rule mp)
  qed
  then obtain b where "A(b)  Ca(b,a)" by (rule exE)
  hence "A(b)" by (rule conjunct1)
  moreover
  have "Co(b)"
    proof -
      have "E(a)  (y. Ca(y,a)  Co(y))" using a by (rule conjunct2)
      hence "y. Ca(y,a)  Co(y)" by (rule conjunct2)
      hence "Ca(b,a)  Co(b)" by (rule allE)
      have "Ca(b,a)" using ‘A(b)  Ca(b,a)‘ by (rule conjunct2)
      with ‘Ca(b,a)  Co(b)‘ show "Co(b)" by (rule mp)
    qed
  ultimately have "A(b)  Co(b)" by (rule conjI)
  thus "x. A(x)  Co(x)" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 5. Formalizar, y decidir la corrección, del siguiente
  argumento
     Juan teme a María. Pedro es temido por Juan. Luego, alguien teme a
     María y a Pedro.
  Usar j      para Juan
       m      para María
       p      para Pedro
```

```
       T(x,y) para x teme a y
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_5a:
  "T(j,m); T(j,p)  x. T(x,m)  T(x,p)"
by auto

-- "La demostración estructurada es"
lemma ejercicio_5b:
  assumes "T(j,m)"
          "T(j,p)"
  shows   "x. T(x,m)  T(x,p)"
proof
  show "T(j,m)  T(j,p)" using assms ..
qed

-- "La demostración detallada es"
lemma ejercicio_5c:
  assumes "T(j,m)"
          "T(j,p)"
  shows   "x. T(x,m)  T(x,p)"
proof (rule exI)
  show "T(j,m)  T(j,p)" using assms by (rule conjI)
qed

text {* ----------------------------------------------------------------
  Ejercicio 6. Formalizar, y decidir la corrección, del siguiente
  argumento
     Los hermanos tienen el mismo padre. Juan es hermano de Luis. Carlos
     es padre de Luis. Por tanto, Carlos es padre de Juan.
  Usar H(x,y) para x es hermano de y
       P(x,y) para x es padre de y
       j      para Juan
       l      para Luis
       c      para Carlos
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_6a:
```

```
  assumes "x y. H(x,y)  H(y,x)"
          "x y z. H(x,y)  P(z,x)  P(z,y)"
          "H(j,l)"
          "P(c,l)"
  shows   "P(c,j)"
using assms
by metis

-- "La demostración estructurada es"
lemma ejercicio_6b:
  fixes H P :: "'a ℂ 'a  bool"
  assumes "x y. H(x,y)  H(y,x)"
          "x y z. H(x,y)  P(z,x)  P(z,y)"
          "H(j,l)"
          "P(c,l)"
  shows   "P(c,j)"
proof -
  have "H(l,j)  P(c,l)  P(c,j)"
  proof -
    have "y z. H(l,y)  P(z,l)  P(z,y)" using assms(2) ..
    hence "z. H(l,j)  P(z,l)  P(z,j)" ..
    thus "H(l,j)  P(c,l)  P(c,j)" ..
  qed
  moreover
  have "H(l,j)  P(c,l)"
  proof
    have "y. H(j,y)  H(y,j)" using assms(1) ..
    hence "H(j,l)  H(l,j)" ..
    thus "H(l,j)" using assms(3) ..
  next
    show "P(c,l)" using assms(4) .
  qed
  ultimately show "P(c,j)" ..
qed

-- "La demostración detallada es"
lemma ejercicio_6c:
  fixes H P :: "'a ℂ 'a  bool"
  assumes "x y. H(x,y)  H(y,x)"
          "x y z. H(x,y)  P(z,x)  P(z,y)"
```

```
          "H(j,l)"
          "P(c,l)"
  shows   "P(c,j)"
proof -
  have "H(l,j)  P(c,l)  P(c,j)"
  proof -
    have "y z. H(l,y)  P(z,l)  P(z,y)" using assms(2) by (rule allE)
    hence "z. H(l,j)  P(z,l)  P(z,j)" by (rule allE)
    thus "H(l,j)  P(c,l)  P(c,j)" by (rule allE)
  qed
  moreover
  have "H(l,j)  P(c,l)"
  proof (rule conjI)
    have "y. H(j,y)  H(y,j)" using assms(1) by (rule allE)
    hence "H(j,l)  H(l,j)" by (rule allE)
    thus "H(l,j)" using assms(3) by (rule mp)
  next
    show "P(c,l)" using assms(4) by this
  qed
  ultimately show "P(c,j)" by (rule mp)
qed

text {* ---------------------------------------------------------------
  Ejercicio 7. Formalizar, y decidir la corrección, del siguiente
  argumento
     La existencia de algún canal de TV pública, supone un acicate para
     cualquier canal de TV privada; el que un canal de TV tenga un
     acicate, supone una gran satisfacción para cualquiera de sus
     directivos; en Madrid hay varios canales públicos de TV; TV5 es un
     canal de TV privada; por tanto, todos los directivos de TV5 están
     satisfechos.
  Usar Pu(x)  para x es un canal de TV pública
       Pr(x)  para x es un canal de TV privada
       A(x)   para x posee un acicate
       D(x,y) para x es un directivo del canal y
       S(x)   para x está satisfecho
       t      para TV5
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
```

```
lemma ejercicio_7a:
  assumes "(x. Pu(x))  (x. Pr(x)  A(x))"
          "x. A(x)  (y. D(y,x)  S(y))"
          "x. Pu(x)"
          "Pr(t)"
  shows   "x. D(x,t)  S(x)"
using assms
by auto

-- "La demostración estructurada es"
lemma ejercicio_7b:
  assumes "(x. Pu(x))  (x. Pr(x)  A(x))"
          "x. A(x)  (y. D(y,x)  S(y))"
          "x. Pu(x)"
          "Pr(t)"
  shows   "x. D(x,t)  S(x)"
proof
  fix a
  show "D(a,t)  S(a)"
  proof
    assume "D(a,t)"
    have "x. Pr(x)  A(x)" using assms(1) assms(3) ..
    hence "Pr(t)  A(t)" ..
    hence "A(t)" using assms(4) ..
    have "A(t)  (y. D(y,t)  S(y))" using assms(2) ..
    hence "y. D(y,t)  S(y)" using `A(t)` ..
    hence "D(a,t)  S(a)" ..
    thus "S(a)" using `D(a,t)` ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_7c:
  assumes "(x. Pu(x))  (x. Pr(x)  A(x))"
          "x. A(x)  (y. D(y,x)  S(y))"
          "x. Pu(x)"
          "Pr(t)"
  shows   "x. D(x,t)  S(x)"
proof (rule allI)
  fix a
```

```
  show "D(a,t)  S(a)"
  proof (rule impI)
    assume "D(a,t)"
    have "x. Pr(x)  A(x)" using assms(1) assms(3) by (rule mp)
    hence "Pr(t)  A(t)" by (rule allE)
    hence "A(t)" using assms(4) by (rule mp)
    have "A(t)  (y. D(y,t)  S(y))" using assms(2) by (rule allE)
    hence "y. D(y,t)  S(y)" using ‘A(t)‘ by (rule mp)
    hence "D(a,t)  S(a)" by (rule allE)
    thus "S(a)" using ‘D(a,t)‘ by (rule mp)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 8. Formalizar, y decidir la corrección, del siguiente
  argumento
     Quien intente entrar en un país y no tenga pasaporte, encontrará
     algún aduanero que le impida el paso. A algunas personas
     motorizadas que intentan entrar en un país le impiden el paso
     únicamente personas motorizadas. Ninguna persona motorizada tiene
     pasaporte. Por tanto, ciertos aduaneros están motorizados.
  Usar E(x)   para x entra en un país
       P(x)   para x tiene pasaporte
       A(x)   para x es aduanero
       I(x,y) para x impide el paso a y
       M(x)   para x está motorizada
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_8a:
  assumes "x. E(x)  ňP(x)  (y. A(y)  I(y,x))"
          "x. M(x)  E(x)  (y. I(y,x)  M(y))"
          "x. M(x)  ňP(x)"
  shows "x. A(x)  M(x)"
using assms
by blast

-- "La demostración estructurada es"
lemma ejercicio_8b:
  assumes "x. E(x)  ňP(x)  (y. A(y)  I(y,x))"
```

```
          "∀x. M(x) → E(x) ∧ (∃y. I(y,x) ∧ M(y))"
          "∀x. M(x) → ¬P(x)"
  shows "∀x. A(x) ∧ M(x)"
proof -
  obtain a where a: "M(a) ∧ E(a) ∧ (∃y. I(y,a) ∧ M(y))" using assms(2) ..
  hence "M(a)" ..
  have "M(a) → ¬P(a)" using assms(3) ..
  hence "¬P(a)" using ‘M(a)‘ ..
  have a1: "E(a) ∧ (∃y. I(y,a) ∧ M(y))" using a ..
  hence "E(a)" ..
  hence "E(a) ∧ ¬P(a)" using ‘¬P(a)‘ ..
  have "E(a) → ¬P(a) → (∃y. A(y) ∧ I(y,a))" using assms(1) ..
  hence "∃y. A(y) ∧ I(y,a)" using ‘E(a) ∧ ¬P(a)‘ ..
  then obtain b where b: "A(b) ∧ I(b,a)" ..
  have "A(b) ∧ M(b)"
  proof
    show "A(b)" using b ..
  next
    have "I(b,a)" using b ..
    have "∃y. I(y,a) ∧ M(y)" using a1 ..
    hence "I(b,a) ∧ M(b)" ..
    thus "M(b)" using ‘I(b,a)‘ ..
  qed
  thus "∀x. A(x) ∧ M(x)" ..
qed

-- "La demostración detallada es"
lemma ejercicio_8c:
  assumes "∀x. E(x) → ¬P(x) → (∃y. A(y) ∧ I(y,x))"
          "∀x. M(x) → E(x) ∧ (∃y. I(y,x) ∧ M(y))"
          "∀x. M(x) → ¬P(x)"
  shows "∀x. A(x) ∧ M(x)"
proof -
  obtain a where a: "M(a) ∧ E(a) ∧ (∃y. I(y,a) ∧ M(y))"
    using assms(2) by (rule exE)
  hence "M(a)" by (rule conjunct1)
  have "M(a) → ¬P(a)" using assms(3) by (rule allE)
  hence "¬P(a)" using ‘M(a)‘ by (rule mp)
  have a1: "E(a) ∧ (∃y. I(y,a) ∧ M(y))" using a by (rule conjunct2)
  hence "E(a)" by (rule conjunct1)
```

```
  hence "E(a)  ňP(a)" using ‘ňP(a)‘ by (rule conjI)
  have "E(a)  ňP(a)  (y. A(y)  I(y,a))"
    using assms(1) by (rule allE)
  hence "y. A(y)  I(y,a)" using ‘E(a)  ňP(a)‘by (rule mp)
  then obtain b where b: "A(b)  I(b,a)" by (rule exE)
  have "A(b)  M(b)"
  proof (rule conjI)
    show "A(b)" using b by (rule conjunct1)
  next
    have "I(b,a)" using b by (rule conjunct2)
    have "y. I(y,a)  M(y)" using a1 by (rule conjunct2)
    hence "I(b,a)  M(b)" by (rule allE)
    thus "M(b)" using ‘I(b,a)‘by (rule mp)
  qed
  thus "x. A(x)  M(x)" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 9. Formalizar, y decidir la corrección, del siguiente
  argumento
     Los aficionados al fútbol aplauden a cualquier futbolista
     extranjero. Juanito no aplaude a futbolistas extranjeros. Por
     tanto, si hay algún futbolista extranjero nacionalizado español,
     Juanito no es aficionado al fútbol.
  Usar Af(x)   para x es aficicionado al fútbol
       Ap(x,y) para x aplaude a y
       E(x)    para x es un futbolista extranjero
       N(x)    para x es un futbolista nacionalizado español
       j       para Juanito
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_9a:
  assumes "x. Af(x)  (y. E(y)  Ap(x,y))"
          "x. E(x)  ňAp(j,x)"
  shows   "(x. E(x)  N(x))  ňAf(j)"
using assms
by blast

-- "La demostración estructurada es"
```

```
lemma ejercicio_9b:
  assumes "x. Af(x)  (y. E(y)  Ap(x,y))"
          "x. E(x)  ňAp(j,x)"
  shows   "(x. E(x)  N(x))  ňAf(j)"
proof
  assume "x. E(x)  N(x)"
  then obtain a where a: "E(a)  N(a)" ..
  show "ňAf(j)"
  proof
    assume "Af(j)"
    have "Af(j)  (y. E(y)  Ap(j,y))" using assms(1) ..
    hence "y. E(y)  Ap(j,y)" using ‘Af(j)‘ ..
    hence "E(a)  Ap(j,a)" ..
    have "E(a)" using a ..
    with ‘E(a)  Ap(j,a)‘ have "Ap(j,a)" ..
    have "E(a)  ňAp(j,a)" using assms(2) ..
    hence "ňAp(j,a)" using ‘E(a)‘ ..
    thus False using ‘Ap(j,a)‘ ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_9c:
  assumes "x. Af(x)  (y. E(y)  Ap(x,y))"
          "x. E(x)  ňAp(j,x)"
  shows   "(x. E(x)  N(x))  ňAf(j)"
proof (rule impI)
  assume "x. E(x)  N(x)"
  then obtain a where a: "E(a)  N(a)" by (rule exE)
  show "ňAf(j)"
  proof (rule notI)
    assume "Af(j)"
    have "Af(j)  (y. E(y)  Ap(j,y))" using assms(1) by (rule allE)
    hence "y. E(y)  Ap(j,y)" using ‘Af(j)‘ by (rule mp)
    hence "E(a)  Ap(j,a)" by (rule allE)
    have "E(a)" using a by (rule conjunct1)
    with ‘E(a)  Ap(j,a)‘ have "Ap(j,a)" by (rule mp)
    have "E(a)  ňAp(j,a)" using assms(2) by (rule allE)
    hence "ňAp(j,a)" using ‘E(a)‘ by (rule mp)
    thus False using ‘Ap(j,a)‘ by (rule notE)
```

```
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 10. Formalizar, y decidir la corrección, del siguiente
  argumento
     Ningún aristócrata debe ser condenado a galeras a menos que sus
     crímenes sean vergonzosos y lleve una vida licenciosa. En la ciudad
     hay aristócratas que han cometido crímenes vergonzosos aunque su
     forma de vida no sea licenciosa. Por tanto, hay algún aristócrata
     que no está condenado a galeras.
  Usar A(x)  para x es aristócrata
       G(x)  para x está condenado a galeras
       L(x)  para x lleva una vida licenciosa
       V(x)  para x ha cometido crímenes vergonzoso
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_10a:
  assumes "x. A(x)  G(x)  L(x)  V(x)"
          "x. A(x)  V(x)  ňL(x)"
  shows   "x. A(x)  ňG(x)"
using assms
by blast

-- "La demostración estructurada es"
lemma ejercicio_10b:
  assumes "x. A(x)  G(x)  L(x)  V(x)"
          "x. A(x)  V(x)  ňL(x)"
  shows   "x. A(x)  ňG(x)"
proof -
  obtain b where b: "A(b)  V(b)  ňL(b)" using assms(2) ..
  have "A(b)  ňG(b)"
  proof
    show "A(b)" using b ..
  next
    show "ňG(b)"
    proof
      assume "G(b)"
      have "ňL(b)"
```

```
      proof -
        have "V(b)  ňL(b)" using b ..
        thus "ňL(b)" ..
      qed
      moreover have "L(b)"
      proof -
        have "A(b)  G(b)  L(b)  V(b)" using assms(1) ..
        have "A(b)" using b ..
        hence "A(b)  G(b)" using ‘G(b)‘ ..
        with ‘A(b)  G(b)  L(b)  V(b)‘ have "L(b)  V(b)" ..
        thus "L(b)" ..
      qed
      ultimately show False ..
    qed
  qed
  thus "x. A(x)  ňG(x)" ..
qed

-- "La demostración detallada es"
lemma ejercicio_10c:
  assumes "x. A(x)  G(x)  L(x)  V(x)"
          "x. A(x)  V(x)  ňL(x)"
  shows   "x. A(x)  ňG(x)"
proof -
  obtain b where b: "A(b)  V(b)  ňL(b)" using assms(2) by (rule exE)
  have "A(b)  ňG(b)"
  proof (rule conjI)
    show "A(b)" using b by (rule conjunct1)
  next
    show "ňG(b)"
    proof (rule notI)
      assume "G(b)"
      have "ňL(b)"
      proof -
        have "V(b)  ňL(b)" using b by (rule conjunct2)
        thus "ňL(b)" by (rule conjunct2)
      qed
      moreover have "L(b)"
      proof -
        have "A(b)  G(b)  L(b)  V(b)" using assms(1) by (rule allE)
```

```
        have "A(b)" using b by (rule conjunct1)
        hence "A(b)  G(b)" using `G(b)` by (rule conjI)
        with `A(b)  G(b)  L(b)  V(b)` have "L(b)  V(b)" by (rule mp)
        thus "L(b)" by (rule conjunct1)
      qed
      ultimately show False by (rule notE)
    qed
  qed
  thus "x. A(x)  ňG(x)" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 11. Formalizar, y decidir la corrección, del siguiente
  argumento
     Todo individuo que esté conforme con el contenido de cualquier
     acuerdo internacional lo apoya o se inhibe en absoluto de asuntos
     políticos. Cualquiera que se inhiba de los asuntos políticos, no
     participará en el próximo referéndum. Todo español, está conforme
     con el acuerdo internacional de Maastricht, al que sin embargo no
     apoya. Por tanto, cualquier individuo o no es español, o en otro
     caso, está conforme con el contenido del acuerdo internacional de
     Maastricht y no participará en el próximo referéndum.
  Usar C(x,y) para la persona x conforme con el contenido del acuerdo y
       A(x,y) para la persona x apoya el acuerdo y
       I(x)   para la persona x se inibe de asuntos políticos
       R(x)   para la persona x participará en el próximo referéndum
       E(x)   para la persona x es española
       m      para el acuerdo de Maastricht
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_11a:
  assumes "x y. C(x,y)  A(x,y)  I(x)"
          "x. I(x)  ňR(x)"
          "x. E(x)  C(x,m)  ňA(x,m)"
  shows   "x. ňE(x)  (C(x,m)  ňR(x))"
using assms
by blast

-- "La demostración estructurada es"
```

```
lemma ejercicio_11b:
  assumes "x y. C(x,y)  A(x,y)  I(x)"
          "x. I(x)  ňR(x)"
          "x. E(x)  C(x,m)  ňA(x,m)"
  shows   "x. ňE(x)  (C(x,m)  ňR(x))"
proof
  fix a
  have "ňE(a)  E(a)" ..
  thus "ňE(a)  (C(a,m)  ňR(a))"
  proof
    assume "ňE(a)"
    thus "ňE(a)  (C(a,m)  ňR(a))" ..
  next
    assume "E(a)"
    have "E(a)  C(a,m)  ňA(a,m)" using assms(3) ..
    hence "C(a,m)  ňA(a,m)" using ‘E(a)‘ ..
    hence "C(a,m)" ..
    have "y. C(a,y)  A(a,y)  I(a)" using assms(1) ..
    hence "C(a,m)  A(a,m)  I(a)" ..
    hence "A(a,m)  I(a)" using ‘C(a,m)‘ ..
    hence "ňR(a)"
    proof
      assume "A(a,m)"
      have "ňA(a,m)" using ‘C(a,m)  ňA(a,m)‘..
      thus "ňR(a)" using ‘A(a,m)‘ ..
    next
      assume "I(a)"
      have "I(a)  ňR(a)" using assms(2) ..
      thus "ňR(a)" using ‘I(a)‘ ..
    qed
    with ‘C(a,m)‘ have "C(a,m)  ňR(a)" ..
    thus "ňE(a)  (C(a,m)  ňR(a))" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_11c:
  assumes "x y. C(x,y)  A(x,y)  I(x)"
          "x. I(x)  ňR(x)"
          "x. E(x)  C(x,m)  ňA(x,m)"
```

```
  shows   "∀x. ¬E(x) ∨ (C(x,m) ∧ ¬R(x))"
proof (rule allI)
  fix a
  have "¬E(a) ∨ E(a)" by (rule excluded_middle)
  thus "¬E(a) ∨ (C(a,m) ∧ ¬R(a))"
  proof (rule disjE)
    assume "¬E(a)"
    thus "¬E(a) ∨ (C(a,m) ∧ ¬R(a))" by (rule disjI1)
  next
    assume "E(a)"
    have "E(a) → C(a,m) ∧ ¬A(a,m)" using assms(3) by (rule allE)
    hence "C(a,m) ∧ ¬A(a,m)" using ‘E(a)‘ by (rule mp)
    hence "C(a,m)" by (rule conjunct1)
    have "∀y. C(a,y) → A(a,y) ∨ I(a)" using assms(1) by (rule allE)
    hence "C(a,m) → A(a,m) ∨ I(a)" by (rule allE)
    hence "A(a,m) ∨ I(a)" using ‘C(a,m)‘ by (rule mp)
    hence "¬R(a)"
    proof (rule disjE)
      assume "A(a,m)"
      have "¬A(a,m)" using ‘C(a,m) ∧ ¬A(a,m)‘by (rule conjunct2)
      thus "¬R(a)" using ‘A(a,m)‘ by (rule notE)
    next
      assume "I(a)"
      have "I(a) → ¬R(a)" using assms(2) by (rule allE)
      thus "¬R(a)" using ‘I(a)‘ by (rule mp)
    qed
    with ‘C(a,m)‘ have "C(a,m) ∧ ¬R(a)" by (rule conjI)
    thus "¬E(a) ∨ (C(a,m) ∧ ¬R(a))" by (rule disjI2)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 12. Formalizar, y decidir la corrección, del siguiente
  argumento
     Toda persona pobre tiene un padre rico. Por tanto, existe una
     persona rica que tiene un abuelo rico.
  Usar R(x) para x es rico
       p(x) para el padre de x
  --------------------------------------------------------------- *}
```

```
-- "La demostración automática es"
lemma ejercicio_12a:
  assumes "∀x. ¬R(x) ∨ R(p(x))"
  shows   "∃x. R(x) ∧ R(p(p(x)))"
using assms
by blast

-- "La demostración estructurada es"
lemma ejercicio_12b:
  assumes "∀x. ¬R(x) ∨ R(p(x))"
  shows   "∃x. R(x) ∧ R(p(p(x)))"
proof -
  have "∃x. R(x)"
  proof (rule ccontr)
    assume "¬(∃x. R(x))"
    hence "∀x. ¬R(x)" by (rule no_ex)
    hence "¬R(a)" ..
    have "¬R(a) ∨ R(p(a))" using assms ..
    hence "R(p(a))" using ‹¬R(a)› ..
    hence "∃x. R(x)" ..
    with ‹¬(∃x. R(x))› show False ..
  qed
  then obtain a where "R(a)" ..
  have "¬R(p(p(a))) ∨ R(p(p(a)))" by (rule excluded_middle)
  thus "∃x. R(x) ∧ R(p(p(x)))"
  proof
    assume "¬R(p(p(a)))"
    have "R(p(a)) ∧ R(p(p(p(a))))"
    proof
      show "R(p(a))"
      proof (rule ccontr)
        assume "¬R(p(a))"
        have "¬R(p(a)) ∨ R(p(p(a)))" using assms ..
        hence "R(p(p(a)))" using ‹¬R(p(a))› ..
        with ‹¬R(p(p(a)))› show False ..
      qed
    next
      show "R(p(p(p(a))))"
      proof -
        have "¬R(p(p(a))) ∨ R(p(p(p(a))))" using assms ..
```

```
        thus "R(p(p(p(a))))" using ‘¬R(p(p(a)))‘ ..
      qed
    qed
    thus "∀x. R(x) ∨ R(p(p(x)))" ..
  next
    assume "R(p(p(a)))"
    with ‘R(a)‘ have "R(a) ∨ R(p(p(a)))" ..
    thus "∀x. R(x) ∨ R(p(p(x)))" ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_12c:
  assumes "∀x. ¬R(x) ∨ R(p(x))"
  shows   "∀x. R(x) ∨ R(p(p(x)))"
proof -
  have "∃x. R(x)"
  proof (rule ccontr)
    assume "¬(∃x. R(x))"
    hence "∀x. ¬R(x)" by (rule no_ex)
    hence "¬R(a)" by (rule allE)
    have "¬R(a) ∨ R(p(a))" using assms by (rule allE)
    hence "R(p(a))" using ‘¬R(a)‘ by (rule mp)
    hence "∃x. R(x)" by (rule exI)
    with ‘¬(∃x. R(x))‘ show False by (rule notE)
  qed
  then obtain a where "R(a)" by (rule exE)
  have "¬R(p(p(a))) ∨ R(p(p(a)))" by (rule excluded_middle)
  thus "∀x. R(x) ∨ R(p(p(x)))"
  proof (rule disjE)
    assume "¬R(p(p(a)))"
    have "R(p(a)) ∨ R(p(p(p(a))))"
    proof (rule conjI)
      show "R(p(a))"
      proof (rule ccontr)
        assume "¬R(p(a))"
        have "¬R(p(a)) ∨ R(p(p(a)))" using assms by (rule allE)
        hence "R(p(p(a)))" using ‘¬R(p(a))‘ by (rule mp)
        with ‘¬R(p(p(a)))‘ show False by (rule notE)
      qed
```

```
    next
      show "R(p(p(p(a))))"
      proof -
        have "¬R(p(p(a)))  R(p(p(p(a))))" using assms by (rule allE)
        thus "R(p(p(p(a))))" using `¬R(p(p(a)))` by (rule mp)
      qed
    qed
    thus "x. R(x)  R(p(p(x)))" by (rule exI)
  next
    assume "R(p(p(a)))"
    with `R(a)` have "R(a)  R(p(p(a)))" by (rule conjI)
    thus "x. R(x)  R(p(p(x)))" by (rule exI)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 13. Formalizar, y decidir la corrección, del siguiente
  argumento
     Todo deprimido que estima a un submarinista es listo. Cualquiera
     que se estime a sí mismo es listo. Ningún deprimido se estima a sí
     mismo. Por tanto, ningún deprimido estima a un submarinista.
  Usar D(x)   para x está deprimido
       E(x,y) para x estima a y
       L(x)   para x es listo
       S(x)   para x es submarinista
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_13a:
  assumes "x. D(x)  (y. S(y)  E(x,y))  L(x)"
          "x. E(x,x)  L(x)"
          "¬(x. D(x)  E(x,x))"
  shows   "¬(x. D(x)  (y. S(y)  E(x,y)))"
quickcheck
oops

text {*
  El argumento es incorrecto como muestra el siguiente contraejemplo:
     Quickcheck found a counterexample:
     D = {a\<^isub>2}
```

```
     S = {a\<^isub>1}
     E = {(a\<^isub>2, a\<^isub>1)}
     L = {a\<^isub>2}
*}

text {* ---------------------------------------------------------------
  Ejercicio 14. Formalizar, y decidir la corrección, del siguiente
  argumento
     Todos los robots obedecen a los amigos del programador jefe.
     Alvaro es amigo del programador jefe, pero Benito no le
     obedece. Por tanto, Benito no es un robot.
  Usar R(x)    para x es un robot
       Ob(x,y) para x obedece a y
       A(x)    para x es amigo del programador jefe
       b       para Benito
       a       para Alvaro
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_14a:
  assumes "∀x y. R(x) ∧ A(y) → Ob(x,y)"
          "A(a)"
          "¬Ob(b,a)"
  shows   "¬R(b)"
using assms
by blast

-- "La demostración estructurada es"
lemma ejercicio_14b:
  fixes R A :: "'a ⇒ bool" and
        Ob :: "'a ⇒ 'a ⇒ bool"
  assumes "∀x y. R(x) ∧ A(y) → Ob(x,y)"
          "A(a)"
          "¬Ob(b,a)"
  shows   "¬R(b)"
proof
  assume "R(b)"
  have "∀y. R(b) ∧ A(y) → Ob(b,y)" using assms(1) ..
  hence "R(b) ∧ A(a) → Ob(b,a)" ..
  have "R(b) ∧ A(a)" using `R(b)` assms(2) ..
```

```
  with ‘R(b)  A(a)  Ob(b,a)‘ have "Ob(b,a)" ..
  with assms(3) show False ..
qed

-- "La demostración detallada es"
lemma ejercicio_14c:
  fixes R A :: "'a  bool" and
        Ob :: "'a ⇒ 'a  bool"
  assumes "∀x y. R(x)  A(y)  Ob(x,y)"
          "∃x. A(x)"
          "∀x. ¬Ob(x,a)"
  shows   "¬R(b)"
proof (rule notI)
  assume "R(b)"
  have "∀y. R(b)  A(y)  Ob(b,y)" using assms(1) by (rule allE)
  hence "R(b)  A(a)  Ob(b,a)" by (rule allE)
  have "R(b)  A(a)" using ‘R(b)‘ assms(2) by (rule conjI)
  with ‘R(b)  A(a)  Ob(b,a)‘ have "Ob(b,a)" by (rule mp)
  with assms(3) show False by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 15. Formalizar, y decidir la corrección, del siguiente
  argumento
     En una pecera nadan una serie de peces. Se observa que:
     * Hay algún pez x que para cualquier pez y, si el pez x no se come
       al pez y entonces existe un pez z tal que z es un tiburón o bien
       z protege al pez y.
     * No hay ningún pez que se coma a todos los demás.
     * Ningún pez protege a ningún otro.
     Por tanto, existe algún tiburón en la pecera.
  Usar C(x,y) para x se come a y
       P(x,y) para x protege a y
       T(x)   para x es un tiburón
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_15a:
  assumes "∀x. ∃y. ¬C(x,y)  (∃z. T(z)  P(z,y))"
          "∀x. ∃y. ¬C(x,y)"
```

```
          "x y. ¬P(x,y)"
  shows   "∃x. T(x)"
using assms
by auto

-- "La demostración estructurada es"
lemma ejercicio_15b:
  assumes "∀x. ∃y. ¬C(x,y) ∧ (∀z. T(z) ∨ P(z,y))"
          "∀x. ∃y. ¬C(x,y)"
          "∀x y. ¬P(x,y)"
  shows   "∃x. T(x)"
proof -
  obtain a where a: "∃y. ¬C(a,y) ∧ (∀z. T(z) ∨ P(z,y))"
    using assms(1) ..
  have "∃y. ¬C(a,y)" using assms(2) ..
  then obtain b where "¬C(a,b)" ..
  have "¬C(a,b) ∧ (∀z. T(z) ∨ P(z,b))" using a ..
  hence "∀z. T(z) ∨ P(z,b)" using ‘¬C(a,b)‘ ..
  then obtain c where "T(c) ∨ P(c,b)" ..
  thus "∃x. T(x)"
  proof
    assume "T(c)"
    thus "∃x. T(x)" ..
  next
    assume "P(c,b)"
    have "∃y. ¬P(c,y)" using assms(3) ..
    hence "¬P(c,b)" ..
    thus "∃x. T(x)" using ‘P(c,b)‘ ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_15c:
  assumes "∀x. ∃y. ¬C(x,y) ∧ (∀z. T(z) ∨ P(z,y))"
          "∀x. ∃y. ¬C(x,y)"
          "∀x y. ¬P(x,y)"
  shows   "∃x. T(x)"
proof -
  obtain a where a: "∃y. ¬C(a,y) ∧ (∀z. T(z) ∨ P(z,y))"
    using assms(1) by (rule exE)
```

```
  have "∀y. ¬C(a,y)" using assms(2) by (rule allE)
  then obtain b where "¬C(a,b)" by (rule exE)
  have "¬C(a,b) ⟶ (∃z. T(z) ∧ P(z,b))" using a by (rule allE)
  hence "∃z. T(z) ∧ P(z,b)" using ‹¬C(a,b)› by (rule mp)
  then obtain c where "T(c) ∧ P(c,b)" by (rule exE)
  thus "∃x. T(x)"
  proof (rule disjE)
    assume "T(c)"
    thus "∃x. T(x)" by (rule exI)
  next
    assume "P(c,b)"
    have "∀y. ¬P(c,y)" using assms(3) by (rule allE)
    hence "¬P(c,b)" by (rule allE)
    thus "∃x. T(x)" using ‹P(c,b)› by (rule notE)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 16. Formalizar, y decidir la corrección, del siguiente
  argumento
     Supongamos conocidos los siguientes hechos acerca del número de
     aprobados de dos asignaturas A y B:
     * Si todos los alumnos aprueban la asignatura A, entonces todos
       aprueban la asignatura B.
     * Si algún delegado de la clase aprueba A y B, entonces todos los
       alumnos aprueban A.
     * Si nadie aprueba B, entonces ningún delegado aprueba A.
     * Si Manuel no aprueba B, entonces nadie aprueba B.
     Por tanto, si Manuel es un delegado y aprueba la asignatura A,
     entonces todos los alumnos aprueban las asignaturas A y B.
  Usar A(x,y) para x aprueba la asignatura y
       D(x)   para x es delegado
       m      para Manuel
       a      para la asignatura A
       b      para la asignatura B
   --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_16a:
  assumes "(∃x. A(x,a)) ⟶ (∀x. A(x,b))"
```

```
          "(x. D(x)  A(x,a)  A(x,b))  (x. A(x,a))"
          "(x. ňA(x,b))  (x. D(x)  ňA(x,a))"
          "ňA(m,b)  (x. ňA(x,b))"
   shows  "D(m)  A(m,a)  (x. A(x,a)  A(x,b))"
using assms
by blast

-- "La demostración estructurada es"
lemma ejercicio_16b:
  assumes "(x. A(x,a))  (x. A(x,b))"
          "(x. D(x)  A(x,a)  A(x,b))  (x. A(x,a))"
          "(x. ňA(x,b))  (x. D(x)  ňA(x,a))"
          "ňA(m,b)  (x. ňA(x,b))"
   shows  "D(m)  A(m,a)  (x. A(x,a)  A(x,b))"
proof
  assume "D(m)  A(m,a)"
  hence "D(m)" ..
  have "A(m,a)" using ‘D(m)  A(m,a)‘ ..
  have "A(m,b)"
  proof (rule ccontr)
    assume "ňA(m,b)"
    with assms(4) have "x. ňA(x,b)" ..
    with assms(3) have "x. D(x)  ňA(x,a)" ..
    hence "D(m)  ňA(m,a)" ..
    hence "ňA(m,a)" using ‘D(m)‘ ..
    thus False using ‘A(m,a)‘ ..
  qed
  with ‘A(m,a)‘ have "A(m,a)  A(m,b)" ..
  with ‘D(m)‘ have "D(m)  A(m,a)  A(m,b)" ..
  hence "x. D(x)  A(x,a)  A(x,b)" ..
  with assms(2) have "x. A(x,a)" ..
  with assms(1) have "x. A(x,b)" ..
  show "x. A(x,a)  A(x,b)"
  proof
    fix c
    show "A(c,a)  A(c,b)"
    proof
      show "A(c,a)" using ‘x. A(x,a)‘ ..
    next
      show "A(c,b)" using ‘x. A(x,b)‘ ..
```

```
    qed
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_16c:
  assumes "(x. A(x,a))  (x. A(x,b))"
          "(x. D(x)  A(x,a)  A(x,b))  (x. A(x,a))"
          "(x. ňA(x,b))  (x. D(x)  ňA(x,a))"
          "ňA(m,b)  (x. ňA(x,b))"
   shows  "D(m)  A(m,a)  (x. A(x,a)  A(x,b))"
proof (rule impI)
  assume "D(m)  A(m,a)"
  hence "D(m)" by (rule conjunct1)
  have "A(m,a)" using ‘D(m)  A(m,a)‘ by (rule conjunct2)
  have "A(m,b)"
  proof (rule ccontr)
    assume "ňA(m,b)"
    with assms(4) have "x. ňA(x,b)" by (rule mp)
    with assms(3) have "x. D(x)  ňA(x,a)" by (rule mp)
    hence "D(m)  ňA(m,a)" by (rule allE)
    hence "ňA(m,a)" using ‘D(m)‘ by (rule mp)
    thus False using ‘A(m,a)‘ by (rule notE)
  qed
  with ‘A(m,a)‘ have "A(m,a)  A(m,b)" by (rule conjI)
  with ‘D(m)‘ have "D(m)  A(m,a)  A(m,b)" by (rule conjI)
  hence "x. D(x)  A(x,a)  A(x,b)" by (rule exI)
  with assms(2) have "x. A(x,a)" by (rule mp)
  with assms(1) have "x. A(x,b)" by (rule mp)
  show "x. A(x,a)  A(x,b)"
  proof (rule allI)
    fix c
    show "A(c,a)  A(c,b)"
    proof (rule conjI)
      show "A(c,a)" using ‘x. A(x,a)‘ by (rule allE)
    next
      show "A(c,b)" using ‘x. A(x,b)‘ by (rule allE)
    qed
  qed
qed
```

```
text {* ---------------------------------------------------------------
  Ejercicio 17. Formalizar, y decidir la corrección, del siguiente
  argumento
     En cierto país oriental se ha celebrado la fase final del
     campeonato mundial de fútbol. Cierto diario deportivo ha publicado
     las siguientes estadísticas de tan magno acontecimiento:
     * A todos los porteros que no vistieron camiseta negra les marcó un
       gol algún delantero europeo.
     * Algún portero jugó con botas blancas y sólo le marcaron goles
       jugadores con botas blancas.
     * Ningún portero se marcó un gol a sí mismo.
     * Ningún jugador con botas blancas vistió camiseta negra.
     Por tanto, algún delantero europeo jugó con botas blancas.
  Usar P(x)   para x es portero
       D(x)   para x es delantero europeo
       N(x)   para x viste camiseta negra
       B(x)   para x juega con botas blancas
       M(x,y) para x marcó un gol a y
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_17a:
  assumes "∀x. P(x) ∧ ¬N(x) ⟶ (∃y. D(y) ∧ M(y,x))"
          "∀x. P(x) ∧ B(x) ⟶ (∃y. M(y,x) ∧ B(y))"
          "¬(∃x. P(x) ∧ M(x,x))"
          "¬(∃x. B(x) ∧ N(x))"
  shows   "∃x. D(x) ∧ B(x)"
using assms
by blast

-- "La demostración estructurada es"
lemma ejercicio_17b:
  assumes "∀x. P(x) ∧ ¬N(x) ⟶ (∃y. D(y) ∧ M(y,x))"
          "∀x. P(x) ∧ B(x) ⟶ (∃y. M(y,x) ∧ B(y))"
          "¬(∃x. P(x) ∧ M(x,x))"
          "¬(∃x. B(x) ∧ N(x))"
  shows   "∃x. D(x) ∧ B(x)"
proof -
  obtain a where a: "P(a) ∧ B(a) ∧ (∃y. M(y,a) ∧ B(y))"
```

```
    using assms(2) ..
  have "P(a)  ¬N(a)"
  proof
    show "P(a)" using a ..
  next
    have "B(a)  (y. M(y,a)  B(y))" using a ..
    hence "B(a)" ..
    have "x. ¬(B(x)  N(x))" using assms(4) by (rule no_ex)
    hence "¬(B(a)  N(a))" ..
    show "¬N(a)"
    proof
      assume "N(a)"
      with `B(a)` have "B(a)  N(a)" ..
      with `¬(B(a)  N(a))` show False ..
    qed
  qed
  hence "y. D(y)  M(y,a)"
  proof -
    have "P(a)  ¬N(a)  (y. D(y)  M(y,a))" using assms(1) ..
    thus "y. D(y)  M(y,a)" using `P(a)  ¬N(a)` ..
  qed
  then obtain b where "D(b)  M(b,a)" ..
  have "D(b)  B(b)"
  proof
    show "D(b)" using `D(b)  M(b,a)` ..
  next
    have "M(b,a)" using `D(b)  M(b,a)` ..
    have "B(a)  (y. M(y,a)  B(y))" using a ..
    hence "y. M(y,a)  B(y)" ..
    hence "M(b,a)  B(b)" ..
    thus "B(b)" using `M(b,a)` ..
  qed
  thus "x. D(x)  B(x)" ..
qed

-- "La demostración estructurada es"
lemma ejercicio_17c:
  assumes "x. P(x)  ¬N(x)  (y. D(y)  M(y,x))"
          "x. P(x)  B(x)  (y. M(y,x)  B(y))"
          "¬(x. P(x)  M(x,x))"
```

```
          "ň(x. B(x)  N(x))"
  shows   "x. D(x)  B(x)"
proof -
  obtain a where a: "P(a)  B(a)  (y. M(y,a)  B(y))"
    using assms(2) by (rule exE)
  have "P(a)  ňN(a)"
  proof (rule conjI)
    show "P(a)" using a by (rule conjunct1)
  next
    have "B(a)  (y. M(y,a)  B(y))" using a by (rule conjunct2)
    hence "B(a)" by (rule conjunct1)
    have "x. ň(B(x)  N(x))" using assms(4) by (rule no_ex)
    hence "ň(B(a)  N(a))" by (rule allE)
    show "ňN(a)"
    proof (rule notI)
      assume "N(a)"
      with ‘B(a)‘ have "B(a)  N(a)" by (rule conjI)
      with ‘ň(B(a)  N(a))‘ show False by (rule notE)
    qed
  qed
  hence "y. D(y)  M(y,a)"
  proof -
    have "P(a)  ňN(a)  (y. D(y)  M(y,a))"
      using assms(1) by (rule allE)
    thus "y. D(y)  M(y,a)" using ‘P(a)  ňN(a)‘ by (rule mp)
  qed
  then obtain b where "D(b)  M(b,a)" by (rule exE)
  have "D(b)  B(b)"
  proof (rule conjI)
    show "D(b)" using ‘D(b)  M(b,a)‘ by (rule conjunct1)
  next
    have "M(b,a)" using ‘D(b)  M(b,a)‘ by (rule conjunct2)
    have "B(a)  (y. M(y,a)  B(y))" using a by (rule conjunct2)
    hence "y. M(y,a)  B(y)" by (rule conjunct2)
    hence "M(b,a)  B(b)" by (rule allE)
    thus "B(b)" using ‘M(b,a)‘ by (rule mp)
  qed
  thus "x. D(x)  B(x)" by (rule exI)
qed
```

```
text {* ---------------------------------------------------------------
  Ejercicio 18. Formalizar, y decidir la corrección, del siguiente
  argumento
     Las relaciones de parentesco verifican la siguientes propiedades
     generales:
     * Si x es hermano de y, entonces y es hermano de x.
     * Todo el mundo es hijo de alguien.
     * Nadie es hijo del hermano de su padre.
     * Cualquier padre de una persona es también padre de todos los
       hermanos de esa persona.
     * Nadie es hijo ni hermano de sí mismo.
     Tenemos los siguientes miembros de la familia Peláez: Don Antonio,
     Don Luis, Antoñito y Manolito y sabemos que Don Antonio y Don Luis
     son hermanos, Antoñito y Manolito son hermanos, y Antoñito es hijo
     de Don Antonio. Por tanto, Don Luis no es el padre de Manolito.
  Usar A       para Don Antonio
       He(x,y) para x es hermano de y
       Hi(x,y) para x es hijo de y
       L       para Don Luis
       a       para Antoñito
       m       para Manolito
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_18a:
  assumes "x y. He(x,y)  He(y,x)"
          "x. y. Hi(x,y)"
          "x y z. Hi(x,y)  He(z,y)  ňHi(x,z)"
          "x y. Hi(x,y)  (z. He(z,x)  Hi(z,y))"
          "x. ňHi(x,x)  ňHe(x,x)"
          "He(A,L)"
          "He(a,m)"
          "Hi(a,A)"
  shows   "ňHi(m,L)"
using assms
by blast

-- "La demostración estructurada es"
lemma ejercicio_18b:
  assumes "x y. He(x,y)  He(y,x)"
```

```
          "x. y. Hi(x,y)"
          "x y z. Hi(x,y)  He(z,y)  ñHi(x,z)"
          "x y. Hi(x,y)  (z. He(z,x)  Hi(z,y))"
          "x. ñHi(x,x)  ñHe(x,x)"
          "He(A,L)"
          "He(a,m)"
          "Hi(a,A)"
  shows   "ñHi(m,L)"
proof
  assume "Hi(m,L)"
  have "He(L,A)"
  proof -
    have "y. He(A,y)  He(y,A)" using assms(1) ..
    hence "He(A,L)  He(L,A)" ..
    thus "He(L,A)" using assms(6) ..
  qed
  have "Hi(a,L)"
  proof -
    have "y. Hi(m,y)  (z. He(z,m)  Hi(z,y))" using assms(4) ..
    hence "Hi(m,L)  (z. He(z,m)  Hi(z,L))" ..
    hence "z. He(z,m)  Hi(z,L)" using ‘Hi(m,L)‘ ..
    hence "He(a,m)  Hi(a,L)" ..
    thus "Hi(a,L)" using assms(7) ..
  qed
  have "ñHi(a,L)"
  proof -
    have "Hi(a,A)  He(L,A)" using assms(8) ‘He(L,A)‘ ..
    have "y z. Hi(a,y)  He(z,y)  ñHi(a,z)" using assms(3) ..
    hence "z. Hi(a,A)  He(z,A)  ñHi(a,z)" ..
    hence "Hi(a,A)  He(L,A)  ñHi(a,L)" ..
    thus "ñHi(a,L)" using ‘Hi(a,A)  He(L,A)‘ ..
  qed
  thus False using ‘Hi(a,L)‘ ..
qed

-- "La demostración detallada es"
lemma ejercicio_18c:
  assumes "x y. He(x,y)  He(y,x)"
          "x. y. Hi(x,y)"
          "x y z. Hi(x,y)  He(z,y)  ñHi(x,z)"
```

```
          "x y. Hi(x,y)  (z. He(z,x)  Hi(z,y))"
          "x. ňHi(x,x)  ňHe(x,x)"
          "He(A,L)"
          "He(a,m)"
          "Hi(a,A)"
  shows   "ňHi(m,L)"
proof (rule notI)
  assume "Hi(m,L)"
  have "He(L,A)"
  proof -
    have "y. He(A,y)  He(y,A)" using assms(1) by (rule allE)
    hence "He(A,L)  He(L,A)" by (rule allE)
    thus "He(L,A)" using assms(6) by (rule mp)
  qed
  have "Hi(a,L)"
  proof -
    have "y. Hi(m,y)  (z. He(z,m)  Hi(z,y))"
      using assms(4) by (rule allE)
    hence "Hi(m,L)  (z. He(z,m)  Hi(z,L))" by (rule allE)
    hence "z. He(z,m)  Hi(z,L)" using ‘Hi(m,L)‘ by (rule mp)
    hence "He(a,m)  Hi(a,L)" by (rule allE)
    thus "Hi(a,L)" using assms(7) by (rule mp)
  qed
  have "ňHi(a,L)"
  proof -
    have "Hi(a,A)  He(L,A)" using assms(8) ‘He(L,A)‘ by (rule conjI)
    have "y z. Hi(a,y)  He(z,y)  ňHi(a,z)"
      using assms(3) by (rule allE)
    hence "z. Hi(a,A)  He(z,A)  ňHi(a,z)" by (rule allE)
    hence "Hi(a,A)  He(L,A)  ňHi(a,L)" by (rule allE)
    thus "ňHi(a,L)" using ‘Hi(a,A)  He(L,A)‘ by (rule mp)
  qed
  thus False using ‘Hi(a,L)‘ by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 19. [Problema del apisonador de Schubert (en inglés,
  "Schuberts steamroller")] Formalizar, y decidir la corrección, del
  siguiente argumento
     Si uno de los miembros del club afeita a algún otro (incluido a
```

```
    sí mismo), entonces todos los miembros del club lo han afeitado
    a él (aunque no necesariamente al mismo tiempo). Guido, Lorenzo,
    Petruccio y Cesare pertenecen al club de barberos. Guido ha
    afeitado a Cesare. Por tanto, Petruccio ha afeitado a Lorenzo.
  Usar g      para Guido
       l      para Lorenzo
       p      para Petruccio
       c      para Cesare
       B(x)   para x es un miembro del club de barberos
       A(x,y) para x ha afeitado a y
   ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma ejercicio_19a:
  assumes "∀x. B(x) → (∃y. B(y) ∧ A(x,y)) ∨ (∃z. B(z) ∧ A(z,x))"
          "B(g)"
          "B(l)"
          "B(p)"
          "B(c)"
          "A(g,c)"
  shows   "A(p,l)"
using assms
by meson

-- "La demostración estructurada es"
lemma ejercicio_19b:
  assumes "∀x. B(x) → (∃y. B(y) ∧ A(x,y)) ∨ (∃z. B(z) ∧ A(z,x))"
          "B(g)"
          "B(l)"
          "B(p)"
          "B(c)"
          "A(g,c)"
  shows   "A(p,l)"
proof -
  have "A(l,g)"
  proof -
    have "B(g) → (∃y. B(y) ∧ A(g,y)) ∨ (∃z. B(z) ∧ A(z,g))"
      using assms(1) ..
    have "B(c) ∧ A(g,c)" using assms(5,6) ..
    hence "(∃y. B(y) ∧ A(g,y))" ..
```

```
    with assms(2) have "B(g)  (y. B(y)  A(g,y))" ..
    with `B(g)  (y. B(y)  A(g,y))  (z. B(z)  A(z,g))`
    have "(z. B(z)  A(z,g))" ..
    hence "B(l)  A(l,g)" ..
    thus "A(l,g)" using assms(3) ..
  qed
  have "B(l)  (y. B(y)  A(l,y))  (z. B(z)  A(z,l))"
    using assms(1) ..
  have "B(g)  A(l,g)" using assms(2) `A(l,g)` ..
  hence "(y. B(y)  A(l,y))" ..
  with assms(3) have "B(l)  (y. B(y)  A(l,y))" ..
  with `B(l)  (y. B(y)  A(l,y))  (z. B(z)  A(z,l))`
  have "(z. B(z)  A(z,l))" ..
  hence "B(p)  A(p,l)" ..
  thus "A(p,l)" using assms(4) ..
qed

-- "La demostración detallada es"
lemma ejercicio_19c:
  assumes "x. B(x)  (y. B(y)  A(x,y))  (z. B(z)  A(z,x))"
          "B(g)"
          "B(l)"
          "B(p)"
          "B(c)"
          "A(g,c)"
  shows   "A(p,l)"
proof -
  have "A(l,g)"
  proof -
    have "B(g)  (y. B(y)  A(g,y))  (z. B(z)  A(z,g))"
      using assms(1) by (rule allE)
    have "B(c)  A(g,c)" using assms(5,6) by (rule conjI)
    hence "(y. B(y)  A(g,y))" by (rule exI)
    with assms(2) have "B(g)  (y. B(y)  A(g,y))" by (rule conjI)
    with `B(g)  (y. B(y)  A(g,y))  (z. B(z)  A(z,g))`
    have "(z. B(z)  A(z,g))" by (rule mp)
    hence "B(l)  A(l,g)" by (rule allE)
    thus "A(l,g)" using assms(3) by (rule mp)
  qed
  have "B(l)  (y. B(y)  A(l,y))  (z. B(z)  A(z,l))"
```

```
    using assms(1) by (rule allE)
  have "B(g)  A(l,g)" using assms(2) `A(l,g)` by (rule conjI)
  hence "(y. B(y)  A(l,y))" by (rule exI)
  with assms(3) have "B(l)  (y. B(y)  A(l,y))" by (rule conjI)
  with `B(l)  (y. B(y)  A(l,y))  (z. B(z)  A(z,l))`
  have "(z. B(z)  A(z,l))" by (rule mp)
  hence "B(p)  A(p,l)" by (rule allE)
  thus "A(p,l)" using assms(4) by (rule mp)
qed

text {* ---------------------------------------------------------------
  Ejercicio 20. Formalizar, y decidir la corrección, del siguiente
  argumento
     Carlos afeita a todos los habitantes de Las Chinas que no se
     afeitan a sí mismo y sólo a ellos. Carlos es un habitante de las
     Chinas. Por consiguiente, Carlos no afeita a nadie.
  Usar A(x,y) para x afeita a y
       C(x)   para x es un habitante de Las Chinas
       c      para Carlos
   --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_20a:
  assumes "x. A(c,x)  C(x)  ňA(x,x)"
          "C(c)"
  shows   "ň(x. A(c,x))"
using assms
by blast

-- "La demostración estructurada es"
lemma ejercicio_20b:
  assumes "x. A(c,x)  C(x)  ňA(x,x)"
          "C(c)"
  shows   "ň(x. A(c,x))"
proof -
  have 1: "A(c,c)  C(c)  ňA(c,c)" using assms(1) ..
  have "A(c,c)"
  proof (rule ccontr)
    assume "ňA(c,c)"
    with assms(2) have "C(c)  ňA(c,c)" ..
```

```
    with 1 have "A(c,c)" ..
    with ‘ňA(c,c)‘ show False ..
  qed
  have "ňA(c,c)"
  proof -
    have "C(c)  ňA(c,c)" using 1 ‘A(c,c)‘ ..
    thus "ňA(c,c)" ..
  qed
  thus "ň(x. A(c,x))" using ‘A(c,c)‘ ..
qed

-- "La demostración detallada es"
lemma ejercicio_20c:
  assumes "x. A(c,x)  C(x)  ňA(x,x)"
          "C(c)"
  shows   "ň(x. A(c,x))"
proof -
  have 1: "A(c,c)  C(c)  ňA(c,c)" using assms(1) by (rule allE)
  have "A(c,c)"
  proof (rule ccontr)
    assume "ňA(c,c)"
    with assms(2) have "C(c)  ňA(c,c)" by (rule conjI)
    with 1 have "A(c,c)" by (rule iffD2)
    with ‘ňA(c,c)‘ show False by (rule notE)
  qed
  have "ňA(c,c)"
  proof -
    have "C(c)  ňA(c,c)" using 1 ‘A(c,c)‘ by (rule iffD1)
    thus "ňA(c,c)" by (rule conjunct2)
  qed
  thus "ň(x. A(c,x))" using ‘A(c,c)‘ by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 21. Formalizar, y decidir la corrección, del siguiente
  argumento
     Quien desprecia a todos los fanáticos desprecia también a todos los
     políticos. Alguien no desprecia a un determinado político. Por
     consiguiente, hay un fanático al que no todo el mundo desprecia.
   Usar D(x,y) para x desprecia a y
```

```
        F(x)   para x es fanático
        P(x)   para x es político
    ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma ejercicio_21a:
  assumes "x. (y. F(y)  D(x,y))   (y. P(y)  D(x,y))"
          "x y. P(y)  ňD(x,y)"
  shows   "x. F(x)  ň(y. D(y,x))"
using assms
by blast

-- "La demostración semiautomática es"
lemma ejercicio_21b:
  assumes "x. (y. F(y)  D(x,y))   (y. P(y)  D(x,y))"
          "x y. P(y)  ňD(x,y)"
  shows   "x. F(x)  ň(y. D(y,x))"
proof -
  obtain a where "y. P(y)  ňD(a,y)" using assms(2) ..
  then obtain b where "P(b)  ňD(a,b)" ..
  hence "ň(y. P(y)  D(a,y))" by auto
  have "(y. F(y)  D(a,y))   (y. P(y)  D(a,y))" using assms(1) ..
  hence "ň(y. F(y)  D(a,y))" using `ň(y. P(y)  D(a,y))` by (rule mt)
  hence "y. ň(F(y)  D(a,y))" by (rule no_para_todo)
  then obtain c where "ň(F(c)  D(a,c))" ..
  hence "F(c)  ň(y. D(y,c))" by auto
  thus "x. F(x)  ň(y. D(y,x))" ..
qed

-- "En la demostración estructurada usaremos el siguiente lema"
lemma no_implica:
  assumes "ň(p  q)"
  shows   "p  ňq"
using assms
by auto

-- "La demostración estructurada del lema es"
lemma no_implica_1:
  assumes "ň(p  q)"
  shows   "p  ňq"
```

```
proof
  show "p"
  proof (rule ccontr)
    assume "¬p"
    have "p ∨ q"
    proof
      assume "p"
      with ‘¬p‘ show "q" ..
    qed
    with assms show False ..
  qed
next
  show "¬q"
  proof
    assume "q"
    have "p ∨ q"
    proof
      assume "p"
      show "q" using ‘q‘ .
    qed
    with assms show False ..
  qed
qed

-- "La demostración detallada del lema es"
lemma no_implica_2:
  assumes "¬(p → q)"
  shows   "p ∧ ¬q"
proof (rule conjI)
  show "p"
  proof (rule ccontr)
    assume "¬p"
    have "p ∨ q"
    proof (rule impI)
      assume "p"
      with ‘¬p‘ show "q" by (rule notE)
    qed
    with assms show False by (rule notE)
  qed
next
```

```
  show "ň q"
  proof (rule notI)
    assume "q"
    have "p  q"
    proof (rule impI)
      assume "p"
      show "q" using ‘q‘ by this
    qed
    with assms show False by (rule notE)
  qed
qed

-- "La demostración estructurada es"
lemma ejercicio_21c:
  assumes "x. (y. F(y)  D(x,y))  (y. P(y)  D(x,y))"
          "x y. P(y)  ňD(x,y)"
  shows   "x. F(x)  ň(y. D(y,x))"
proof -
  obtain a where "y. P(y)  ňD(a,y)" using assms(2) ..
  then obtain b where b: "P(b)  ňD(a,b)" ..
  have "ň(y. P(y)  D(a,y))"
  proof
    assume "y. P(y)  D(a,y)"
    hence "P(b)  D(a,b)" ..
    have "P(b)" using b ..
    with ‘P(b)  D(a,b)‘ have "D(a,b)" ..
    have "ňD(a,b)" using b ..
    thus False using ‘D(a,b)‘ ..
  qed
  have "(y. F(y)  D(a,y))  (y. P(y)  D(a,y))" using assms(1) ..
  hence "ň(y. F(y)  D(a,y))" using ‘ň(y. P(y)  D(a,y))‘ by (rule mt)
  hence "y. ň(F(y)  D(a,y))" by (rule no_para_todo)
  then obtain c where c: "ň(F(c)  D(a,c))" ..
  have "F(c)  ň(y. D(y,c))"
  proof
    have "F(c)  ňD(a,c)" using c by (rule no_implica)
    thus "F(c)" ..
  next
    show "ň(y. D(y,c))"
    proof
```

```
      assume "∀y. D(y,c)"
      hence "D(a,c)" ..
      have "F(c) ⟶ ¬D(a,c)" using c by (rule no_implica)
      hence "¬D(a,c)" ..
      thus False using ‹D(a,c)› ..
    qed
  qed
  thus "∃x. F(x) ⟶ ¬(∀y. D(y,x))" ..
qed

-- "La demostración detallada es"
lemma ejercicio_21d:
  assumes "∃x. (∀y. F(y) ⟶ D(x,y)) ⟶ (∀y. P(y) ⟶ D(x,y))"
          "∀x y. P(y) ⟶ ¬D(x,y)"
  shows   "∃x. F(x) ⟶ ¬(∀y. D(y,x))"
proof -
  obtain a where "∀y. P(y) ⟶ ¬D(a,y)" using assms(2) by (rule exE)
  then obtain b where b: "P(b) ⟶ ¬D(a,b)" by (rule exE)
  have "¬(∀y. P(y) ⟶ D(a,y))"
  proof (rule notI)
    assume "∀y. P(y) ⟶ D(a,y)"
    hence "P(b) ⟶ D(a,b)" by (rule allE)
    have "P(b)" using b by (rule conjunct1)
    with ‹P(b) ⟶ D(a,b)› have "D(a,b)" by (rule mp)
    have "¬D(a,b)" using b by (rule conjunct2)
    thus False using ‹D(a,b)› by (rule notE)
  qed
  have "(∀y. F(y) ⟶ D(a,y)) ⟶ (∀y. P(y) ⟶ D(a,y))"
    using assms(1) by (rule allE)
  hence "¬(∀y. F(y) ⟶ D(a,y))" using ‹¬(∀y. P(y) ⟶ D(a,y))› by (rule mt)
  hence "∃y. ¬(F(y) ⟶ D(a,y))" by (rule no_para_todo)
  then obtain c where c: "¬(F(c) ⟶ D(a,c))" by (rule exE)
  have "F(c) ⟶ ¬(∀y. D(y,c))"
  proof (rule conjI)
    have "F(c) ⟶ ¬D(a,c)" using c by (rule no_implica)
    thus "F(c)" by (rule conjunct1)
  next
    show "¬(∀y. D(y,c))"
    proof (rule notI)
      assume "∀y. D(y,c)"
```

```
      hence "D(a,c)" by (rule allE)
      have "F(c)  ňD(a,c)" using c by (rule no_implica)
      hence "ňD(a,c)" by (rule conjunct2)
      thus False using ‘D(a,c)‘ by (rule notE)
    qed
  qed
  thus "x. F(x)  ň(y. D(y,x))" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 22. Formalizar, y decidir la corrección, del siguiente
  argumento
     El hombre puro ama todo lo que es puro. Por tanto, el hombre puro
     se ama a sí mismo.
  Usar A(x,y) para x ama a y
       H(x)   para x es un hombre
       P(x)   para x es puro
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_22a:
  assumes "x. H(x)  P(x)  (y. P(y)  A(x,y))"
  shows   "x. H(x)  P(x)  A(x,x)"
using assms
by auto

-- "La demostración estructurada es"
lemma ejercicio_22b:
  assumes "x. H(x)  P(x)  (y. P(y)  A(x,y))"
  shows   "x. H(x)  P(x)  A(x,x)"
proof
  fix b
  show "H(b)  P(b)  A(b,b)"
  proof
    assume "H(b)  P(b)"
    hence "P(b)" ..
    have "H(b)  P(b)  (y. P(y)  A(b,y))" using assms ..
    hence "y. P(y)  A(b,y)" using ‘H(b)  P(b)‘ ..
    hence "P(b)  A(b,b)" ..
    thus "A(b,b)" using ‘P(b)‘ ..
```

```
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_22c:
  assumes "x. H(x)  P(x)  (y. P(y)  A(x,y))"
  shows   "x. H(x)  P(x)  A(x,x)"
proof (rule allI)
  fix b
  show "H(b)  P(b)  A(b,b)"
  proof (rule impI)
    assume "H(b)  P(b)"
    hence "P(b)" by (rule conjunct2)
    have "H(b)  P(b)  (y. P(y)  A(b,y))"
      using assms by (rule allE)
    hence "y. P(y)  A(b,y)" using `H(b)  P(b)` by (rule mp)
    hence "P(b)  A(b,b)" by (rule allE)
    thus "A(b,b)" using `P(b)` by (rule mp)
  qed
qed

text {* ---------------------------------------------------------------
  Ejercicio 23. Formalizar, y decidir la corrección, del siguiente
  argumento
     Ningún socio del club está en deuda con el tesorero del club. Si
     un socio del club no paga su cuota está en deuda con el tesorero
     del club. Por tanto, si el tesorero del club es socio del club,
     entonces paga su cuota.
  Usar P(x) para x es socio del club
       Q(x) para x paga su cuota
       R(x) para x está en deuda con el tesorero
       a    para el tesorero del club
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_23a:
  assumes "ň(x. P(x)  R(x))"
          "x. P(x)  ňQ(x)  R(x)"
  shows   "P(a)  Q(a)"
using assms
```

```
by auto

-- "La demostración estructurada es"
lemma ejercicio_23b:
  assumes "ň(x. P(x)  R(x))"
          "x. P(x)  ňQ(x)  R(x)"
  shows   "P(a)  Q(a)"
proof
  assume "P(a)"
  show "Q(a)"
  proof (rule ccontr)
    assume "ňQ(a)"
    with ‘P(a)‘ have "P(a)  ňQ(a)" ..
    have "P(a)  ňQ(a)  R(a)" using assms(2) ..
    hence "R(a)" using ‘P(a)  ňQ(a)‘ ..
    with ‘P(a)‘ have "P(a)  R(a)" ..
    hence "x. P(x)  R(x)" ..
    with assms(1) show False ..
  qed
qed

-- "La demostración detallada es"
lemma ejercicio_23c:
  assumes "ň(x. P(x)  R(x))"
          "x. P(x)  ňQ(x)  R(x)"
  shows   "P(a)  Q(a)"
proof (rule impI)
  assume "P(a)"
  show "Q(a)"
  proof (rule ccontr)
    assume "ňQ(a)"
    with ‘P(a)‘ have "P(a)  ňQ(a)" by (rule conjI)
    have "P(a)  ňQ(a)  R(a)" using assms(2) by (rule allE)
    hence "R(a)" using ‘P(a)  ňQ(a)‘ by (rule mp)
    with ‘P(a)‘ have "P(a)  R(a)" by (rule conjI)
    hence "x. P(x)  R(x)" by (rule exI)
    with assms(1) show False by (rule notE)
  qed
qed
```

```
text {* ---------------------------------------------------------------
  Ejercicio 24. Formalizar, y decidir la corrección, del siguiente
  argumento
     1. Los lobos, zorros, pájaros, orugas y caracoles son animales y
        existen algunos ejemplares de estos animales.
     2. También hay algunas semillas y las semillas son plantas.
     3. A todo animal le gusta o bien comer todo tipo de plantas o bien
        le gusta comerse a todos los animales más pequeños que él mismo
        que gustan de comer algunas plantas.
     4. Las orugas y los caracoles son mucho más pequeños que los
        pájaros, que son mucho más pequeños que los zorros que a su vez
        son mucho más pequeños que los lobos.
     5. A los lobos no les gusta comer ni zorros ni semillas, mientras
        que a los pájaros les gusta comer orugas pero no caracoles.
     6. Las orugas y los caracoles gustan de comer algunas plantas.
     7. Luego, existe un animal al que le gusta comerse un animal al que
        le gusta comer semillas.
  Usar A(x)    para x es un animal
       Ca(x)   para x es un caracol
       Co(x,y) para x le gusta comerse a y
       L(x)    para x es un lobo
       M(x,y)  para x es más pequeño que y
       Or(x)   para x es una oruga
       Pa(x)   para x es un pájaro
       Pl(x)   para x es una planta
       S(x)    para x es una semilla
       Z(x)    para x es un zorro
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_24a:
  assumes
   (*  1 *) "∃x. L(x) ∧ A(x)"
   (*  2 *) "∃x. Z(x) ∧ A(x)"
   (*  3 *) "∃x. Pa(x) ∧ A(x)"
   (*  4 *) "∃x. Or(x) ∧ A(x)"
   (*  5 *) "∃x. Ca(x) ∧ A(x)"
   (*  6 *) "∃x. L(x)"
   (*  7 *) "∃x. Z(x)"
   (*  8 *) "∃x. Pa(x)"
```

```
    (*  9 *) "x. Or(x)"
    (* 10 *) "x. Ca(x)"
    (* 11 *) "x. S(x)"
    (* 12 *) "x. S(x)  Pl(x)"
    (* 13 *) "x. A(x)
                 (y. Pl(y)  Co(x,y))
                 (y. A(y)  M(y,x)  (z. Pl(z)  Co(y,z))  Co(x,y))"
    (* 14 *) "x y. Pa(y)  (Ca(x)  Or(x))  M(x,y)"
    (* 15 *) "x y. Pa(x)  Z(y)  M(x,y)"
    (* 16 *) "x y. Z(x)  L(y)  M(x,y)"
    (* 17 *) "x y. L(x)  (Z(y) S(y))  ňCo(x,y)"
    (* 18 *) "x y. Pa(x)  Or(y)  Co(x,y)"
    (* 19 *) "x y. Pa(x)  Ca(y)  ňCo(x,y)"
    (* 20 *) "x. Or(x)  Ca(x)  (y. Pl(y)  Co(x,y))"
  shows
   "x y. A(x)  A(y)  (z. S(z)  Co(y,z)  Co(x,y))"
using assms
by meson

-- "La demostración semiautomática es"
lemma ejercicio_24b:
  assumes
    (*  1 *) "x. L(x)  A(x)"
    (*  2 *) "x. Z(x)  A(x)"
    (*  3 *) "x. Pa(x)  A(x)"
    (*  4 *) "x. Or(x)  A(x)"
    (*  5 *) "x. Ca(x)  A(x)"
    (*  6 *) "x. L(x)"
    (*  7 *) "x. Z(x)"
    (*  8 *) "x. Pa(x)"
    (*  9 *) "x. Or(x)"
    (* 10 *) "x. Ca(x)"
    (* 11 *) "x. S(x)"
    (* 12 *) "x. S(x)  Pl(x)"
    (* 13 *) "x. A(x)
                 (y. Pl(y)  Co(x,y))
                 (y. A(y)  M(y,x)  (z. Pl(z)  Co(y,z))  Co(x,y))"
    (* 14 *) "x y. Pa(y)  (Ca(x)  Or(x))  M(x,y)"
    (* 15 *) "x y. Pa(x)  Z(y)  M(x,y)"
    (* 16 *) "x y. Z(x)  L(y)  M(x,y)"
```

```
    (* 17 *) "x y. L(x)  (Z(y) S(y))  ňCo(x,y)"
    (* 18 *) "x y. Pa(x)  Or(y)  Co(x,y)"
    (* 19 *) "x y. Pa(x)  Ca(y)  ňCo(x,y)"
    (* 20 *) "x. Or(x)  Ca(x)  (y. Pl(y)  Co(x,y))"
  shows
    "x y. A(x)  A(y)  (z. S(z)  Co(y,z)  Co(x,y))"
proof -
  obtain l where l: "L(l)" using assms(6) ..
  obtain z where z: "Z(z)" using assms(7) ..
  obtain p where p: "Pa(p)" using assms(8) ..
  obtain c where c: "Ca(c)" using assms(10) ..
  obtain s where s: "S(s)" using assms(11) ..
  have 1: "Co(p,s)" using p c s assms(3,5,12,13,14,19,20) by meson
  have 2: "ňCo(z,s)" using z l s assms(1,2,12,16,17,13) by meson
  have 3: "M(p,z)" using p z assms(15) by auto
  have 4: "Co(z,p)" using z p s 1 2 3 assms(2,3,12,13) by meson
  hence "Co(z,p)  Co(p,s)" using 1 ..
  thus "x y. A(x)  A(y)  (z. S(z)  Co(y,z)  Co(x,y))"
    using z p s assms(2,3) by meson
qed

lemma instancia:
  assumes "x. P(x)  Q(x)"
          "P(a)"
  shows   "Q(a)"
proof -
  have "P(a)  Q(a)" using assms(1) ..
  thus "Q(a)" using assms(2) ..
qed

lemma mt2:
  assumes "p  q  r  s"
          "p"
          "q"
          "ňs"
  shows   "ňr"
proof
  assume "r"
  with ‘q‘ have "q  r" ..
  with ‘p‘ have "p  q  r" ..
```

```
  with assms(1) have "s" ..
  with ‘ñs‘ show False ..
qed

lemma mp3:
  assumes "p  q  r  s"
          "p"
          "q"
          "r"
  shows   "s"
proof -
  have "q  r" using assms(3,4) ..
  with ‘p‘ have "p  q  r" ..
  with assms(1) show "s" ..
qed

lemma conjI3:
  assumes "p"
          "q"
          "r"
  shows   "p  q  r"
proof -
  have "q  r" using assms(2,3) ..
  with ‘p‘ show "p  q  r" ..
qed

-- "La demostración estructurada es"
lemma ejercicio_24c:
  assumes
    (*  1 *) "x. L(x)  A(x)"
    (*  2 *) "x. Z(x)  A(x)"
    (*  3 *) "x. Pa(x)  A(x)"
    (*  4 *) "x. Or(x)  A(x)"
    (*  5 *) "x. Ca(x)  A(x)"
    (*  6 *) "x. L(x)"
    (*  7 *) "x. Z(x)"
    (*  8 *) "x. Pa(x)"
    (*  9 *) "x. Or(x)"
    (* 10 *) "x. Ca(x)"
    (* 11 *) "x. S(x)"
```

```
    (* 12 *) "x. S(x)  Pl(x)"
    (* 13 *) "x. A(x)
                  (y. Pl(y)  Co(x,y))
                  (y. A(y)  M(y,x)  (z. Pl(z)  Co(y,z))  Co(x,y))"
    (* 14 *) "x y. Pa(y)  (Ca(x)  Or(x))  M(x,y)"
    (* 15 *) "x y. Pa(x)  Z(y)  M(x,y)"
    (* 16 *) "x y. Z(x)  L(y)  M(x,y)"
    (* 17 *) "x y. L(x)  (Z(y) S(y))  ňCo(x,y)"
    (* 18 *) "x y. Pa(x)  Or(y)  Co(x,y)"
    (* 19 *) "x y. Pa(x)  Ca(y)  ňCo(x,y)"
    (* 20 *) "x. Or(x)  Ca(x)  (y. Pl(y)  Co(x,y))"
  shows
   "x y. A(x)  A(y)  (z. S(z)  Co(y,z)  Co(x,y))"
proof -
  obtain l where l: "L(l)" using assms(6) ..
  obtain z where z: "Z(z)" using assms(7) ..
  obtain p where p: "Pa(p)" using assms(8) ..
  obtain c where c: "Ca(c)" using assms(10) ..
  obtain s where s: "S(s)" using assms(11) ..
  have 1: "Co(p,s)"
  proof -
    have "ňCo(p,c)"
    proof -
      have "Pa(p)  Ca(c)" using p c ..
      have "y. Pa(p)  Ca(y)  ňCo(p,y)" using assms(19) ..
      hence "Pa(p)  Ca(c)  ňCo(p,c)" ..
      thus "ňCo(p,c)" using ‘Pa(p)  Ca(c)‘ ..
    qed
    have "y. Pl(y)  Co(c,y)"
    proof -
      have "Or(c)  Ca(c)" using c ..
      have "Or(c)  Ca(c)  (y. Pl(y)  Co(c,y))" using assms(20) ..
      thus "y. Pl(y)  Co(c,y)" using ‘Or(c)  Ca(c)‘ ..
    qed
    have "M(c,p)"
    proof -
      have "y. Pa(y)  (Ca(c)  Or(c))  M(c,y)" using assms(14) ..
      hence "Pa(p)  (Ca(c)  Or(c))  M(c,p)" ..
      have "Ca(c)  Or(c)" using c ..
      with p have "Pa(p)  (Ca(c)  Or(c))" ..
```

```
    with ‘Pa(p)  (Ca(c)  Or(c))  M(c,p)‘ show "M(c,p)" ..
  qed
  show "Co(p,s)"
  proof -
    have "A(p)" using assms(3) p by (rule instancia)
    have "A(p)  (y. Pl(y)  Co(p,y))
                  (y. A(y)  M(y,p)  (z. Pl(z)  Co(y,z))  Co(p,y))"
      using assms(13) ..
    hence "(y. Pl(y)  Co(p,y))
           (y. A(y)  M(y,p)  (z. Pl(z)  Co(y,z))  Co(p,y))"
      using ‘A(p)‘ ..
    thus "Co(p,s)"
    proof
      assume "y. Pl(y)  Co(p,y)"
      hence "Pl(s)  Co(p,s)" ..
      have "Pl(s)" using assms(12) s by (rule instancia)
      with ‘Pl(s)  Co(p,s)‘ show "Co(p,s)" ..
    next
      assume "y. A(y)  M(y,p)  (z. Pl(z)  Co(y,z))  Co(p,y)"
      hence "A(c)  M(c,p)  (z. Pl(z)  Co(c,z))  Co(p,c)" ..
      have "Co(p,c)"
      proof -
        have "A(c)" using assms(5) c by (rule instancia)
        have "M(c,p)  (z. Pl(z)  Co(c,z))"
          using ‘M(c,p)‘ ‘z. Pl(z)  Co(c,z)‘ ..
        with ‘A(c)‘ have "A(c)  M(c,p)  (z. Pl(z)  Co(c,z))" ..
        with ‘A(c)  M(c,p)  (z. Pl(z)  Co(c,z))  Co(p,c)‘
        show "Co(p,c)" ..
      qed
      with ‘ňCo(p,c)‘ show "Co(p,s)" ..
    qed
  qed
qed
have 2: "ňCo(z,s)"
proof -
  have "M(z,l)"
  proof -
    have "Z(z)  L(l)" using z l ..
    have "y. Z(z)  L(y)  M(z,y)" using assms(16) ..
    hence "Z(z)  L(l)  M(z,l)" ..
```

```
    thus "M(z,l)" using ‘Z(z)  L(l)‘ ..
  qed
  have "ňCo(l,z)"
  proof -
    have "y. L(l)  (Z(y) S(y))  ňCo(l,y)" using assms(17) ..
    hence "L(l)  (Z(z) S(z))  ňCo(l,z)" ..
    have "Z(z) S(z)" using z ..
    with l have "L(l)  (Z(z) S(z))" ..
    with ‘L(l)  (Z(z) S(z))  ňCo(l,z)‘ show "ňCo(l,z)" ..
  qed
  have "ňCo(l,s)"
  proof -
    have "y. L(l)  (Z(y) S(y))  ňCo(l,y)" using assms(17) ..
    hence "L(l)  (Z(s) S(s))  ňCo(l,s)" ..
    have "Z(s) S(s)" using s ..
    with l have "L(l)  (Z(s) S(s))" ..
    with ‘L(l)  (Z(s) S(s))  ňCo(l,s)‘ show "ňCo(l,s)" ..
  qed
  show "ňCo(z,s)"
  proof -
    have "A(l)" using assms(1) l by (rule instancia)
    have "A(l)  (y. Pl(y)  Co(l,y))
                  (y. A(y)  M(y,l)  (z. Pl(z)  Co(y,z))  Co(l,y))"
      using assms(13) ..
    hence  "(y. Pl(y)  Co(l,y))
            (y. A(y)  M(y,l)  (z. Pl(z)  Co(y,z))  Co(l,y))"
      using ‘A(l)‘ ..
    thus "ňCo(z,s)"
    proof
      assume "y. Pl(y)  Co(l,y)"
      hence "Pl(s)  Co(l,s)" ..
      have "Pl(s)" using assms(12) s by (rule instancia)
      with ‘Pl(s)  Co(l,s)‘ have "Co(l,s)" ..
      with ‘ňCo(l,s)‘ show "ňCo(z,s)" ..
    next
      assume "y. A(y)  M(y,l)  (u. Pl(u)  Co(y,u))  Co(l,y)"
      hence zl: "A(z)  M(z,l)  (u. Pl(u)  Co(z,u))  Co(l,z)" ..
      have "A(z)" using assms(2) z by (rule instancia)
      have "ň(u. Pl(u)  Co(z,u))"
        using zl ‘A(z)‘ ‘M(z,l)‘ ‘ňCo(l,z)‘ by (rule mt2)
```

```
      show "ňCo(z,s)"
      proof
        assume "Co(z,s)"
        have "Pl(s)" using assms(12) s by (rule instancia)
        hence "Pl(s)  Co(z,s)" using ‘Co(z,s)‘ ..
        hence "u. Pl(u)  Co(z,u)" ..
        with ‘ň(u. Pl(u)  Co(z,u))‘ show False ..
      qed
    qed
  qed
qed
have 3: "M(p,z)"
proof -
  have "Pa(p)  Z(z)" using p z ..
  have "y. Pa(p)  Z(y)  M(p,y)" using assms(15) ..
  hence "Pa(p)  Z(z)  M(p,z)" ..
  thus "M(p,z)" using ‘Pa(p)  Z(z)‘ ..
qed
have 4: "Co(z,p)"
proof -
  have "A(z)" using assms(2) z by (rule instancia)
  have "A(z)  (y. Pl(y)  Co(z,y))
                (y. A(y)  M(y,z)  (u. Pl(u)  Co(y,u))  Co(z,y))"
    using assms(13) ..
  hence "(y. Pl(y)  Co(z,y))
         (y. A(y)  M(y,z)  (u. Pl(u)  Co(y,u))  Co(z,y))"
    using ‘A(z)‘ ..
  thus "Co(z,p)"
  proof
    assume "y. Pl(y)  Co(z,y)"
    hence "Pl(s)  Co(z,s)" ..
    have "Pl(s)" using assms(12) s by (rule instancia)
    with ‘Pl(s)  Co(z,s)‘ have "Co(z,s)" ..
    with ‘ňCo(z,s)‘ show "Co(z,p)" ..
  next
    assume "y. A(y)  M(y,z)  (u. Pl(u)  Co(y,u))  Co(z,y)"
    hence pz: "A(p)  M(p,z)  (u. Pl(u)  Co(p,u))  Co(z,p)" ..
    have "A(p)" using assms(3) p by (rule instancia)
    have "u. Pl(u)  Co(p,u)"
    proof -
```

```
        have "Pl(s)" using assms(12) s by (rule instancia)
        hence "Pl(s)  Co(p,s)" using ‘Co(p,s)‘ ..
        thus "u. Pl(u)  Co(p,u)" ..
      qed
      show "Co(z,p)" using pz ‘A(p)‘ ‘M(p,z)‘ ‘u. Pl(u)  Co(p,u)‘
        by (rule mp3)
    qed
  qed
  hence "Co(z,p)  Co(p,s)" using 1 ..
  show "x y. A(x)  A(y)  (u. S(u)  Co(y,u)  Co(x,y))"
  proof -
    have "A(z)" using assms(2) z by (rule instancia)
    have "A(p)" using assms(3) p by (rule instancia)
    have "S(s)  Co(p,s)  Co(z,p)" using s ‘Co(p,s)‘ ‘Co(z,p)‘
      by (rule conjI3)
    hence "u. S(u)  Co(p,u)  Co(z,p)" ..
    have "A(z)  A(p)  (u. S(u)  Co(p,u)  Co(z,p))"
      using ‘A(z)‘ ‘A(p)‘ ‘u. S(u)  Co(p,u)  Co(z,p)‘
      by (rule conjI3)
    hence "y. A(z)  A(y)  (u. S(u)  Co(y,u)  Co(z,y))" ..
    thus "x y. A(x)  A(y)  (u. S(u)  Co(y,u)  Co(x,y))" ..
  qed
qed

end
```

4.4. Ejercicios: Argumentación lógica de primer orden con igualdad

```
chapter {* T4R3: Argumentación en lógica de primer orden con igualdad *}

theory T4R3
imports Main
begin

text {*
  ---------------------------------------------------------------------
  El objetivo de esta relación es formalizar y decidir la corrección
  de los argumentos. En el caso de que sea correcto, demostrarlo usando
```

```
sólo las reglas básicas de deducción natural de la lógica de primer
orden (sin usar el método auto). En el caso de que sea incorrecto,
calcular un contraejemplo con QuickCheck.

Las reglas básicas de la deducción natural son las siguientes:
ů conjI:      P; Q ⊢ P ∧ Q
ů conjunct1:  P ∧ Q ⊢ P
ů conjunct2:  P ∧ Q ⊢ Q
ů notnotD:    ¬¬ P ⊢ P
ů notnotI:    P ⊢ ¬¬ P
ů mp:         P → Q; P ⊢ Q
ů mt:         F → G; ¬G ⊢ ¬F
ů impI:       (P ⊢ Q) ⊢ P → Q
ů disjI1:     P ⊢ P ∨ Q
ů disjI2:     Q ⊢ P ∨ Q
ů disjE:      P ∨ Q; P → R; Q → R ⊢ R
ů FalseE:     False ⊢ P
ů notE:       ¬P; P ⊢ R
ů notI:       (P ⊢ False) ⊢ ¬P
ů iffI:       P → Q; Q → P ⊢ P = Q
ů iffD1:      Q = P; Q ⊢ P
ů iffD2:      P = Q; Q ⊢ P
ů ccontr:     (¬P ⊢ False) ⊢ P
ů excluded_middle: ¬P ∨ P

ů allI:       x. P x; P x → R ⊢ R
ů allE:       (x. P x) ⊢ x. P x
ů exI:        P x ⊢ x. P x
ů exE:        x. P x; x. P x → Q ⊢ Q

ů refl:       t = t
ů subst:      s = t; P s ⊢ P t

ů trans:      r = s; s = t ⊢ r = t
ů sym:        s = t ⊢ t = s
ů not_sym:    t ≠ s ⊢ s ≠ t
ů ssubst:     t = s; P s ⊢ P t
ů box_equals: a = b; a = c; b = d ⊢ a: = d
ů arg_cong:   x = y ⊢ f x = f y
ů fun_cong:   f = g ⊢ f x = g x
```

```
  ů cong:       f = g; x = y  ⟹ f x = g y
  ------------------------------------------------------------------
*}

text {*
  Se usarán, además, siquientes reglas que demostramos a continuación.
  *}

text {* ------------------------------------------------------------------
  Ejercicio 1. Formalizar, y decidir la corrección, del siguiente
  argumento
     Rosa ama a Curro. Paco no simpatiza con Ana. Quien no simpatiza con
     Ana ama a Rosa. Si una persona ama a otra, la segunda ama a la
     primera. Hay como máximo una persona que ama a Rosa. Por tanto,
     Paco es Curro.
  Usar A(x,y) para x ama a y
       S(x,y) para x simpatiza con y
       a      para Ana
       c      para Curro
       p      para Paco
       r      para Rosa
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma ejercicio_1a:
  assumes "A(r,c)"
          "ňS(p,a)"
          "x. ňS(x,a) ⟹ A(x,r)"
          "x y. A(x,y) ⟹ A(y,x)"
          "x y. A(x,r) ⟹ A(y,r) ⟹ x=y"
  shows   "p = c"
using assms
by auto

-- "La demostración estructurada es"
lemma ejercicio_1b:
  assumes 1: "A(r,c)" and
          2: "ňS(p,a)" and
          3: "x. ňS(x,a) ⟹ A(x,r)" and
          4: "x y. A(x,y) ⟹ A(y,x)" and
```

```
          5: "x y. A(x,r)  A(y,r)  x=y"
  shows   "p = c"
proof -
  have " y. A(p,r)  A(y,r)  p=y" using 5 ..
  hence "A(p,r)  A(c,r)  p=c" ..
  moreover
  have "A(p,r)  A(c,r)"
  proof
    have "ňS(p,a)  A(p,r)" using 3 ..
    thus "A(p,r)" using 2 ..
  next
    have "y. A(r,y)  A(y,r)" using 4 ..
    hence "A(r,c)  A(c,r)" ..
    thus "A(c,r)" using 1 ..
  qed
  ultimately show "p=c" ..
qed

-- "La demostración detallada es"
lemma ejercicio_1c:
  assumes 1: "A(r,c)" and
          2: "ňS(p,a)" and
          3: "x. ňS(x,a)  A(x,r)" and
          4: "x y. A(x,y)  A(y,x)" and
          5: "x y. A(x,r)  A(y,r)  x=y"
  shows   "p = c"
proof -
  have " y. A(p,r)  A(y,r)  p=y" using 5 by (rule allE)
  hence "A(p,r)  A(c,r)  p=c" by (rule allE)
  moreover
  have "A(p,r)  A(c,r)"
  proof (rule conjI)
    have "ňS(p,a)  A(p,r)" using 3 by (rule allE)
    thus "A(p,r)" using 2 by (rule mp)
  next
    have "y. A(r,y)  A(y,r)" using 4 by (rule allE)
    hence "A(r,c)  A(c,r)" by (rule allE)
    thus "A(c,r)" using 1 by (rule mp)
  qed
  ultimately show "p=c" by (rule mp)
```

```
qed

text {* ---------------------------------------------------------------
  Ejercicio 2. Formalizar, y decidir la corrección, del siguiente
  argumento
     Sólo hay un sofista que enseña gratuitamente, y éste es
     Sócrates. Sócrates argumenta mejor que ningún otro sofista. Platón
     argumenta mejor que algún sofista que enseña gratuitamente. Si una
     persona argumenta mejor que otra segunda, entonces la segunda no
     argumenta mejor que la primera. Por consiguiente, Platón no es un
     sofista.
  Usar G(x)   para x enseña gratuitamente
       M(x,y) para x argumenta mejor que y
       S(x)   para x es un sofista
       p      para Platón
       s      para Sócrates
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_2a:
  assumes 1: "∀y. (∃x. S(x) ∧ G(x) ∧ x=y) → y=s" and
          2: "∃x. S(x) ∧ x ≠ s ∧ M(s,x)" and
          3: "∀x. S(x) → G(x) → M(p,x)" and
          4: "∀x y. M(x,y) → ¬M(y,x)"
  shows "¬S(p)"
using assms
by metis

-- "La demostración semiautomática es"
lemma ejercicio_2b:
  assumes 1: "∀y. (∃x. S(x) ∧ G(x) ∧ x = y) → y = s" and
          2: "∃x. S(x) ∧ x ≠ s ∧ M(s,x)" and
          3: "∀x. S(x) → G(x) → M(p,x)" and
          4: "∀x y. M(x,y) → ¬M(y,x)"
  shows "¬S(p)"
proof
  assume "S(p)"
  obtain a where a: "(∃x. S(x) ∧ G(x) ∧ x = a) → a = s" using 1 ..
  hence s: "∃x. S(x) ∧ G(x) ∧ x = s" by metis
  obtain b where b: "S(b) ∧ G(b) ∧ M(p,b)" using 3 ..
```

```
  hence "b = s" using s by metis
  hence "M(p,s)" using b by metis
  hence "¬M(s,p)" using 4 by metis
  hence "p = s" using 2 ‘S(p)‘ by metis
  hence "M(s,s)" using ‘M(p,s)‘ by auto
  have "¬M(s,s)" using ‘p = s‘ ‘¬M(s,p)‘ by auto
  thus False using ‘M(s,s)‘ by auto
qed

-- "La demostración estructurada es"
lemma ejercicio_2c:
  assumes 1: "∃y. (∀x. S(x) → G(x) → x = y) ∧ y = s" and
          2: "∀x. S(x) ∧ x ≠ s → M(s,x)" and
          3: "∀x. S(x) → G(x) → M(p,x)" and
          4: "∀x y. M(x,y) → ¬M(y,x)"
  shows "¬S(p)"
proof
  assume "S(p)"
  obtain a where a: "(∀x. S(x) → G(x) → x = a) ∧ a = s" using 1 ..
  hence s: "∀x. S(x) → G(x) → x = s"
  proof
    have "a = s" using a ..
    have "∀x. S(x) → G(x) → x = a" using a ..
    with ‘a = s‘ show "∀x. S(x) → G(x) → x = s" by (rule subst)
  qed
  obtain b where b: "S(b) → G(b) → M(p,b)" using 3 ..
  hence "b = s"
  proof
    have "S(b) → G(b)"
    proof
      show "S(b)" using b ..
    next
      have "G(b) → M(p,b)" using b ..
      thus "G(b)" ..
    qed
    have "S(b) → G(b) → b = s" using s ..
    thus "b = s" using ‘S(b) → G(b)‘ ..
  qed
  have "M(p,s)"
  proof -
```

```
    have "G(b)  M(p,b)" using b ..
    hence "M(p,b)" ..
    with ‘b = s‘ show "M(p,s)" by (rule subst)
  qed
  have "ňM(s,p)"
  proof -
    have "y. M(p,y)  ňM(y,p)" using 4 ..
    hence "M(p,s)  ňM(s,p)" ..
    thus "ňM(s,p)" using ‘M(p,s)‘ ..
  qed
  have "p = s"
  proof (rule ccontr)
    assume "p  s"
    with ‘S(p)‘ have "S(p)  p  s" ..
    have "S(p)  p  s  M(s,p)" using 2 ..
    hence "M(s,p)" using ‘S(p)  p  s‘ ..
    with ‘ňM(s,p)‘ show False ..
  qed
  hence "M(s,s)" using ‘M(p,s)‘ by (rule subst)
  have "ňM(s,s)" using ‘p = s‘ ‘ňM(s,p)‘ by (rule subst)
  thus False using ‘M(s,s)‘ ..
qed

-- "La demostración detallada es"
lemma ejercicio_2d:
  assumes 1: "y. (x. S(x)  G(x)  x = y)  y = s" and
          2: "x. S(x)  x  s  M(s,x)" and
          3: "x. S(x)  G(x)  M(p,x)"  and
          4: "x y. M(x,y)  ňM(y,x)"
  shows "ňS(p)"
proof
  assume "S(p)"
  obtain a where a: "(x. S(x)  G(x)  x = a)  a = s" using 1 by (rule exE)
  have s: "x. S(x)  G(x)  x = s"
  proof -
    have "a = s" using a ..
    have "x. S(x)  G(x)  x = a" using a ..
    with ‘a = s‘ show "x. S(x)  G(x)  x = s"  by (rule subst)
  qed
  obtain b where b: "S(b)  G(b)  M(p,b)" using 3 ..
```

```
  have "b = s"
  proof -
    have "S(b)  G(b)"
    proof (rule conjI)
      show "S(b)" using b by (rule conjunct1)
    next
      have "G(b)  M(p,b)" using b by (rule conjunct2)
      thus "G(b)" by (rule conjunct1)
    qed
    have "S(b)  G(b)  b = s" using s by (rule allE)
    thus "b = s" using `S(b)  G(b)` by (rule iffD1)
  qed
  have "M(p,s)"
  proof -
    have "G(b)  M(p,b)" using b ..
    hence "M(p,b)" ..
    with `b = s` show "M(p,s)" by (rule subst)
  qed
  have "ňM(s,p)"
  proof -
    have "y. M(p,y)  ňM(y,p)" using 4 ..
    hence "M(p,s)  ňM(s,p)" ..
    thus "ňM(s,p)" using `M(p,s)` ..
  qed
  have "p = s"
  proof (rule ccontr)
    assume "p  s"
    with `S(p)` have "S(p)  p  s" ..
    have "S(p)  p  s  M(s,p)" using 2 ..
    hence "M(s,p)" using `S(p)  p  s` ..
    with `ňM(s,p)` show False ..
  qed
  hence "M(s,s)" using `M(p,s)` by (rule subst)
  have "ňM(s,s)" using `p = s` `ňM(s,p)` by (rule subst)
  thus False using `M(s,s)` by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 3. Formalizar, y decidir la corrección, del siguiente
  argumento
```

```
    Todos los filósofos se han preguntado qué es la filosofía. Los que
    se preguntan qué es la filosofía se vuelven locos. Nietzsche es
    filósofo. El maestro de Nietzsche no acabó loco. Por tanto,
    Nietzsche y su maestro son diferentes personas.
  Usar F(x) para x es filósofo
       L(x) para x se vuelve loco
       P(x) para x se ha preguntado qué es la filosofía.
       m    para el maestro de Nietzsche
       n    para Nietzsche
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma ejercicio_3a:
  assumes "∀x. F(x) → P(x)"
          "∀x. P(x) → L(x)"
          "F(n)"
          "¬L(m)"
  shows   "n ≠ m"
using assms
by auto

-- "En las siguientes demostraciones se usará este lema"
lemma para_todo_implica:
  assumes "∀x. P(x) → Q(x)"
          "P(a)"
  shows   "Q(a)"
proof -
  have "P(a) → Q(a)" using assms(1) ..
  thus "Q(a)" using assms(2) ..
qed

-- "La demostración estructurada es"
lemma ejercicio_3b:
  assumes "∀x. F(x) → P(x)"
          "∀x. P(x) → L(x)"
          "F(n)"
          "¬L(m)"
  shows   "n ≠ m"
proof
  assume "n = m"
```

```
  hence "F(m)" using assms(3) by (rule subst)
  with assms(1) have "P(m)" by (rule para_todo_implica)
  with assms(2) have "L(m)" by (rule para_todo_implica)
  with assms(4) show False ..
qed

-- "La demostración detallada es"
lemma ejercicio_3c:
  assumes "x. F(x)  P(x)"
          "x. P(x)  L(x)"
          "F(n)"
          "ňL(m)"
  shows   "n  m"
proof (rule notI)
  assume "n = m"
  hence "F(m)" using assms(3) by (rule subst)
  with assms(1) have "P(m)" by (rule para_todo_implica)
  with assms(2) have "L(m)" by (rule para_todo_implica)
  with assms(4) show False by (rule notE)
qed

text {* ---------------------------------------------------------------
  Ejercicio 4. Formalizar, y decidir la corrección, del siguiente
  argumento
     Los padres son mayores que los hijos. Juan es el padre de Luis. Por
     tanto, Juan es mayor que Luis.
  Usar M(x,y) para x es mayor que y
       p(x)   para el padre de x
       j      para Juan
       l      para Luis
  --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_4a:
  assumes "x. M(p(x),x)"
          "j = p(l)"
shows     "M(j,l)"
using assms
by auto
```

```
-- "La demostración estructurada es"
lemma ejercicio_4b:
  assumes "∀x. M(p(x),x)"
          "j = p(l)"
shows     "M(j,l)"
proof -
  have "M(p(l),l)" using assms(1) ..
  with assms(2) show "M(j,l)" by (rule ssubst)
qed

-- "La demostración detallada es"
lemma ejercicio_4c:
  assumes "∀x. M(p(x),x)"
          "j = p(l)"
shows     "M(j,l)"
proof -
  have "M(p(l),l)" using assms(1) by (rule allE)
  with assms(2) show "M(j,l)" by (rule ssubst)
qed

text {* ----------------------------------------------------------------
  Ejercicio 5. Formalizar, y decidir la corrección, del siguiente
  argumento
     El esposo de la hermana de Toni es Roberto. La hermana de Toni es
     María. Por tanto, el esposo de María es Roberto.
  Usar e(x) para el esposo de x
       h    para la hermana de Toni
       m    para María
       r    para Roberto
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_5a:
  assumes "e(h) = r"
          "h = m"
  shows   "e(m) = r"
using assms
by auto

-- "La demostración detallada es"
```

```
lemma ejercicio_5b:
  assumes "e(h) = r"
          "h = m"
  shows   "e(m) = r"
using assms(2,1)
by (rule subst)

text {* ---------------------------------------------------------------
  Ejercicio 6. Formalizar, y decidir la corrección, del siguiente
  argumento
     Luis y Jaime tienen el mismo padre. La madre de Rosa es
     Eva. Eva ama a Carlos. Carlos es el padre de Jaime. Por tanto,
     la madre de Rosa ama al padre de Luis.
  Usar A(x,y) para x ama a y
       m(x)   para la madre de x
       p(x)   para el padre de x
       c      para Carlos
       e      para Eva
       j      para Jaime
       l      para Luis
       r      para Rosa
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_6a:
  assumes "p(l) = p(j)"
          "m(r) = e"
          "A(e,c)"
          "c = p(j)"
  shows   "A(m(r),p(l))"
using assms
by auto

-- "La demostración detallada es"
lemma ejercicio_6b:
  assumes "p(l) = p(j)"
          "m(r) = e"
          "A(e,c)"
          "c = p(j)"
  shows   "A(m(r),p(l))"
```

```
proof -
  have "A(m(r),c)" using assms(2,3) by (rule ssubst)
  with assms(4) have "A(m(r),p(j))" by (rule subst)
  with assms(1) show "A(m(r),p(l))" by (rule ssubst)
qed

text {* ---------------------------------------------------------------
  Ejercicio 7. Formalizar, y decidir la corrección, del siguiente
  argumento
     Si dos personas son hermanos, entonces tienen la misma madre y el
     mismo padre. Juan es hermano de Luis. Por tanto, la madre del padre
     de Juan es la madre del padre de Luis.
  Usar H(x,y) para x es hermano de y
       m(x)   para la madre de x
       p(x)   para el padre de x
       j      para Juan
       l      para Luis
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_7a:
  assumes "x y. H(x,y)  m(x) = m(y)  p(x) = p(y)"
          "H(j,l)"
  shows   "m(p(j)) = m(p(l))"
using assms
by auto

-- "La demostración estructurada es"
lemma ejercicio_7b:
  assumes "x y. H(x,y)  m(x) = m(y)  p(x) = p(y)"
          "H(j,l)"
  shows   "m(p(j)) = m(p(l))"
proof -
  have "y. H(j,y)  m(j) = m(y)  p(j) = p(y)" using assms(1) ..
  hence "H(j,l)  m(j) = m(l)  p(j) = p(l)" ..
  hence "m(j) = m(l)  p(j) = p(l)" using assms(2) ..
  hence "p(j) = p(l)" ..
  have "m(p(j)) = m(p(j))" by (rule refl)
  with `p(j) = p(l)` show "m(p(j)) = m(p(l))" by (rule subst)
qed
```

```
-- "La demostración detallada es"
lemma ejercicio_7c:
  assumes "x y. H(x,y)  m(x) = m(y)  p(x) = p(y)"
          "H(j,l)"
  shows   "m(p(j)) = m(p(l))"
proof -
  have "y. H(j,y)  m(j) = m(y)  p(j) = p(y)"
    using assms(1)  by (rule allE)
  hence "H(j,l)  m(j) = m(l)  p(j) = p(l)" by (rule allE)
  hence "m(j) = m(l)  p(j) = p(l)" using assms(2) by (rule mp)
  hence "p(j) = p(l)" by (rule conjunct2)
  have "m(p(j)) = m(p(j))" by (rule refl)
  with `p(j) = p(l)` show "m(p(j)) = m(p(l))" by (rule subst)
qed

text {* ---------------------------------------------------------------
  Ejercicio 8. Formalizar, y decidir la corrección, del siguiente
  argumento
     Todos los miembros del claustro son asturianos. El secretario forma
     parte del claustro. El señor Martínez es el secretario. Por tanto,
     el señor Martínez es asturiano.
  Usar C(x) para x es miembro del claustro
       A(x) para x es asturiano
       s    para el secretario
       m    para el señor Martínez
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_8a:
  assumes "x. C(x)  A(x)"
          "C(s)"
          "m = s"
  shows   "A(m)"
using assms
by auto

-- "La demostración estructurada es"
lemma ejercicio_8b:
  assumes "x. C(x)  A(x)"
```

```
          "C(s)"
          "m = s"
  shows   "A(m)"
proof -
  have "C(s)  A(s)" using assms(1) ..
  hence "A(s)" using assms(2) ..
  with assms(3) show "A(m)" by (rule ssubst)
qed

-- "La demostración detallada es"
lemma ejercicio_8c:
  assumes "x. C(x)  A(x)"
          "C(s)"
          "m = s"
  shows   "A(m)"
proof -
  have "C(s)  A(s)" using assms(1) by (rule allE)
  hence "A(s)" using assms(2) by (rule mp)
  with assms(3) show "A(m)" by (rule ssubst)
qed

text {* ---------------------------------------------------------------
  Ejercicio 10. Formalizar, y decidir la corrección, del siguiente
  argumento
     Eduardo pudo haber visto al asesino. Antonio fue el primer testigo
     de la defensa. O Eduardo estaba en clase o Antonio dio falso
     testimonio. Nadie en clase pudo haber visto al asesino. Luego, el
     primer testigo de la defensa dio falso testimonio.
  Usar C(x) para x estaba en clase
       F(x) para x dio falso testimonio
       V(x) para x pudo haber visto al asesino
       a    para Antonio
       e    para Eduardo
       p    para el primer testigo de la defensa
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_10a:
  assumes "V(e)"
          "a = p"
```

```
          "C(e)  F(a)"
          "x. C(x)  ňV(x)"
   shows  "F(p)"
using assms
by auto

-- "La demostración estructurada es"
lemma ejercicio_10b:
  assumes "V(e)"
          "a = p"
          "C(e)  F(a)"
          "x. C(x)  ňV(x)"
   shows  "F(p)"
proof -
  have "C(e)  F(a)" using assms(3) .
  hence "F(a)"
  proof
    assume "C(e)"
    have "C(e)  ňV(e)" using assms(4) ..
    hence "ňV(e)" using ‘C(e)‘ ..
    thus "F(a)" using assms(1) ..
  next
    assume "F(a)"
    thus "F(a)" .
  qed
  with assms(2) show "F(p)" by (rule subst)
qed

-- "La demostración detallada es"
lemma ejercicio_10c:
  assumes "V(e)"
          "a = p"
          "C(e)  F(a)"
          "x. C(x)  ňV(x)"
   shows  "F(p)"
proof -
  have "C(e)  F(a)" using assms(3) by this
  hence "F(a)"
  proof (rule disjE)
    assume "C(e)"
```

```
    have "C(e)  ¬V(e)" using assms(4) by (rule allE)
    hence "¬V(e)" using `C(e)` by (rule mp)
    thus "F(a)" using assms(1) by (rule notE)
  next
    assume "F(a)"
    thus "F(a)" by this
  qed
  with assms(2) show "F(p)" by (rule subst)
qed

text {* ---------------------------------------------------------------
  Ejercicio 11. Formalizar, y decidir la corrección, del siguiente
  argumento
     La luna hoy es redonda. La luna de hace dos semanas tenía forma de
     cuarto creciente. Luna no hay más que una, es decir, siempre es la
     misma. Luego existe algo que es a la vez redondo y con forma de
     cuarto creciente.
  Usar L(x) para la luna del momento x
       R(x) para x es redonda
       C(x) para x tiene forma de cuarto creciente
       h    para hoy
       d    para hace dos semanas
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_11a:
  assumes "R(l(h))"
          "C(l(d))"
          "x y. l(x) = l(y)"
  shows   "x. R(x)  C(x)"
using assms
by metis

-- "La demostración estructurada es"
lemma ejercicio_11b:
  assumes "R(l(h))"
          "C(l(d))"
          "x y. l(x) = l(y)"
  shows   "x. R(x)  C(x)"
proof -
```

```
  have "R(l(h))  C(l(d))" using assms(1,2) ..
  have "y. l(d) = l(y)" using assms(3) ..
  hence "l(d) = l(h)" ..
  hence "R(l(h))  C(l(h))" using `R(l(h))  C(l(d))` by (rule subst)
  thus "x. R(x)  C(x)" ..
qed

-- "La demostración detallada es"
lemma ejercicio_11c:
  assumes "R(l(h))"
          "C(l(d))"
          "x y. l(x) = l(y)"
  shows   "x. R(x)  C(x)"
proof -
  have "R(l(h))  C(l(d))" using assms(1,2) by (rule conjI)
  have "y. l(d) = l(y)" using assms(3) by (rule allE)
  hence "l(d) = l(h)" by (rule allE)
  hence "R(l(h))  C(l(h))" using `R(l(h))  C(l(d))` by (rule subst)
  thus "x. R(x)  C(x)" by (rule exI)
qed

text {* ---------------------------------------------------------------
  Ejercicio 12. Formalizar, y decidir la corrección, del siguiente
  argumento
     Juana sólo tiene un marido. Juana está casada con Tomás. Tomás es
     delgado y Guillermo no. Luego, Juana no está casada con Guillermo.
  Usar D(x)   para x es delgado
       C(x,y) para x está casada con y
       g      para Guillermo
       j      para Juana
       t      para Tomás
   --------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_12a:
  assumes "x. y. C(j,y)  y = x"
          "C(j,t)"
          "D(t)  ňD(g)"
  shows   "ňC(j,g)"
using assms
```

```
by auto

-- "La demostración estructurada es"
lemma ejercicio_12b:
  assumes "∀x. ∀y. C(j,y) → y = x"
          "C(j,t)"
          "D(t) ∧ ¬D(g)"
  shows   "¬C(j,g)"
proof
  assume "C(j,g)"
  obtain a where a: "∀y. C(j,y) → y = a" using assms(1) ..
  hence "C(j,t) → t = a" ..
  hence "t = a" using assms(2) ..
  have "C(j,g) → g = a" using a ..
  hence "g = a" using ‹C(j,g)› ..
  hence "t = g" using ‹t = a› by (rule ssubst)
  hence "D(g) ∧ ¬D(g)" using assms(3) by (rule subst)
  hence "D(g)" ..
  have "¬D(g)" using ‹D(g) ∧ ¬D(g)› ..
  thus False using ‹D(g)› ..
qed

-- "La demostración detallada es"
lemma ejercicio_12c:
  assumes "∀x. ∀y. C(j,y) → y = x"
          "C(j,t)"
          "D(t) ∧ ¬D(g)"
  shows   "¬C(j,g)"
proof (rule notI)
  assume "C(j,g)"
  obtain a where a: "∀y. C(j,y) → y = a" using assms(1) by (rule exE)
  hence "C(j,t) → t = a" by (rule allE)
  hence "t = a" using assms(2) by (rule iffD1)
  have "C(j,g) → g = a" using a by (rule allE)
  hence "g = a" using ‹C(j,g)› by (rule iffD1)
  hence "t = g" using ‹t = a› by (rule ssubst)
  hence "D(g) ∧ ¬D(g)" using assms(3) by (rule subst)
  hence "D(g)" by (rule conjunct1)
  have "¬D(g)" using ‹D(g) ∧ ¬D(g)› by (rule conjunct2)
  thus False using ‹D(g)› by (rule notE)
```

```
qed

text {* ---------------------------------------------------------------
  Ejercicio 13. Formalizar, y decidir la corrección, del siguiente
  argumento
     Sultán no es Chitón. Sultán no obtendrá un plátano a menos que
     pueda resolver cualquier problema. Si el chimpancé Chitón trabaja
     más que Sultán resolverá problemas que Sultán no puede resolver.
     Todos los chimpancés distintos de Sultán trabajan más que Sultán.
     Por consiguiente, Sultán no obtendrá un plátano.
  Usar Pl(x)  para x obtiene el plátano
       Pr(x)  para x es un problema
       R(x,y) para x resuelve y
       T(x,y) para x trabaja más que y
       c      para Chitón
       s      para Sultán
   ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ejercicio_13a:
  assumes "s  c"
          "Pl(s)  (x. Pr(x)  R(s,x))"
          "T(c,s)  (x. Pr(x)  R(c,x)  ňR(s,x))"
          "x. x  s  T(x,s)"
  shows   "ňPl(s)"
using assms
by metis

-- "La demostración estructurada es"
lemma ejercicio_13b:
  assumes "s  c"
          "Pl(s)  (x. Pr(x)  R(s,x))"
          "T(c,s)  (x. Pr(x)  R(c,x)  ňR(s,x))"
          "x. x  s  T(x,s)"
  shows   "ňPl(s)"
proof
  assume "Pl(s)"
  with assms(2) have "x. Pr(x)  R(s,x)" ..
  have "c  s" using assms(1) by (rule not_sym)
  have "c  s  T(c,s)" using assms(4) ..
```

```
  hence "T(c,s)" using ‘c  s‘ ..
  with assms(3) have "x. Pr(x)  R(c,x)  ňR(s,x)" ..
  then obtain a where a: "Pr(a)  R(c,a)  ňR(s,a)" ..
  have "R(s,a)"
  proof -
    have "Pr(a)" using a ..
    have "Pr(a)  R(s,a)" using ‘x. Pr(x)  R(s,x)‘ ..
    thus "R(s,a)" using ‘Pr(a)‘ ..
  qed
  have "ňR(s,a)"
  proof -
    have "R(c,a)  ňR(s,a)" using a ..
    thus "ňR(s,a)" ..
  qed
  thus False using ‘R(s,a)‘ ..
qed

-- "La demostración detallada es"
lemma ejercicio_13c:
  assumes "s  c"
          "Pl(s)  (x. Pr(x)  R(s,x))"
          "T(c,s)  (x. Pr(x)  R(c,x)  ňR(s,x))"
          "x. x  s  T(x,s)"
  shows   "ňPl(s)"
proof (rule notI)
  assume "Pl(s)"
  with assms(2) have "x. Pr(x)  R(s,x)" by (rule mp)
  have "c  s" using assms(1) by (rule not_sym)
  have "c  s  T(c,s)" using assms(4) by (rule allE)
  hence "T(c,s)" using ‘c  s‘ by (rule mp)
  with assms(3) have "x. Pr(x)  R(c,x)  ňR(s,x)" by (rule mp)
  then obtain a where a: "Pr(a)  R(c,a)  ňR(s,a)" by (rule exE)
  have "R(s,a)"
  proof -
    have "Pr(a)" using a by (rule conjunct1)
    have "Pr(a)  R(s,a)" using ‘x. Pr(x)  R(s,x)‘ by (rule allE)
    thus "R(s,a)" using ‘Pr(a)‘ by (rule mp)
  qed
  have "ňR(s,a)"
  proof -
```

```
    have "R(c,a)  ňR(s,a)" using a by (rule conjunct2)
    thus "ňR(s,a)" by (rule conjunct2)
  qed
  thus False using ‘R(s,a)‘ by (rule notE)
qed

end
```

Capítulo 5

Resumen de Isabelle/Isar y de la lógica

```
chapter {* Tema 5: Resumen del lenguaje Isabelle/Isar y las reglas de la lógica *}

theory T5
imports Main
begin

section {* Sintaxis (simplificada) de Isabelle/Isar *}

text {*
  Representación de lemas (y teoremas)
  ů Un lema (o teorema) comienza con una etiqueta seguida por algunas
    premisas y una conclusión.
  ů Las premisas se introducen con la palabra "assumes" y se separan
    con "and".
  ů Cada premisa puede etiquetarse para referenciarse en la demostración.
  ů La conclusión se introduce con la palabra "shows".

  Gramática (simplificada) de las demostraciones en Isabelle/Isar
  <demostración> ::= proof <método> <declaración>* qed
                   | by <método>
  <declaración>  ::= fix <variable>+
                   | assume <proposición>+
                   | (from <hecho>+)? have <proposición>+ <demostración>
                   | (from <hecho>+)? show <proposición>+ <demostración>
  <proposición>  ::= (<etiqueta>:)? <cadena>
  hecho          ::= <etiqueta>
  método         ::= -
                   | this
```

```
                    | rule <hecho>
                    | simp
                    | blast
                    | auto
                    | induct <variable>

  La declaración "show" demuestra la conclusión de la demostración
  mientras que la declaración "have" demuestra un resultado intermedio.
*}

section {* Atajos de Isabelle/Isar *}

text {*
  Isar tiene muchos atajos, como los siguientes:
  this        | éste           | el hecho probado en la declaración anterior
  then        | entonces       | from this
  hence       | por lo tanto   | then have
  thus        | de esta manera | then show
  with hecho+ | con            | from hecho+ and this
  .           | por ésto       | by this
  ..          | trivialmente   | by regla (Isabelle adivina la regla)
*}

section {* Resumen de las reglas de la lógica *}

text {*
  Resumen de reglas proposicionales:
  ů TrueI:          True
  ů conjI:          P; Q  P  Q
  ů conjunct1:      P  Q  P
  ů conjunct2:      P  Q  Q
  ů conjE:          P  Q; P; Q  R  R
  ů disjI1:         P  P  Q
  ů disjI2:         Q  P  Q
  ů disjE:          P  Q; P  R; Q  R  R
  ů notI:           (P  False)  ňP
  ů notE:           ňP; P  R
  ů FalseE:         False  P
  ů notnotD:        ňň P  P
  ů impI:           (P  Q)  P  Q
```

```
  ů impE:            P ⟶ Q; P; Q ⟶ R ⟹ R
  ů mp:              P ⟶ Q; P ⟹ Q
  ů iff:             (P ⟶ Q) ⟶ (Q ⟶ P) ⟶ P = Q
  ů iffI:            P ⟶ Q; Q ⟶ P ⟹ P = Q
  ů iffD1:           Q = P; Q ⟹ P
  ů iffD2:           P = Q; Q ⟹ P
  ů iffE:            P = Q; P ⟶ Q; Q ⟶ P ⟹ R ⟹ R
  ů ccontr:          (ňP ⟹ False) ⟹ P
  ů classical:       (ňP ⟹ P) ⟹ P
  ů exluded_middle: ňP ∨ P
  ů disjCI:          (ňQ ⟹ P) ⟹ P ∨ Q
  ů impCE:           P ⟶ Q; ňP ⟹ R; Q ⟹ R ⟹ R
  ů iffCE:           P = Q; P; Q ⟹ R; ňP; ňQ ⟹ R ⟹ R
  ů swap:            ňP; ňR ⟹ P ∨ R
*}

text {*
  Resumen de reglas de cuantificadores:
  ů allE:    ∀x. P x; P y ⟹ R ⟹ R
  ů allI:    (∀x. P x) ⟹ ∀x. P x
  ů exI:     P a ⟹ ∃x. P x
  ů exE:     ∃x. P x; ∀x. P x ⟶ Q ⟹ Q
*}


text {*
  Resumen de reglas de la igualdad:
  ů refl:        t = t
  ů subst:       s = t; P s ⟹ P t
  ů trans:       r = s; s = t ⟹ r = t
  ů sym:         s = t ⟹ t = s
  ů not_sym:     t ≠ s ⟹ s ≠ t
  ů ssubst:      t = s; P s ⟹ P t
  ů box_equals: a = b; a = c; b = d ⟹ c = d
  ů arg_cong:    x = y ⟹ f x = f y
  ů fun_cong:    f = g ⟹ f x = g x
  ů cong:        f = g; x = y ⟹ f x = g y
*}

text {*
```

```
  Nota: Más información sobre las reglas de inferencia se encuentra en
  la sección  2.2 de "Isabelle's Logics: HOL" http://goo.gl/ZwdUu
  (página 8).
*}

end
```

Capítulo 6

Razonamiento por casos y por inducción

```
chapter {* Tema 6: Razonamiento por casos y por inducción *}

theory T6
imports Main Parity
begin

section {* Razonamiento por distinción de casos *}

subsection {* Distinción de casos booleanos *}

text {*
  Lema. [Demostración por distinción de casos booleanos]
     ňA  A
*}

-- "La demostración estructurada es"
lemma "ňA  A"
proof cases
  assume "A"
  thus ?thesis ..
next
  assume "ňA"
  thus ?thesis ..
qed

-- "La demostración detallada es"
lemma "ňA  A"
proof cases
```

```
  assume "A"
  thus ?thesis by (rule disjI2)
next
  assume "¬A"
  thus ?thesis by (rule disjI1)
qed

-- "La demostración automática es"
lemma "¬A ∨ A"
by auto

text {*
  Lema. [Demostración por distinción de casos booleanos nominados]
     ¬A ∨ A
*}

-- "La demostración estructurada es"
lemma "¬A ∨ A"
proof (cases "A")
  case True
  thus ?thesis ..
next
  case False
  thus ?thesis ..
qed

-- "La demostración detallada es"
lemma "¬A ∨ A"
proof (cases "A")
  case True
  thus ?thesis by (rule disjI2)
next
  case False
  thus ?thesis by (rule disjI1)
qed

text {*
  El método "cases" sobre una fórmula:
  ∘ El método (cases F) es una abreviatura de la aplicación de la regla
       F ⟹ Q; ¬F ⟹ Q ⟹ Q
```

```
  ů La expresión "case True" es una abreviatura de F.
  ů La expresión "case False" es una abreviatura de ňF.
  ů Ventajas de "cases" con nombre:
    ů reduce la escritura de la fórmula y
    ů es independiente del orden de los casos.
*}

subsection {* Distinción de casos sobre otros tipos de datos *}

text {*
  Lema. [Distinción de casos sobre listas]
  La longitud del resto de una lista es la longitud de la lista menos 1.
*}

-- "La demostración detallada es"
lemma "length(tl xs) = length xs - 1"
proof (cases xs)
  case Nil thus ?thesis by simp
next
  case Cons thus ?thesis by simp
qed

-- "La demostración automática es"
lemma "length(tl xs) = length xs - 1"
by auto

text {*
  Distinción de casos sobre listas:
  ů El método de distinción de casos se activa con (cases xs) donde xs
    es del tipo lista.
  ů "case Nil" es una abreviatura de
       "assume Nil: xs =[]".
  ů "case Cons" es una abreviatura de
       "fix ? ?? assume Cons: xs = ? # ??"
    donde ? y ?? son variables anónimas.
*}

text {*
  Lema. [Ejemplo de análisis de casos]
  El resultado de eliminar los n+1 primeros elementos de xs es el mismo
```

```
  que eliminar los n primeros elementos del resto de xs.
*}

-- "La demostración detallada es"
lemma "drop (n + 1) xs = drop n (tl xs)"
proof (cases xs)
  case Nil thus "drop (n + 1) xs = drop n (tl xs)" by simp
next
  case Cons thus "drop (n + 1) xs = drop n (tl xs)" by simp
qed

-- "La demostración automática es"
lemma "drop (n + 1) xs = drop n (tl xs)"
by (cases xs, auto)

text {*
  La función drop está definida en la teoría List de forma que
  (drop n xs) la lista obtenida eliminando en xs} los n primeros
  elementos. Su definición es la siguiente
     drop_Nil:  "drop n [] = []" |
     drop_Cons: "drop n (x#xs) = (case n of
                                    0 => x#xs |
                                    Suc(m) => drop m xs)"
*}

section {* Inducción matemática *}

text {*
  [Principio de inducción matemática]
  Para demostrar una propiedad P para todos los números naturales basta
  probar que el 0 tiene la propiedad P y que si n tiene la propiedad P,
  entonces n+1 también la tiene.
     P 0; n. P n  P (Suc n)  P m

  En Isabelle el principio de inducción matemática está formalizado en
  el teorema nat.induct y puede verse con
     thm nat.induct

  Ejemplo de demostración por inducción: Usaremos el principio de
  inducción matemática para demostrar que
```

```
     1 + 3 + ... + (2n-1) = n^2

  Definición. [Suma de los primeros impares]
  (suma_impares n) la suma de los n números impares.
*}

fun suma_impares :: "nat  nat" where
  "suma_impares 0 = 0"
| "suma_impares (Suc n) = (2*(Suc n) - 1) + suma_impares n"

text {*
  La suma de los 3 primero número impares se puede calcular mediante "value".
*}

value "suma_impares 3"

text {*
  Lema. [Ejemplo de demostración por inducción matemática]
  La suma de los n primeros números impares es n^2.
*}

-- "La demostración automática es"
lemma "suma_impares n = n * n"
by (induct n) simp_all

text {*
  En la demostración "by (induct n) simp_all" se aplica inducción en n y
  los dos casos se prueban por simplificación.
*}

-- "  Demostración del lema anterior usando patrones"
lemma "suma_impares n = n * n" (is "?P n")
proof (induct n)
  show "?P 0" by simp
next
  fix n assume "?P n"
  thus "?P (Suc n)" by simp
qed

text {*
```

```
  Patrones: Cualquier fórmula seguida de (is patrón) equipara el patrón
  con la fórmula.
*}

-- "Demostración del lema anterior con patrones y razonamiento ecuacional"
lemma "suma_impares n = n * n" (is "?P n")
proof (induct n)
  show "?P 0" by simp
next
  fix n assume HI: "?P n"
  have "suma_impares (Suc n) = (2 * (Suc n) - 1) + suma_impares n" by simp
  also have " = (2 * (Suc n) - 1) + n * n" using HI by simp
  also have " = n * n + 2 * n + 1" by simp
  finally show "?P (Suc n)" by simp
qed

-- "Demostración del lema anterior por inducción y razonamiento ecuacional"
lemma "suma_impares n = n * n"
proof (induct n)
  show "suma_impares 0 = 0 * 0" by simp
next
  fix n assume HI: "suma_impares n = n * n"
  have "suma_impares (Suc n) = (2 * (Suc n) - 1) + suma_impares n" by simp
  also have " = (2 * (Suc n) - 1) + n * n" using HI by simp
  also have " = n * n + 2 * n + 1" by simp
  finally show "suma_impares (Suc n) = (Suc n) * (Suc n)" by simp
qed

text {*
  Definición. Un número natural n es par si existe un natural m tal que
  n=m+m.
*}

definition par ::  "nat  bool" where
  "par n  m. n=m+m"

text {*
  Lema. [Ejemplo de inducción y existenciales] Para todo número natural
  n, se verifica que n*(n+1) par.
*}
```

```
lemma
  fixes n :: "nat"
  shows "par (n*(n+1))"
proof (induct n)
  show "par (0*(0+1))" by (simp add:par_def)
next
  fix n assume "par (n*(n+1))"
  hence "m. n*(n+1) = m+m" by (simp add:par_def)
  then obtain m where m: "n*(n+1) = m+m" by (rule exE)
  hence "(Suc n)*((Suc n)+1) = (m+n+1)+(m+n+1)" by auto
  hence "m. (Suc n)*((Suc n)+1) = m+m" by (rule exI)
  thus "par ((Suc n)*((Suc n)+1))" by (simp add:par_def)
qed

text {*
  En Isabelle puede demostrarse de manera más simple un lema equivalente
  usando en lugar de la función "par" la función "even" definida en la
  teoría Parity por
     even x  x mod 2 = 0"
*}

lemma
  fixes n :: "nat"
  shows "even (n*(n+1))"
by auto

text {*
  Para completar la demostración basta demostrar la equivalencia de las
  funciones "par" y "even".
*}

lemma
  fixes n :: "nat"
  shows "par n = even n"
proof -
  have "par n = (m. n = m+m)" by (simp add:par_def)
  thus "par n = even n" by presburger
qed
```

```
text {*
  En la demostración anterior hemos usado la táctica "presburger" que
  corresponde a la aritmética de Presburger.
*}

section {* Inducción estructural *}

text {*
  Inducción estructural]
  ů En Isabelle puede hacerse inducción estructural sobre cualquier tipo
    recursivo.
  ů La inducción matemática es la inducción estructural sobre el tipo de
    los naturales.
  ů El esquema de inducción estructural sobre listas es
    ů list.induct: P []; x ys. P ys  P (x # ys)  P zs
  ů Para demostrar una propiedad para todas las listas basta demostrar
    que la lista vacía tiene la propiedad y que al añadir un elemento a una
    lista que tiene la propiedad se obtiene una lista que también tiene la
    propiedad.
  ů En Isabelle el principio de inducción sobre listas está formalizado
    mediante el teorema list.induct que puede verse con
       thm list.induct

  Concatenación de listas:
  En la teoría List.thy está definida la concatenación de listas (que
  se representa por @) como sigue
     append_Nil:  "[]@ys = ys"
     append_Cons: "(x#xs)@ys = x#(xs@ys)"

  Lema. [Ejemplo de inducción sobre listas]
  La concatenación de listas es asociativa.
*}

-- "La demostración automática es"
lemma conc_asociativa_1: "xs @ (ys @ zs) = (xs @ ys) @ zs"
by (induct xs) simp_all

-- "La demostración estructurada es"
lemma conc_asociativa: "xs @ (ys @ zs) = (xs @ ys) @ zs"
proof (induct xs)
```

```
  show "[] @ (ys @ zs) = ([] @ ys) @ zs"
  proof -
    have "[] @ (ys @ zs) = ys @ zs" by simp
    also have " = ([] @ ys) @ zs" by simp
    finally show ?thesis .
  qed
next
  fix x xs
  assume HI: "xs @ (ys @ zs) = (xs @ ys) @ zs"
  show "(x#xs) @ (ys @ zs) = ((x#xs) @ ys) @ zs"
  proof -
    have "(x#xs) @ (ys @ zs) = x#(xs @ (ys @ zs))" by simp
    also have " = x#((xs @ ys) @ zs)" using HI by simp
    also have " = (x#(xs @ ys)) @ zs" by simp
    also have " = ((x#xs) @ ys) @ zs" by simp
    finally show ?thesis .
  qed
qed

text {*
  Ejemplo [Árboles binarios]
  Definir un tipo de dato para los árboles binarios.
*}

datatype 'a arbolB = Hoja "'a"
                   | Nodo "'a" "'a arbolB" "'a arbolB"

text {*
  Ejemplo. [Imagen especular]
  Definir la función "espejo" que aplicada a un árbol devuelve su imagen
  especular.
*}

fun espejo :: "'a arbolB  'a arbolB" where
  "espejo (Hoja a) = (Hoja a)"
| "espejo (Nodo f x y) = (Nodo f (espejo y) (espejo x))"

text {*
  Ejemplo [La imagen especular es involutiva]
  Demostrar que la función "espejo" involutiva; es decir, para cualquier
```

```
  árbol t, se tiene que
     espejo (espejo(t)) = t.
*}

-- "La demostración automática es"
lemma espejo_involutiva_1: "espejo(espejo(t)) = t"
by (induct t) auto

-- "La demostración estructurada es"
lemma espejo_involutiva: "espejo(espejo(t)) = t" (is "?P t")
proof (induct t)
  fix x :: 'a show "?P (Hoja x)" by simp
next
  fix t1 :: "'a arbolB" assume h1: "?P t1"
  fix t2 :: "'a arbolB" assume h2: "?P t2"
  fix x :: 'a
  show "?P (Nodo x t1 t2)"
  proof -
    have "espejo(espejo(Nodo x t1 t2)) = espejo(Nodo x (espejo t2) (espejo t1))"
      by simp
    also have " = Nodo x (espejo (espejo t1)) (espejo (espejo t2))" by simp
    also have " = Nodo x t1 t2" using h1 h2 by simp
    finally show ?thesis .
 qed
qed

text {*
  Ejemplo. [Aplanamiento de árboles]
  Definir la función "aplana" que aplane los árboles recorriéndolos en
  orden infijo.
*}

fun aplana :: "'a arbolB  'a list" where
  "aplana (Hoja a) = [a]"
| "aplana (Nodo x t1 t2) = (aplana t1)@[x]@(aplana t2)"

text {*
  Ejemplo. [Aplanamiento de la imagen especular] Demostrar que
     aplana (espejo t) = rev (aplana t)
*}
```

```
-- "La demostración automática es"
lemma "aplana (espejo t) = rev (aplana t)"
by (induct t) auto

-- "La demostración estructurada es"
lemma "aplana (espejo t) = rev (aplana t)" (is "?P t")
proof (induct t)
  fix x :: 'a
  show "?P (Hoja x)" by simp
next
  fix t1 :: "'a arbolB" assume h1: "?P t1"
  fix t2 :: "'a arbolB" assume h2: "?P t2"
  fix x :: 'a
  show "?P (Nodo x t1 t2)"
  proof -
    have "aplana (espejo (Nodo x t1 t2)) =
          aplana (Nodo x (espejo t2) (espejo t1))" by simp
    also have " = (aplana(espejo t2))@[x]@(aplana(espejo t1))" by simp
    also have " = (rev(aplana t2))@[x]@(rev(aplana t1))" using h1 h2 by simp
    also have " = rev((aplana t1)@[x]@(aplana t2))" by simp
    also have " = rev(aplana (Nodo x t1 t2))" by simp
    finally show ?thesis .
 qed
qed

section {* Heurísticas para la inducción *}

text {*
  Definición. [Definición recursiva de inversa]
  (inversa xs) la inversa de la lista xs. Por ejemplo,
     inversa [a,b,c] = [c,b,a]
*}

fun inversa :: "'a list  'a list" where
  "inversa [] = []"
| "inversa (x#xs) = (inversa xs) @ [x]"

value "inversa [a,b,c]"
```

```
text {*
  Definición. [Definición de inversa con acumuladores]
  (inversaAc xs) es la inversa de la lista xs calculada con
  acumuladores. Por ejemplo,
     inversaAc [a,b,c]       = [c,b,a]
     inversaAcAux [a,b,c] [] = [c,b,a]
*}

fun inversaAcAux :: "'a list  'a list  'a list" where
  "inversaAcAux [] ys = ys"
| "inversaAcAux (x#xs) ys = inversaAcAux xs (x#ys)"

definition inversaAc :: "'a list  'a list" where
  "inversaAc xs  inversaAcAux xs []"

value "inversaAcAux [a,b,c] []"
value "inversaAc [a,b,c]"

text {*
  Lema. [Ejemplo de equivalencia entre las definiciones]
  La inversa de [a,b,c] es lo mismo calculada con la primera definición
  que con la segunda.
*}

lemma "inversaAc [a,b,c] = inversa [a,b,c]"
by (simp add: inversaAc_def)

text {*
  Nota. [Ejemplo fallido de demostración por inducción]
  El siguiente intento de demostrar que para cualquier lista xs, se
  tiene que  "inversaAc xs = inversa xs" falla.
*}

lemma "inversaAc xs = inversa xs"
proof (induct xs)
  show "inversaAc [] = inversa []" by (simp add: inversaAc_def)
next
  fix a xs assume HI: "inversaAc xs = inversa xs"
  have "inversaAc (a#xs) = inversaAcAux (a#xs) []" by (simp add: inversaAc_def)
  also have " = inversaAcAux xs [a]" by simp
```

```
  also have " = inversa (a#xs)"
  -- "Problema: la hipótesis de inducción no es aplicable."
oops

text {*
  Nota. [Heurística de generalización]
  Cuando se use demostración estructural, cuantificar universalmente las
  variables libres (o, equivalentemente, considerar las variables libres
  como variables arbitrarias).

  Lema. [Lema con generalización]
  Para toda lista ys se tiene
     inversaAcAux xs ys = (inversa xs) @ ys
*}

-- "La demostración automática es"
lemma inversaAcAux_es_inversa_1:
  "inversaAcAux xs ys = (inversa xs)@ys"
by (induct xs arbitrary: ys) auto

-- "La demostración estructurada es"
lemma inversaAcAux_es_inversa:
  "inversaAcAux xs ys = (inversa xs)@ys"
proof (induct xs arbitrary: ys)
  show "ys. inversaAcAux [] ys = (inversa [])@ys" by simp
next
  fix a xs
  assume HI: "ys. inversaAcAux xs ys = inversa xs@ys"
  show "ys. inversaAcAux (a#xs) ys = inversa (a#xs)@ys"
  proof -
    fix ys
    have "inversaAcAux (a#xs) ys = inversaAcAux xs (a#ys)" by simp
    also have " = inversa xs@(a#ys)" using HI by simp
    also have " = inversa (a#xs)@ys" by simp
    finally show "inversaAcAux (a#xs) ys = inversa (a#xs)@ys" by simp
  qed
qed

text {*
  Corolario.  Para cualquier lista xs, se tiene que
```

```
     inversaAc xs = inversa xs
*}

corollary "inversaAc xs = inversa xs"
by (simp add: inversaAcAux_es_inversa inversaAc_def)

text {*
  Nota. En el paso "inversa xs@(a#ys) = inversa (a#xs)@ys" se usan
  lemas de la teoría List. Se puede observar, activando "Trace
  Simplifier" y D"|Trace Rules", que los lemas usados son
  ů append_assoc:       (xs @ ys) @ zs = xs @ (ys @ zs)
  ů append.append_Cons: (x#xs)@ys = x#(xs@ys)
  ů append.append_Nil:  []@ys = ys
  Los dos últimos son las ecuaciones de la definición de append.

  En la siguiente demostración se detallan los lemas utilizados.
*}

lemma "(inversa xs)@(a#ys) = (inversa (a#xs))@ys"
proof -
  have "(inversa xs)@(a#ys) = (inversa xs)@(a#([]@ys))"
    by (simp only:append.append_Nil)
  also have " = (inversa xs)@([a]@ys)" by (simp only:append.append_Cons)
  also have " = ((inversa xs)@[a])@ys" by (simp only:append_assoc)
  also have " = (inversa (a#xs))@ys" by (simp only:inversa.simps(2))
  finally show ?thesis .
qed

section {* Recursión general. La función de Ackermann *}

text {*
  El objetivo de esta sección es mostrar el uso de las definiciones
  recursivas generales y sus esquemas de inducción. Como ejemplo se usa la
  función de Ackermann (se puede consultar información sobre dicha función en
  http://en.wikipedia.org/wiki/Ackermann_function).

  Definición.  La función de Ackermann se define por
    A(m,n) = n+1,              si m=0,
             A(m-1,1),         si m>0 y n=0,
             A(m-1,A(m,n-1)),  si m>0 y n>0
```

```
  para todo los números naturales.

  La función de Ackermann es recursiva, pero no es primitiva recursiva.
*}

fun ack :: "nat  nat  nat" where
  "ack 0 n = n+1"
| "ack (Suc m) 0 = ack m 1"
| "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"

text {*
  Nota. [Ejemplo de cálculo]
  El cálculo del valor de la función de Ackermann para 2 y 3 se realiza
  mediante "value"
*}

value "ack 2 3" (* devuelve 9 *)

text {*
  Nota. [Definiciones recursivas generales]
  ů Las definiciones recursivas generales se identifican mediante "fun".
  ů Al definir una función recursiva general se genera una regla de
    inducción. En la definición anterior, la regla generada es
    ack.induct:
       n. P 0 n;
        m. P m 1  P (Suc m) 0;
        m n. P (Suc m) n; P m (ack (Suc m) n)  P (Suc m) (Suc n)
        P a b

  Lema. Para todos m y n, A(m,n) > n.
*}

text {*
  El lema anterior se puede demostrar automáticamente, como sigue.
*}

-- "La demostración automática es"
lemma "ack m n > n"
by (induct m n rule: ack.induct) simp_all
```

```
-- "La demostración detallada es"
lemma "ack m n > n"
proof (induct m n rule: ack.induct)
  fix n :: "nat"
  show "ack 0 n > n" by simp
next
  fix m assume "ack m 1 > 1"
  thus "ack (Suc m) 0 > 0" by simp
next
  fix m n
  assume "n < ack (Suc m) n" and
         "ack (Suc m) n < ack m (ack (Suc m) n)"
  thus "Suc n < ack (Suc m) (Suc n)" by simp
qed

text {*
  Nota. [Inducción sobre recursión]
  El formato para iniciar una demostración por inducción en la regla
  inductiva correspondiente a la definición recursiva de la función f m
  n es
     proof (induct m n rule:f.induct)
*}

section {* Recursión mutua e inducción *}

text {*
  Nota. [Ejemplo de definición de tipos mediante recursión cruzada]
  ů Un árbol de tipo a es una hoja o un nodo de tipo a junto con un
    bosque de tipo a.
  ů Un bosque de tipo a es el boque vacío o un bosque contruido añadiendo
    un árbol de tipo a a un bosque de tipo a.
*}

datatype 'a arbol = Hoja | Nodo "'a" "'a bosque"
and      'a bosque = Vacio | ConsB "'a arbol" "'a bosque"

text {*
  Nota. [Regla de inducción correspondiente a la recursión cruzada]
  La regla de inducción sobre árboles y bosques es arbol_bosque.induct:
     P1 Hoja;
```

```
      x b. P2 b  P1 (Nodo x b);
      P2 Vacio;
      a b. P1 a; P2 b  P2 (ConsB a b)
      P1 a  P2 b

  Nota. [Ejemplos de definición por recursión cruzada]
  ů aplana_arbol a) es la lista obtenida aplanando el árbol a.
  ů (aplana_bosque b) es la lista obtenida aplanando el bosque b.
  ů (map_arbol a h) es el árbol obtenido aplicando la función h a
    todos los nodos del árbol a.
  ů (map_bosque b h) es el bosque obtenido aplicando la función h a
    todos los nodos del bosque b.
*}

fun
  aplana_arbol :: "'a arbol  'a list" and
  aplana_bosque :: "'a bosque  'a list" where
  "aplana_arbol Hoja = []"
| "aplana_arbol (Nodo x b) = x#(aplana_bosque b)"
| "aplana_bosque Vacio = []"
| "aplana_bosque (ConsB a b) = (aplana_arbol a) @ (aplana_bosque b)"

fun
  map_arbol :: "'a arbol  ('a  'b)  'b arbol" and
  map_bosque :: "'a bosque  ('a  'b)  'b bosque" where
  "map_arbol Hoja h = Hoja"
| "map_arbol (Nodo x b) h = Nodo (h x) (map_bosque b h)"
| "map_bosque Vacio h = Vacio"
| "map_bosque (ConsB a b) h = ConsB (map_arbol a h) (map_bosque b h)"

text {*
  Lema. [Ejemplo de inducción cruzada]
  ů aplana_arbol (map_arbol a h) = map h (aplana_arbol a)
  ů aplana_bosque (map_bosque b h) = map h (aplana_bosque b)
*}

-- "La demostración automática es"
lemma "aplana_arbol (map_arbol a h) = map h (aplana_arbol a)
      aplana_bosque (map_bosque b h) = map h (aplana_bosque b)"
by (induct_tac a and b) auto
```

```
-- "La demostración detallada es"
lemma "aplana_arbol (map_arbol a h) = map h (aplana_arbol a)
      aplana_bosque (map_bosque b h) = map h (aplana_bosque b)"
proof (induct_tac a and b)
  show "aplana_arbol (map_arbol Hoja h) = map h (aplana_arbol Hoja)" by simp
next
  fix x b
  assume HI: "aplana_bosque (map_bosque b h) = map h (aplana_bosque b)"
  have "aplana_arbol (map_arbol (Nodo x b) h)
        = aplana_arbol (Nodo (h x) (map_bosque b h))" by simp
  also have " = (h x)#(aplana_bosque (map_bosque b h))" by simp
  also have " = (h x)#(map h (aplana_bosque b))" using HI by simp
  also have " = map h (aplana_arbol (Nodo x b))" by simp
  finally show "aplana_arbol (map_arbol (Nodo x b) h)
                = map h (aplana_arbol (Nodo x b))" .
next
  show "aplana_bosque (map_bosque Vacio h) = map h (aplana_bosque Vacio)"
    by simp
next
  fix a b
  assume HI1: "aplana_arbol (map_arbol a h) = map h (aplana_arbol a)"
     and HI2: "aplana_bosque (map_bosque b h) = map h (aplana_bosque b)"
  have "aplana_bosque (map_bosque (ConsB a b) h)
        = aplana_bosque (ConsB (map_arbol a h) (map_bosque b h))" by simp
  also have " = aplana_arbol(map_arbol a h)@aplana_bosque(map_bosque b h)"
    by simp
  also have " = (map h (aplana_arbol a))@(map h (aplana_bosque b))"
    using HI1 HI2 by simp
  also have " = map h (aplana_bosque (ConsB a b))" by simp
  finally show "aplana_bosque (map_bosque (ConsB a b) h)
                = map h (aplana_bosque (ConsB a b))" by simp
qed

end
```

6.1. Ejercicios de inducción sobre listas

6.1.1. Cons inverso y cuantificadores sobre listas

```
chapter {* T6R1a: Cons inverso y cuantificadores sobre listas *}

theory T6R1a
imports Main
begin

section {* Cons inverso *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir recursivamente la función
     snoc :: "'a list  'a  'a list"
  tal que (snoc xs a) es la lista obtenida al añadir el elemento a al
  final de la lista xs. Por ejemplo,
     value "snoc [2,5] (3::int)" == [2,5,3]

  Nota: No usar @.
  ----------------------------------------------------------------------
*}

fun snoc :: "'a list  'a  'a list" where
  "snoc [] a = [a]"
| "snoc (x#xs) a = x # (snoc xs a)"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar el siguiente teorema
     snoc xs a = xs @ [a]
  ----------------------------------------------------------------------
*}

-- "La demostración automática del lema es"
lemma snoc_append: "snoc xs a = xs @ [a]"
by (induct xs) auto

-- "La demostración semiautomática del lema es"
lemma snoc_append_2: "snoc xs a = xs @ [a]"
```

```
proof (induct xs)
  show "snoc [] a = [] @ [a]" by auto
next
  fix b xs assume "snoc xs a = xs @ [a]"
  thus "snoc (b # xs) a = (b # xs) @ [a]" by auto
qed

-- "La demostración estructurada del lema es"
lemma snoc_append_3: "snoc xs a = xs @ [a]"
proof (induct "xs")
  show "snoc [] a = [] @ [a]"
  proof -
    have "snoc [] a = [a]" by simp
    also have " = [] @ [a]" by simp
    finally show "snoc [] a = [] @ [a]" .
  qed
next
  fix b xs assume HI: "snoc xs a = xs @ [a]"
  show "snoc (b # xs) a = (b # xs) @ [a]"
  proof -
    have "snoc (b # xs) a = b # (snoc xs a)" by simp
    also have " = b # (xs @ [a])" using HI by simp
    also have " = (b # xs) @ [a]" by simp
    finally show "snoc (b # xs) a = (b # xs) @ [a]" .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 3. Demostrar el siguiente teorema
     rev (x # xs) = snoc (rev xs) x"
  ---------------------------------------------------------------------
*}

-- "La demostración automática del teorema es"
theorem rev_cons: "rev (x # xs) = snoc (rev xs) x"
by (auto simp add: snoc_append)

-- "La demostración estructurada del teorema es"
theorem rev_cons_2: "rev (x # xs) = snoc (rev xs) x"
```

```
proof -
  have "rev (x # xs) = (rev xs) @ [x]" by simp
  also have " = snoc (rev xs) x" by (simp add:snoc_append)
  finally show "rev (x # xs) = snoc (rev xs) x" .
qed

section {* Cuantificadores sobre listas *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Definir la función
     todos :: ('a  bool)  'a list  bool
  tal que (todos p xs) se verifica si todos los elementos de la lista
  xs cumplen la propiedad p. Por ejemplo, se verifica
     todos (x. 1<length x) [[2,1,4],[1,3]]
     ňtodos (x. 1<length x) [[2,1,4],[3]]

  Nota: La función todos es equivalente a la predefinida list_all.
  ----------------------------------------------------------------------
*}

fun todos :: "('a  bool)  'a list  bool" where
  "todos p []     = True"
| "todos p (y#ys) = ((p y)  (todos p ys))"

value "todos (x. 1<length x) [[2,1,4],[1,3]]" -- "= True"
value "todos (x. 1<length x) [[2,1,4],[3]]"   -- "= False"

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Definir la función
     algunos :: ('a  bool)  'a list  bool
  tal que (algunos p xs) se verifica si algunos elementos de la lista
  xs cumplen la propiedad p. Por ejemplo, se verifica
     algunos (x. 1<length x) [[2,1,4],[3]]
     ňalgunos (x. 1<length x) [[],[3]]"

  Nota: La función algunos es equivalente a la predefinida list_ex.
  ----------------------------------------------------------------------
*}
```

```
fun algunos  :: "('a  bool)  'a list  bool" where
  "algunos p []     = False"
| "algunos p (x#xs) = ((p x)  (algunos p xs))"

value "algunos (x. 1<length x) [[2,1,4],[3]]" -- "= True"
value "algunos (x. 1<length x) [[],[3]]"      -- "= False"

text {*
  ---------------------------------------------------------------------
  Ejercicio 6. Demostrar o refutar:
     todos (x. P x  Q x) xs = (todos P xs  todos Q xs)
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma "todos (x. P x  Q x) xs = (todos P xs  todos Q xs)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma "todos (x. P x  Q x) xs = (todos P xs  todos Q xs)"
proof (induct xs)
  show "todos (x. P x  Q x) [] = (todos P []  todos Q [])" by simp
next
  fix a xs
  assume "todos (x. P x  Q x) xs = (todos P xs  todos Q xs)"
  thus "todos (x. P x  Q x) (a#xs) = (todos P (a#xs)  todos Q (a#xs))"
    by auto
qed

-- "La demostración detallada es"
lemma "todos (x. P x  Q x) xs = (todos P xs  todos Q xs)"
proof (induct xs)
  show "todos (x. P x  Q x) [] = (todos P []  todos Q [])" by simp
next
  fix a xs
  assume HI: "todos (x. P x  Q x) xs = (todos P xs  todos Q xs)"
  show "todos (x. P x  Q x) (a#xs) = (todos P (a#xs)  todos Q (a#xs))"
  proof -
    have "todos (x. P x  Q x) (a#xs) =
```

```
        ((P a)  (Q a)  todos (x. P x  Q x) xs)" by simp
    also have " = ((P a)  (Q a)  todos P xs  todos Q xs)" using HI by simp
    also have " = (((P a)  todos P xs)  ((Q a)  todos Q xs))" by auto
    also have " = (todos P (a#xs)  todos Q (a#xs))" by simp
    finally show ?thesis .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Demostrar o refutar:
     todos P (x @ y) = (todos P x  todos P y)
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma todos_append [simp]:
  "todos P (x @ y) = (todos P x  todos P y)"
by (induct x) simp_all

-- "La demostración estructurada es"
lemma todos_append_2:
  "todos P (x @ y) = (todos P x  todos P y)"
proof (induct x)
  show "todos P ([] @ y) = (todos P []  todos P y)" by simp
next
  fix a x
  assume "todos P (x @ y) = (todos P x  todos P y)"
  thus "todos P ((a#x) @ y) = (todos P (a#x)  todos P y)"
    by auto
qed

-- "La demostración detallada es"
lemma todos_append_3:
  "todos P (x @ y) = (todos P x  todos P y)"
proof (induct x)
  show "todos P ([] @ y) = (todos P []  todos P y)" by simp
next
  fix a x
  assume HI: "todos P (x @ y) = (todos P x  todos P y)"
```

```
  show "todos P ((a#x) @ y) = (todos P (a#x)  todos P y)"
  proof -
    have "todos P ((a#x) @ y) = ((P a)  todos P (x@y))" by simp
    also have " = ((P a)  todos P x  todos P y)" using HI by simp
    also have " = (todos P (a#x)  todos P y)" by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 8. Demostrar o refutar:
     todos P (rev xs) = todos P xs
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma "todos P (rev xs) = todos P xs"
by (induct xs) auto

-- "La demostración estructurada es"
lemma "todos P (rev xs) = todos P xs"
proof (induct xs)
  show "todos P (rev []) = todos P []" by simp
next
  fix a xs
  assume "todos P (rev xs) = todos P xs"
  thus "todos P (rev (a#xs)) = todos P (a#xs)" by auto
qed

-- "La demostración detallada es"
lemma "todos P (rev xs) = todos P xs"
proof (induct xs)
  show "todos P (rev []) = todos P []" by simp
next
  fix a xs
  assume HI: "todos P (rev xs) = todos P xs"
  show "todos P (rev (a#xs)) = todos P (a#xs)"
  proof -
    have "todos P (rev (a#xs)) = todos P ((rev xs)@[a])" by simp
```

```
    also have " = (todos P (rev xs)  todos P [a])" by simp
    also have " = (todos P xs  todos P [a])" using HI by simp
    also have " = (todos P [a]  todos P xs)" by auto
    also have " = (P a  todos P xs)" by simp
    also have " = todos P (a#xs)" by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 9. Demostrar o refutar:
    algunos (x. P x  Q x) xs = (algunos P xs  algunos Q xs)
  ---------------------------------------------------------------------
*}

-- "Se busca un contraejemplo con nitpick"
lemma "algunos (x. P x  Q x) xs = (algunos P xs  algunos Q xs)"
nitpick
oops

text {*
  El contraejemplo encontrado es
     Nitpick found a counterexample for card 'a = 3:

     Free variables:
       P = (x. _)(a\<^bsub>1\<^esub> := True, a\<^bsub>2\<^esub> := True, a\<^bsub>3\
       Q = (x. _)(a\<^bsub>1\<^esub> := False, a\<^bsub>2\<^esub> := False, a\<^bsub>
       xs = [a\<^bsub>3\<^esub>, a\<^bsub>2\<^esub>]
*}

text {*
  ---------------------------------------------------------------------
  Ejercicio 10. Demostrar o refutar:
     algunos P (map f xs) = algunos (P  f) xs
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma "algunos P (map f xs) = algunos (P o f) xs"
```

```
by (induct xs) simp_all

-- "La demostración estructurada es"
lemma "algunos P (map f xs) = algunos (P  f) xs"
proof (induct xs)
  show "algunos P (map f []) = algunos (P  f) []" by simp
next
  fix a xs
  assume "algunos P (map f xs) = algunos (P  f) xs"
  thus "algunos P (map f (a#xs)) = algunos (P  f) (a#xs)" by auto
qed

-- "La demostración detallada es"
lemma "algunos P (map f xs) = algunos (P  f) xs"
proof (induct xs)
  show "algunos P (map f []) = algunos (P  f) []" by simp
next
  fix a xs
  assume HI: "algunos P (map f xs) = algunos (P  f) xs"
  show "algunos P (map f (a#xs)) = algunos (P  f) (a#xs)"
  proof -
    have "algunos P (map f (a#xs)) = algunos P ((f a)#(map f xs))" by simp
    also have " = ((P (f a))  (algunos P (map f xs)))" by simp
    also have " = (((P  f) a)  (algunos (P  f) xs))" using HI by simp
    also have " = algunos (P  f) (a#xs)" by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 11. Demostrar o refutar:
     algunos P (xs @ ys) = (algunos P xs  algunos P ys)
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma algunos_append:
  "algunos P (xs @ ys) = (algunos P xs  algunos P ys)"
by (induct xs) simp_all
```

```
-- "La demostración estructurada es"
lemma algunos_append_2:
  "algunos P (xs @ ys) = (algunos P xs  algunos P ys)"
proof (induct xs)
  show "algunos P ([] @ ys) = (algunos P []  algunos P ys)" by simp
next
  fix a xs
  assume "algunos P (xs @ ys) = (algunos P xs  algunos P ys)"
  thus "algunos P ((a#xs) @ ys) = (algunos P (a#xs)  algunos P ys)"
    by auto
qed

-- "La demostración detallada es"
lemma algunos_append_3:
  "algunos P (xs @ ys) = (algunos P xs  algunos P ys)"
proof (induct xs)
  show "algunos P ([] @ ys) = (algunos P []  algunos P ys)" by simp
next
  fix a xs
  assume HI: "algunos P (xs @ ys) = (algunos P xs  algunos P ys)"
  show "algunos P ((a#xs) @ ys) = (algunos P (a#xs)  algunos P ys)"
  proof -
    have "algunos P ((a#xs) @ ys) = algunos P (a#(xs @ ys))" by simp
    also have " = ((P a)  algunos P (xs @ ys))" by simp
    also have " = ((P a)  algunos P xs  algunos P ys)" using HI by simp
    also have " = (algunos P (a#xs)  algunos P ys)" by simp
    finally show ?thesis .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Demostrar o refutar:
     algunos P (rev xs) = algunos P xs
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma "algunos P (rev xs) = algunos P xs"
```

```
by (induct xs) (auto simp add: algunos_append)

-- "La demostración estructurada es"
lemma "algunos P (rev xs) = algunos P xs"
proof (induct xs)
 show "algunos P (rev []) = algunos P []" by simp
next
  fix a xs
  assume "algunos P (rev xs) = algunos P xs"
  thus "algunos P (rev (a#xs)) = algunos P (a#xs)"
    by (auto simp add: algunos_append)
qed

-- "La demostración detallada es"
lemma "algunos P (rev xs) = algunos P xs"
proof (induct xs)
 show "algunos P (rev []) = algunos P []" by simp
next
  fix a xs
  assume HI: "algunos P (rev xs) = algunos P xs"
  show "algunos P (rev (a#xs)) = algunos P (a#xs)"
  proof -
    have "algunos P (rev (a#xs)) = algunos P ((rev xs) @ [a])" by simp
    also have " = (algunos P (rev xs)  algunos P [a])"
      by (simp add: algunos_append)
    also have " = (algunos P xs  algunos P [a])" using HI by simp
    also have " = (algunos P xs  P a)" by simp
    also have " = (P a  algunos P xs)" by auto
    also have " = algunos P (a#xs)" by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 13. Encontrar un término no trivial Z tal que sea cierta la
  siguiente ecuación:
     algunos (x. P x  Q x) xs = Z
  ---------------------------------------------------------------------
*}
```

```
text {*
  Solución: La ecuación se verifica eligiendo como Z el término
     algunos P xs  algunos Q xs
  En efecto,
*}

lemma "algunos (x. P x  Q x) xs = (algunos P xs  algunos Q xs)"
by (induct xs) auto

-- "De forma estructurada"
lemma "algunos (x. P x  Q x) xs = (algunos P xs  algunos Q xs)"
proof (induct xs)
  show "algunos  (x. P x  Q x) [] = (algunos P []  algunos Q [])" by simp
next
  fix a xs
  assume "algunos (x. (P x  Q x)) xs = (algunos P xs  algunos Q xs)"
  thus "algunos (x. P x  Q x) (a#xs) = (algunos P (a#xs)  algunos Q (a#xs))"
    by auto
qed

-- "De forma detallada"
lemma "algunos (x. P x  Q x) xs = (algunos P xs  algunos Q xs)"
proof (induct xs)
  show "algunos  (x. P x  Q x) [] = (algunos P []  algunos Q [])" by simp
next
  fix a xs
  assume HI: "algunos (x. (P x  Q x)) xs = (algunos P xs  algunos Q xs)"
  show "algunos (x. P x  Q x) (a#xs) = (algunos P (a#xs)  algunos Q (a#xs))"
  proof -
    have "algunos (x. P x  Q x) (a#xs) =
      ((P a)  (Q a)  algunos (x. P x  Q x) xs)" by simp
    also have " = ((P a)  (Q a)  algunos P xs  algunos Q xs)"
      using HI by simp
    also have " = (((P a)  algunos P xs)  ((Q a)  algunos Q xs))" by auto
    also have " = (algunos P (a#xs)  algunos Q (a#xs))" by simp
    finally show ?thesis .
  qed
qed
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 14. Demostrar o refutar:
     algunos P xs = (ň todos (x. (ň P x)) xs)
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma "algunos P xs = (ň todos (x. (ň P x)) xs)"
by (induct xs) simp_all

-- "La demostración estructurada es"
lemma "algunos P xs = (ň todos (x. (ň P x)) xs)"
proof (induct xs)
  show "algunos P [] = (ň todos (x. (ň P x)) [])" by simp
next
  fix a xs
  assume "algunos P xs = (ň todos (x. (ň P x)) xs)"
  thus "algunos P (a#xs) = (ň todos (x. (ň P x)) (a#xs))"
    by auto
qed

-- "La demostración detallada es"
lemma "algunos P xs = (ň todos (x. (ň P x)) xs)"
proof (induct xs)
  show "algunos P [] = (ň todos (x. (ň P x)) [])" by simp
next
  fix a xs
  assume HI: "algunos P xs = (ň todos (x. (ň P x)) xs)"
  show "algunos P (a#xs) = (ň todos (x. (ň P x)) (a#xs))"
  proof -
    have "algunos P (a#xs) = ((P a)  algunos P xs)" by simp
    also have " = ((P a)  ň todos (x. (ň P x)) xs)" using HI by simp
    also have " = (ň (ň (P a)  todos (x. (ň P x)) xs))" by simp
    also have " = (ň todos (x. (ň P x)) (a#xs))" by simp
    finally show ?thesis .
  qed
qed

text {*
```

```
  ------------------------------------------------------------------
  Ejercicio 15. Definir la funcion primitiva recursiva
     estaEn :: 'a  'a list  bool
  tal que (estaEn x xs) se verifica si el elemento x está en la lista
  xs. Por ejemplo,
     estaEn (2::nat) [3,2,4] = True
     estaEn (1::nat) [3,2,4] = False
  ------------------------------------------------------------------
*}

fun estaEn :: "'a  'a list  bool" where
  "estaEn x []     = False"
| "estaEn x (a#xs) = (x=a  estaEn x xs)"

value "estaEn (2::nat) [3,2,4]" -- "= True"
value "estaEn (1::nat) [3,2,4]" -- "= False"

text {*
  ------------------------------------------------------------------
  Ejercicio 16. Expresar la relación existente entre estaEn y algunos.
  ------------------------------------------------------------------
*}

text {*
  Solución: La relación es
     estaEn y xs = algunos (x. x=y) xs
  En efecto,
*}

lemma estaEn_algunos:
  "estaEn y xs = algunos (x. x=y) xs"
by (induct xs) auto

-- "La demostración estructurada es"
lemma estaEn_algunos_2:
  "estaEn y xs = algunos (x. x=y) xs"
proof (induct xs)
  show "estaEn y [] = algunos (x. x=y) []" by simp
next
  fix a xs
```

```
  assume "estaEn y xs = algunos (x. x=y) xs"
  thus "estaEn y (a#xs) = algunos (x. x=y) (a#xs)" by auto
qed

-- "La demostración detallada es"
lemma estaEn_algunos_3:
  "estaEn y xs = algunos (x. x=y) xs"
proof (induct xs)
  show "estaEn y [] = algunos (x. x=y) []" by simp
next
  fix a xs
  assume HI: "estaEn y xs = algunos (x. x=y) xs"
  show "estaEn y (a#xs) = algunos (x. x=y) (a#xs)"
  proof -
    have "estaEn y (a#xs) = (y=a  estaEn y xs)" by simp
    also have " = (y=a  algunos (x. x=y) xs)" using HI by simp
    also have " = (a=y  algunos (x. x=y) xs)" by auto
    also have " = algunos (x. x=y) (a#xs)" by simp
    finally show ?thesis .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 17. Definir la función primitiva recursiva
     sinDuplicados :: 'a list  bool
  tal que (sinDuplicados xs) se verifica si la lista xs no contiene
  duplicados. Por ejemplo,
     sinDuplicados [1::nat,4,2]   = True
     sinDuplicados [1::nat,4,2,4] = False
  ----------------------------------------------------------------------
*}

fun sinDuplicados :: "'a list  bool" where
  "sinDuplicados [] = True"
| "sinDuplicados (a#xs) = ((ň estaEn a xs)  sinDuplicados xs)"

value "sinDuplicados [1::nat,4,2]"   -- "= True"
value "sinDuplicados [1::nat,4,2,4]" -- "= False"
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 18. Definir la función primitiva recursiva
     borraDuplicados :: 'a list  bool
  tal que (borraDuplicados xs) es la lista obtenida eliminando los
  elementos duplicados de la lista xs. Por ejemplo,
     borraDuplicados [1::nat,2,4,2,3] = [1,4,2,3]

  Nota: La función borraDuplicados es equivalente a la predefinida remdups.
  ----------------------------------------------------------------------
*}

fun borraDuplicados :: "'a list  'a list" where
  "borraDuplicados []     = []"
| "borraDuplicados (a#xs) = (if estaEn a xs
                             then borraDuplicados xs
                             else (a#borraDuplicados xs))"

value "borraDuplicados [1::nat,2,4,2,3]" -- "= [1,4,2,3]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 19. Demostrar o refutar:
     length (borraDuplicados xs)  length xs
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma length_borraDuplicados:
  "length (borraDuplicados xs)  length xs"
by (induct xs) simp_all

-- "La demostración estructurada es"
lemma length_borraDuplicados_2:
  "length (borraDuplicados xs)  length xs"
proof (induct xs)
  show "length (borraDuplicados [])  length []" by simp
next
  fix a xs
  assume HI: "length (borraDuplicados xs)  length xs"
```

```
  thus "length (borraDuplicados (a#xs))  length (a#xs)"
  proof (cases)
    assume "estaEn a xs"
    thus "length (borraDuplicados (a#xs))  length (a#xs)"
      using HI by auto
  next
    assume "(ň estaEn a xs)"
    thus "length (borraDuplicados (a#xs))  length (a#xs)"
      using HI by auto
  qed
qed

-- "La demostración detallada es"
lemma length_borraDuplicados_3:
  "length (borraDuplicados xs)  length xs"
proof (induct xs)
  show "length (borraDuplicados [])  length []" by simp
next
  fix a xs
  assume HI: "length (borraDuplicados xs)  length xs"
  show "length (borraDuplicados (a#xs))  length (a#xs)"
  proof (cases)
    assume "estaEn a xs"
    hence "length (borraDuplicados (a#xs)) = length (borraDuplicados xs)"
      by simp
    also have "  length xs" using HI by simp
    also have "  length (a#xs)" by simp
    finally show ?thesis .
  next
    assume "(ň estaEn a xs)"
    hence "length (borraDuplicados (a#xs)) = length (a#(borraDuplicados xs))"
      by simp
    also have " = 1 + length (borraDuplicados xs)" by simp
    also have "  1 + length xs" using HI by simp
    also have " = length (a#xs)" by simp
    finally show ?thesis .
  qed
qed

text {*
```

```
  ------------------------------------------------------------------
  Ejercicio 20. Demostrar o refutar:
     estaEn a (borraDuplicados xs) = estaEn a xs
  ------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma estaEn_borraDuplicados:
  "estaEn a (borraDuplicados xs) = estaEn a xs"
by (induct xs) auto

-- "La demostración estructurada es"
lemma estaEn_borraDuplicados_2:
  "estaEn a (borraDuplicados xs) = estaEn a xs"
proof (induct xs)
  show "estaEn a (borraDuplicados []) = estaEn a []" by simp
next
  fix b xs
  assume HI: "estaEn a (borraDuplicados xs) = estaEn a xs"
  show "estaEn a (borraDuplicados (b#xs)) = estaEn a (b#xs)"
  proof (rule iffI)
    assume c1: "estaEn a (borraDuplicados (b#xs))"
    show "estaEn a (b#xs)"
    proof (cases)
      assume "estaEn b xs"
      thus "estaEn a (b#xs)" using c1 HI by auto
    next
      assume "ň estaEn b xs"
      thus "estaEn a (b#xs)" using c1 HI by auto
    qed
  next
    assume c2: "estaEn a (b#xs)"
    show "estaEn a (borraDuplicados (b#xs))"
    proof (cases)
      assume "a=b"
      thus "estaEn a (borraDuplicados (b#xs))" using HI by auto
    next
      assume "ab"
      thus "estaEn a (borraDuplicados (b#xs))" using `ab` c2 HI by auto
    qed
```

```
  qed
qed

-- "La demostración detallada es"
lemma estaEn_borraDuplicados_3:
  "estaEn a (borraDuplicados xs) = estaEn a xs"
proof (induct xs)
  show "estaEn a (borraDuplicados []) = estaEn a []" by simp
next
  fix b xs
  assume HI: "estaEn a (borraDuplicados xs) = estaEn a xs"
  show "estaEn a (borraDuplicados (b#xs)) = estaEn a (b#xs)"
  proof (rule iffI)
    assume c1: "estaEn a (borraDuplicados (b#xs))"
    show "estaEn a (b#xs)"
    proof (cases)
      assume "estaEn b xs"
      hence "estaEn a (borraDuplicados xs)" using c1 by simp
      hence "estaEn a xs" using HI by simp
      thus "estaEn a (b#xs)" by simp
    next
      assume "ň estaEn b xs"
      hence "estaEn a (b#(borraDuplicados xs))" using c1 by simp
      hence "a=b  (estaEn a (borraDuplicados xs))" by simp
      hence "a=b  (estaEn a xs)" using HI by simp
      thus "estaEn a (b#xs)" by simp
    qed
  next
    assume c2: "estaEn a (b#xs)"
    show "estaEn a (borraDuplicados (b#xs))"
    proof (cases)
      assume "a=b"
      thus "estaEn a (borraDuplicados (b#xs))" using HI by auto
    next
      assume "ab"
      hence "estaEn a xs" using c2 by simp
      hence "estaEn a (borraDuplicados xs)" using HI by simp
      thus "estaEn a (borraDuplicados (b#xs))" using `ab` by simp
    qed
  qed
```

```
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 21. Demostrar o refutar:
     sinDuplicados (borraDuplicados xs)
  ----------------------------------------------------------------------
*}

-- "La demostración automática"
lemma sinDuplicados_borraDuplicados:
  "sinDuplicados (borraDuplicados xs)"
by (induct xs) (auto simp add: estaEn_borraDuplicados)

-- "La demostración estructurada es"
lemma sinDuplicados_borraDuplicados_2:
  "sinDuplicados (borraDuplicados xs)"
proof (induct xs)
  show "sinDuplicados (borraDuplicados [])" by simp
next
  fix a xs
  assume HI: "sinDuplicados (borraDuplicados xs)"
  show "sinDuplicados (borraDuplicados (a#xs))"
  proof (cases)
    assume "estaEn a xs"
    thus "sinDuplicados (borraDuplicados (a#xs))" using HI by simp
  next
    assume "ň estaEn a xs"
    thus "sinDuplicados (borraDuplicados (a#xs))"
      using `ň estaEn a xs` HI
      by (auto simp add: estaEn_borraDuplicados)
  qed
qed

-- "La demostración detallada es"
lemma sinDuplicados_borraDuplicados_3:
  "sinDuplicados (borraDuplicados xs)"
proof (induct xs)
  show "sinDuplicados (borraDuplicados [])" by simp
next
```

```
  fix a xs
  assume HI: "sinDuplicados (borraDuplicados xs)"
  show "sinDuplicados (borraDuplicados (a#xs))"
  proof (cases)
    assume "estaEn a xs"
    thus "sinDuplicados (borraDuplicados (a#xs))" using HI by simp
  next
    assume "¬ estaEn a xs"
    hence "¬ (estaEn a xs)  sinDuplicados (borraDuplicados xs)"
      using HI by simp
    hence "¬ estaEn a (borraDuplicados xs)
           sinDuplicados (borraDuplicados xs)"
      by (simp add: estaEn_borraDuplicados)
    hence "sinDuplicados (a#borraDuplicados xs)" by simp
    thus "sinDuplicados (borraDuplicados (a#xs))"
      using `¬ estaEn a xs` by simp
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 22. Demostrar o refutar:
    borraDuplicados (rev xs) = rev (borraDuplicados xs)
  ---------------------------------------------------------------------
*}

-- "Se busca un contraejemplo con"
lemma "borraDuplicados (rev xs) = rev (borraDuplicados xs)"
quickcheck
oops

text {*
  El contraejemplo encontrado es
     xs = [3, 2, 3]
  En efecto,
      borraDuplicados (rev xs) = borraDuplicados (rev [3,2,3]) = [2,3]
      rev (borraDuplicados xs) = rev (borraDuplicados [3,2,3]) = [3,2]
*}

end
```

6.1.2. Sustitución, inversión y eliminación

```
chapter {* T6R1b: Sustitución, inversión y eliminación *}

theory T6R1b
imports Main
begin

section {* Sustitución, inversión y eliminación *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     sust :: "'a  'a  'a list  'a list"
  tal que (sust x y zs) es la lista obtenida sustituyendo cada
  occurrencia de x por y en la lista zs. Por ejemplo,
     sust (1::nat) 2 [1,2,3,4,1,2,3,4] = [2,2,3,4,2,2,3,4]
  ----------------------------------------------------------------------
*}

fun sust :: "'a  'a  'a list  'a list" where
  "sust x y [] = []"
| "sust x y (z#zs) = (if z=x then y else z)#(sust x y zs)"

value "sust (1::nat) 2 [1,2,3,4,1,2,3,4]" -- "= [2,2,3,4,2,2,3,4]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar o refutar:
     sust x y (xs@ys) = (sust x y xs)@(sust x y ys)"
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma sust_append:
  "sust x y (xs@ys) = (sust x y xs)@(sust x y ys)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma sust_append_2:
  "sust x y (xs @ ys) = (sust x y xs)@(sust x y ys)"
```

```
proof (induct xs)
  show "sust x y ([]@ys) = (sust x y [])@(sust x y ys)" by simp
next
  fix a xs
  assume HI: "sust x y (xs@ys) = (sust x y xs)@(sust x y ys)"
  show "sust x y ((a#xs)@ys) = (sust x y (a#xs))@(sust x y ys)"
  proof (cases)
    assume "x=a"
    thus "sust x y ((a#xs)@ys) = (sust x y (a#xs))@(sust x y ys)"
      using HI by auto
  next
    assume "xa"
    thus "sust x y ((a#xs)@ys) = (sust x y (a#xs))@(sust x y ys)"
      using HI by auto
  qed
qed

-- "La demostración detallada es"
lemma sust_append_3:
  "sust x y (xs @ ys) = (sust x y xs)@(sust x y ys)"
proof (induct xs)
  show "sust x y ([]@ys) = (sust x y [])@(sust x y ys)" by simp
next
  fix a xs
  assume HI: "sust x y (xs@ys) = (sust x y xs)@(sust x y ys)"
  show "sust x y ((a#xs)@ys) = (sust x y (a#xs))@(sust x y ys)"
  proof (cases)
    assume "x=a"
    hence "sust x y ((a#xs)@ys) = sust x y (a#(xs@ys))" by simp
    also have " = y#(sust x y (xs@ys))" using `x=a` by simp
    also have " = y#((sust x y xs)@(sust x y ys))" using HI by simp
    also have " = (y#(sust x y xs))@(sust x y ys)" by simp
    also have " = (sust x y (a#xs))@(sust x y ys)" using `x=a` by simp
    finally show "sust x y ((a#xs)@ys) = (sust x y (a#xs))@(sust x y ys)" .
  next
    assume "xa"
    hence "sust x y ((a#xs)@ys) = sust x y (a#(xs@ys))" by simp
    also have " = a#(sust x y (xs@ys))" using `xa` by simp
    also have " = a#((sust x y xs)@(sust x y ys))" using HI by simp
    also have " = (a#(sust x y xs))@(sust x y ys)" by simp
```

```
    also have " = (sust x y (a#xs))@(sust x y ys)" using `xa` by simp
    finally show "sust x y ((a#xs)@ys) = (sust x y (a#xs))@(sust x y ys)" .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 3. Demostrar o refutar:
     rev (sust x y zs) = sust x y (rev zs)
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
theorem rev_sust:
  "rev(sust x y zs) = sust x y (rev zs)"
by (induct zs) (simp_all add: sust_append)

-- "La demostración estructurada es"
theorem rev_sust_2:
  "rev (sust x y zs) = sust x y (rev zs)"
proof (induct zs)
  show "rev (sust x y []) = sust x y (rev [])" by simp
next
  fix a zs
  assume HI: "rev (sust x y zs) = sust x y (rev zs)"
  show "rev (sust x y (a#zs)) = sust x y (rev (a#zs))"
    using HI by (auto simp add: sust_append)
qed

-- "La demostración detallada es"
theorem rev_sust_3:
  "rev (sust x y zs) = sust x y (rev zs)"
proof (induct zs)
  show "rev (sust x y []) = sust x y (rev [])" by simp
next
  fix a zs
  assume HI: "rev (sust x y zs) = sust x y (rev zs)"
  show "rev (sust x y (a#zs)) = sust x y (rev (a#zs))"
  proof -
    have "rev (sust x y (a#zs)) = rev ((if x=a then y else a)#(sust x y zs))"
```

```
      by simp
    also have " = (rev (sust x y zs))@[if x=a then y else a]" by simp
    also have " = (sust x y (rev zs))@[if x=a then y else a]" using HI by simp
    also have " = (sust x y (rev zs))@(sust x y [a])" by simp
    also have " = sust x y ((rev zs)@[a])" by (simp add: sust_append)
    also have " = sust x y (rev (a#zs))" by simp
    finally show "rev (sust x y (a#zs)) = sust x y (rev (a#zs))" .
  qed
qed

text {*
  ------------------------------------------------------------------------
  Ejercicio 4. Demostrar o refutar:
     sust x y (sust u v zs) = sust u v (sust x y zs)
  ------------------------------------------------------------------------
*}

theorem "sust x y (sust u v zs) = sust u v (sust x y zs)"
quickcheck
oops

text {*
  El contraejemplo encontrado es:
   u = -1
   v = 0
   x = -1
   y = 1
   zs = [-1]
  Efectivamente,
   sust (-1) 1 (sust (-1) 0 [(-1)]) = [0]
   sust (-1) 0 (sust (-1) 1 [(-1)]) = [1]
*}

text {*
  ------------------------------------------------------------------------
  Ejercicio 5. Demostrar o refutar:
     sust y z (sust x y zs) = sust x z zs
  ------------------------------------------------------------------------
*}
```

```
theorem "sust y z (sust x y zs) = sust x z zs"
quickcheck
oops

text {*
  El contraejemplo encontrado es:
  x = 0
  y = 1
  z = 0
  zs = [1]
  En efecto,
  sust 1 0 (sust 0 1 [1]) = [0]
  sust 0 0 [1] = [1]
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 6. Definir la función
     borra :: "'a  'a list  'a list"
  tal que (borra x ys) es la lista obtenida borrando la primera
  ocurrencia del elemento x en la lista ys. Por ejemplo,
     borra (2::nat) [1,2,3,2] = [1,3,2]

  Nota: La función borra es equivalente a la predefinida remove1.
  ----------------------------------------------------------------------
*}

fun borra :: "'a  'a list  'a list" where
  "borra x [] = []"
| "borra x (y#ys) = (if x=y then ys else (y#(borra x ys)))"

value "borra (2::nat) [1,2,3,2]" -- "= [1,3,2]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Definir la función
     borraTodas :: "'a  'a list  'a list"
  tal que (borraTodas x ys) es la lista obtenida borrando todas las
  ocurrencias del elemento x en la lista ys. Por ejemplo,
     borraTodas (2::nat) [1,2,3,2] = [1,3]
```

```
  ----------------------------------------------------------------------
*}

fun borraTodas :: "'a  'a list  'a list" where
  "borraTodas x [] = []"
| "borraTodas x (y#ys) =
    (if x=y
     then (borraTodas x ys)
     else (y#(borraTodas x ys)))"

value "borraTodas (2::nat) [1,2,3,2]" -- "= [1,3]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 8. Demostrar o refutar:
     borra x (borraTodas x xs) = borraTodas x xs
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
theorem "borra x (borraTodas x xs) = borraTodas x xs"
by (induct xs arbitrary: x) simp_all

-- "La demostración estructurada es"
theorem "borra x (borraTodas x xs) = borraTodas x xs"
proof (induct xs arbitrary: x)
  fix x
  show "borra x (borraTodas x []) = borraTodas x []" by simp
next
  fix a x :: "'b" and xs :: "'b list"
  assume HI: "x. borra x (borraTodas x xs) = borraTodas x xs"
  show "borra x (borraTodas x (a#xs)) = borraTodas x (a#xs)"
  proof (cases)
    assume "x=a"
    hence "borra x (borraTodas x (a#xs)) = borra x (borraTodas x xs)" by simp
    also have " = borraTodas x xs" using HI by simp
    also have " = borraTodas x (a#xs)" using `x=a` by simp
    finally show "borra x (borraTodas x (a#xs)) = borraTodas x (a#xs)" .
  next
    assume "xa"
```

```
    hence "borra x (borraTodas x (a#xs)) = borra x (a#(borraTodas x xs))"
      by simp
    also have " = a#(borra x (borraTodas x xs))" using `xa` by simp
    also have " = a#(borraTodas x xs)" using HI by simp
    also have " = borraTodas x (a#xs)" using `xa` by simp
    finally show "borra x (borraTodas x (a#xs)) = borraTodas x (a#xs)" .
    qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 9. Demostrar o refutar:
     borraTodas x (borraTodas x xs) = borraTodas x xs
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
theorem "borraTodas x (borraTodas x xs) = borraTodas x xs"
by (induct xs arbitrary: x) simp_all

-- "La demostración estructurada es"
theorem "borraTodas x (borraTodas x xs) = borraTodas x xs"
proof (induct xs arbitrary: x)
  fix x
  show "borraTodas x (borraTodas x []) = borraTodas x []" by simp
next
  fix a x :: "'b" and xs :: "'b list"
  assume HI: "x. borraTodas x (borraTodas x xs) = borraTodas x xs"
  show "borraTodas x (borraTodas x (a#xs)) = borraTodas x (a#xs)"
  proof (cases)
    assume "x=a"
    hence "borraTodas x (borraTodas x (a#xs)) = borraTodas x (borraTodas x xs)"
      by simp
    also have " = borraTodas x xs" using HI by simp
    also have " = borraTodas x (a#xs)" using `x=a` by simp
    finally show "borraTodas x (borraTodas x (a#xs)) = borraTodas x (a#xs)" .
  next
    assume "xa"
    hence "borraTodas x (borraTodas x (a#xs)) =
      borraTodas x (a#(borraTodas x xs))" by simp
```

```
    also have " = a#(borraTodas x (borraTodas x xs))" using `xa` by simp
    also have " = a#(borraTodas x xs)" using HI by simp
    also have " = borraTodas x (a#xs)" using `xa` by simp
    finally show "borraTodas x (borraTodas x (a#xs)) = borraTodas x (a#xs)" .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 10. Demostrar o refutar:
     borraTodas x (borra x xs) = borraTodas x xs
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
theorem "borraTodas x (borra x xs) = borraTodas x xs"
by (induct xs) simp_all

-- "La demostración estructurada es"
theorem borraTodas_borra:
  "borraTodas x (borra x xs) = borraTodas x xs"
proof (induct xs)
  show "borraTodas x (borra x []) = borraTodas x []" by simp
next
  fix a :: "'a" and xs :: "'a list"
  assume HI: "borraTodas x (borra x xs) = borraTodas x xs"
  show "borraTodas x (borra x (a#xs)) = borraTodas x (a#xs)"
  proof (cases)
    assume "x=a"
    hence "borraTodas x (borra x (a#xs)) = borraTodas x xs" by simp
    also have " = borraTodas x (a#xs)" using `x=a` by simp
    finally show "borraTodas x (borra x (a#xs)) = borraTodas x (a#xs)" .
  next
    assume "xa"
    hence "borraTodas x (borra x (a#xs)) = borraTodas x (a# (borra x xs))"
      by simp
    also have " = a#(borraTodas x (borra x xs))" using `xa` by simp
    also have " = a#(borraTodas x xs)" using HI by simp
    also have " = borraTodas x (a#xs)" using `xa` by simp
    finally show "borraTodas x (borra x (a#xs)) = borraTodas x (a#xs)" .
```

```
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 11. Demostrar o refutar:
     borra x (borra y xs) = borra y (borra x xs)
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
theorem "borra x (borra y xs) = borra y (borra x xs)"
by (induct xs) simp_all

-- "La demostración estructurada es"
theorem "borra x (borra y xs) = borra y (borra x xs)"
proof (induct xs)
  show "borra x (borra y []) = borra y (borra x [])" by simp
next
  fix a xs
  assume HI: "borra x (borra y xs) = borra y (borra x xs)"
  show "borra x (borra y (a # xs)) = borra y (borra x (a # xs))"
  proof (cases)
    assume "x=y"
    thus "borra x (borra y (a # xs)) = borra y (borra x (a # xs))" by simp
  next
    assume "xy"
    show "borra x (borra y (a # xs)) = borra y (borra x (a # xs))"
    proof (cases)
      assume "y=a"
      hence "xa" using `xy`  by simp
      have "borra x (borra y (a # xs)) = borra x xs" using `y=a` by simp
      also have " = borra y (a # (borra x xs))" using `y=a` by simp
      also have " = borra y (borra x (a # xs))" using `xa` by simp
      finally show "borra x (borra y (a # xs)) = borra y (borra x (a # xs))" .
    next
      assume "ya"
      show "borra x (borra y (a # xs)) = borra y (borra x (a # xs))"
        proof (cases)
          assume "x=a"
```

```
        have "borra x (borra y (a # xs)) = borra x (a#(borra y xs))"
          using `ya` by simp
        also have " = borra y xs" using `x=a` by simp
        also have " = borra y (borra x (a#xs))" using `x=a` by simp
        finally show "borra x (borra y (a#xs)) = borra y (borra x (a#xs))" .
      next
        assume "xa"
        have "borra x (borra y (a # xs)) = borra x (a#(borra y xs))"
          using `ya` by simp
        also have " = a#(borra x (borra y xs))" using `xa` by simp
        also have " = a#(borra y (borra x xs))" using HI by simp
        also have " = borra y (a#(borra x xs))" using `ya` by simp
        also have " = borra y (borra x (a#xs))" using `xa` by simp
        finally show "borra x (borra y (a#xs)) = borra y (borra x (a#xs))" .
      qed
  qed
qed

-- "La demostración puede simplificarse como se muestra a continuación"
theorem "borra x (borra y xs) = borra y (borra x xs)"
proof (induct xs)
  show "borra x (borra y []) = borra y (borra x [])" by simp
next
  fix a xs
  assume HI: "borra x (borra y xs) = borra y (borra x xs)"
  show "borra x (borra y (a # xs)) = borra y (borra x (a # xs))"
  proof (cases)
    assume "y=a" thus ?thesis by simp
  next
    assume "ya" thus ?thesis
    proof (cases)
      assume "x=a" thus ?thesis by simp
    next
      assume "xa" thus ?thesis
      proof -
        have "borra x (borra y (a#xs)) = a#(borra x (borra y xs))"
          using `xa` `ya` by simp
        also have " = a#(borra y (borra x xs))" using HI by simp
        also have " = borra y (borra x (a#xs))"
```

```
          using `xa` `ya` by simp
        finally show "borra x (borra y (a # xs)) = borra y (borra x (a # xs))".
      qed
    qed
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 12. Demostrar o refutar el teorema:
     borraTodas x (borra y xs) = borra y (borraTodas x xs)
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
theorem "borraTodas x (borra y xs) = borra y (borraTodas x xs)"
by (induct xs) (auto simp add: borraTodas_borra)

-- "La demostración estructurada es"
theorem "borraTodas x (borra y xs) = borra y (borraTodas x xs)"
proof (induct xs)
  show "borraTodas x (borra y []) = borra y (borraTodas x [])" by simp
next
  fix a xs
  assume HI: "borraTodas x (borra y xs) = borra y (borraTodas x xs)"
  show "borraTodas x (borra y (a#xs)) = borra y (borraTodas x (a#xs))"
  proof (cases)
    assume "x=y"
    show "borraTodas x (borra y (a#xs)) = borra y (borraTodas x (a#xs))"
    proof (cases)
      assume "y=a"
      hence "borraTodas x (borra y (a # xs)) = borraTodas x xs" by simp
      also have " = borraTodas x (borra x xs)"
        by (rule borraTodas_borra[symmetric])
      also have " = borraTodas x (borra y xs)" using `x=y` by simp
      also have " = borra y (borraTodas x xs)" using HI by simp
      also have " = borra y (borraTodas x (a#xs))" using `x=y` `y=a` by simp
      finally show ?thesis .
    next
      assume "ya"
```

```
      hence "borraTodas x (borra y (a # xs)) = a#(borraTodas x (borra y xs))"
        using ‘x=y‘ by simp
      also have " = a#borra y (borraTodas x xs)" using HI by simp
      also have " = borra y (borraTodas x (a # xs))" using ‘x=y‘ ‘ya‘ by simp
      finally show ?thesis .
    qed
  next
    assume "xy"
    show "borraTodas x (borra y (a # xs)) = borra y (borraTodas x (a # xs))"
    proof (cases)
      assume "x=a"
      hence "borraTodas x (borra y (a#xs)) = borraTodas x (a#(borra y xs))"
        using ‘xy‘ by simp
      also have " = borraTodas x (borra y xs)" using ‘x=a‘ by simp
      also have " = borra y (borraTodas x xs)" using HI by simp
      also have " = borra y (borraTodas x (a#xs))" using ‘x=a‘ by simp
      finally show ?thesis .
    next
      assume "x  a"
      show "borraTodas x (borra y (a # xs)) = borra y (borraTodas x (a # xs))"
      proof (cases)
        assume "y = a"
        hence "borraTodas x (borra y (a#xs)) = borraTodas x xs" by simp
        also have " = borra y (a#(borraTodas x xs))" using ‘xa‘ ‘y=a‘ by simp
        also have " = borra y (borraTodas x (a#xs))" using ‘xa‘ by simp
        finally show ?thesis .
      next
        assume "y  a"
        hence "borraTodas x (borra y (a#xs)) = (a#(borraTodas x (borra y xs)))"
          using ‘xa‘ ‘ya‘ by simp
        also have " = (a#borra y (borraTodas x xs))" using HI by simp
        also have " = borra y (borraTodas x (a#xs))" using ‘xa‘ ‘ya‘ by simp
        finally show ?thesis .
      qed
    qed
  qed
qed

text {*
  ----------------------------------------------------------------------
```

```
  Ejercicio 13. Demostrar o refutar:
     borra y (sust x y xs) = borra x xs
  ----------------------------------------------------------------------
*}

theorem "borra y (sust x y xs) = borra x xs"
quickcheck
oops

text {*
  El contraejemplo encontrado es
   x = 1
   xs = [0]
   y = 0
  En efecto,
   borra y (sust x y xs) = borra 0 (sust 1 0 [0]) = []
   borra x xs = borra 1 [0] = [0]
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 14. Demostrar o refutar:
     borraTodas y (sust x y xs) = borraTodas x xs"
  ----------------------------------------------------------------------
*}

theorem "borraTodas y (sust x y xs) = borraTodas x xs"
quickcheck
oops

text {*
  El contraejemplo encontrado es
   x = -1
   xs = [1]
   y = 1
  En efecto,
   borraTodas y (sust x y xs) = borraTodas 1 (sust -1 1 [1]) = []
   borraTodas x xs = borraTodas -1 [1] = [1]
*}
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 15. Demostrar o refutar:
     sust x y (borraTodas x zs) = borraTodas x zs
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
theorem "sust x y (borraTodas x zs) = borraTodas x zs"
by (induct zs) auto

-- "La demostración estructurada es"
theorem sust_borraTodas:
  "sust x y (borraTodas x zs) = borraTodas x zs"
proof (induct zs)
  show "sust x y (borraTodas x []) = borraTodas x []" by simp
next
  fix z zs
  assume HI: "sust x y (borraTodas x zs) = borraTodas x zs"
  show "sust x y (borraTodas x (z#zs)) = borraTodas x (z#zs)"
  proof (cases)
    assume "x=z"
    hence "sust x y (borraTodas x (z#zs)) = sust x y (borraTodas x zs)" by simp
    also have " = borraTodas x zs" using HI by simp
    also have " = borraTodas x (z#zs)" using `x=z` by simp
    finally show ?thesis .
  next
    assume "xz"
    hence "sust x y (borraTodas x (z#zs)) = sust x y (z#(borraTodas x zs))"
      by simp
    also have " = z#(sust x y (borraTodas x zs))" using `xz` by simp
    also have " = z#(borraTodas x zs)" using HI by simp
    also have " = borraTodas x (z#zs)" using `xz` by simp
    finally show ?thesis .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 16. Demostrar o refutar:
```

```
     sust x y (borraTodas z zs) = borraTodas z (sust x y zs)
  ------------------------------------------------------------------------
*}

theorem "sust x y (borraTodas z zs) = borraTodas z (sust x y zs)"
quickcheck
oops

text {*
  El contraejemplo encontrado es
   x = 0
   y = 1
   z = 0
   zs = [0]
  En efecto,
   sust x y (borraTodas z zs) = sust 0 1 (borraTodas 0 [0]) = []
   borraTodas z (sust x y zs) = borraTodas 0 (sust 0 1 [0]) = [1]
*}

text {*
  ------------------------------------------------------------------------
  Ejercicio 17. Demostrar o refutar:
     rev (borra x xs) = borra x (rev xs)
  ------------------------------------------------------------------------
*}

theorem "rev (borra x xs) = borra x (rev xs)"
quickcheck
oops

text {*
  El contraejemplo encontrado es
   x = -4
   xs = [-4, 0, -4]
  En efecto,
   rev (borra x xs) = rev (borra -4 [-4,0,-4]) = [-4,0]
   borra x (rev xs) = borra -4 (rev [-4,0,-4]) = [0,-4]
*}

text {*
```

```
  ----------------------------------------------------------------------
  Ejercicio 18. Demostrar o refutar el teorema:
     borraTodas x (xs@ys) = (borraTodas x xs)@(borraTodas x ys)
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma "borraTodas x (xs@ys) = (borraTodas x xs)@(borraTodas x ys)"
by (induct xs arbitrary: x) simp_all

-- "La demostración estructurada es"
lemma borraTodas_append:
  "borraTodas x (xs@ys) = (borraTodas x xs)@(borraTodas x ys)"
proof (induct xs arbitrary: x)
  show "x. borraTodas x ([]@ys) = (borraTodas x [])@(borraTodas x ys)" by simp
next
  fix a x xs
  assume HI: "x. borraTodas x (xs@ys) = (borraTodas x xs)@(borraTodas x ys)"
  show "borraTodas x ((a#xs)@ys) = (borraTodas x (a#xs))@(borraTodas x ys)"
  proof (cases)
    assume "x=a"
    hence "borraTodas x ((a#xs)@ys) = borraTodas x (xs@ys)" by simp
    also have " = (borraTodas x xs)@(borraTodas x ys)" using HI by simp
    also have " = (borraTodas x (a#xs))@(borraTodas x ys)" using `x=a` by simp
    finally show ?thesis .
  next
    assume "xa"
    hence "borraTodas x ((a#xs)@ys) = a#(borraTodas x (xs@ys))" by simp
    also have " = a#((borraTodas x xs)@(borraTodas x ys))" using HI by simp
    also have " = (borraTodas x (a#xs))@(borraTodas x ys)" using `xa` by simp
    finally show ?thesis .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 19. Demostrar o refutar el teorema:
     rev (borraTodas x xs) = borraTodas x (rev xs)
  ----------------------------------------------------------------------
*}
```

```
-- "La demostración automática es"
theorem "rev (borraTodas x xs) = borraTodas x (rev xs)"
by (induct xs) (auto simp add: borraTodas_append)

-- "La demostración estructurada es"
theorem "rev (borraTodas x xs) = borraTodas x (rev xs)"
proof (induct xs)
  show "rev (borraTodas x []) = borraTodas x (rev [])" by simp
next
  fix a xs
  assume HI: "rev (borraTodas x xs) = borraTodas x (rev xs)"
  show "rev (borraTodas x (a#xs)) = borraTodas x (rev (a#xs))"
  proof (cases)
    assume "x=a"
    hence "rev (borraTodas x (a#xs)) = rev (borraTodas x xs)" by simp
    also have " = borraTodas x (rev xs)" using HI by simp
    also have " = (borraTodas x (rev xs))@(borraTodas x [a])"
      using `x=a` by simp
    also have " = borraTodas x ((rev xs)@[a])"
      by (simp add: borraTodas_append)
    also have " = borraTodas x (rev (a#xs))" by simp
    finally show ?thesis .
  next
    assume "xa"
    hence "rev (borraTodas x (a#xs)) = rev (a#(borraTodas x xs))" by simp
    also have " = (rev (borraTodas x xs))@[a]" by simp
    also have " = (borraTodas x (rev xs))@[a]" using HI by simp
    also have " = (borraTodas x (rev xs))@(borraTodas x [a])"
      using `xa` by simp
    also have " = borraTodas x ((rev xs)@[a])"
      by (simp add: borraTodas_append)
    also have " = borraTodas x (rev (a#xs))" by simp
    finally show ?thesis .
  qed
qed

end
```

6.1.3. Menor posición válida

```
chapter {* T6R1c: Menor posición válida *}

theory T6R1c
imports Main
begin

section {* Menor posición válida *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     menorValida :: "('a  bool)  'a list  nat"
  tal que (menorValida p xs) es el índice del primer elemento de una
  lista xs que satisface el predicado p y es la longitud de xs si
  ningún elemento satisface el predicado p. Por ejemplo,
     menorValida (x. 4<x) [1::nat, 3, 5, 3, 1]     = 2
     menorValida (x. 6<x) [1::nat, 3, 5, 3, 1]     = 5
     menorValida (x. 1<length x) [[], [1, 2], [3]] = 1
  ----------------------------------------------------------------------
*}

fun menorValida :: "('a  bool)  'a list  nat" where
  "menorValida P []     = 0"
| "menorValida P (x#xs) = (if P x then 0 else 1+(menorValida P xs))"

value "menorValida (x. 4<x) [1::nat, 3, 5, 3, 1]"          -- "= 2"
value "menorValida (x. 6<x) [1::nat, 3, 5, 3, 1]"          -- "= 5"
value "menorValida (x. 1<length x) [[], [1, 2], [3::nat]]" -- "= 1"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar que menorValida devuelve la longitud de la
  lista syss ningún elemento satisface el predicado dado.
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma "(menorValida P xs = length xs) = list_all ( x. (¬ P x)) xs"
by (induct xs) auto
```

```
-- "La demostración estructurada es"
lemma menorValida_es_length:
  "(menorValida P xs = length xs) = list_all ( x. ň P x) xs"
proof (induct xs)
  show "(menorValida P [] = length []) = list_all ( x. ň P x) []"
    by simp
next
  fix a xs
  assume HI: "(menorValida P xs = length xs) = list_all ( x. ň P x) xs"
  show "(menorValida P (a#xs) = length (a#xs)) = list_all ( x. ň P x) (a#xs)"
  proof (cases)
    assume "P a"
    thus "(menorValida P (a#xs) = length (a#xs)) =
          list_all (x. ň P x) (a#xs)" by simp
  next
    assume "ň(P a)"
    hence "(menorValida P (a#xs) = length (a#xs)) =
           (1 + menorValida P xs = 1 + length xs)" by simp
    also have " = (menorValida P xs = length xs)" by simp
    also have " = list_all (x. ň(P x)) xs" using HI by simp
    also have " = list_all (x. ň(P x)) (a#xs)" using ‘ň(P a)‘ by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 3. Demostrar si n es el valor de (menorValida P xs),
  entonces ninguno de los primeros n elementos de la lista xs verifica
  la propiedad P.
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma "list_all (x. ň P x) (take (menorValida P xs) xs)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma "list_all (x. ň P x) (take (menorValida P xs) xs)"
```

```
proof (induct xs)
  show "list_all (x. ň P x) (take (menorValida P []) [])" by simp
next
  fix a xs
  assume HI: "list_all (x. ň P x) (take (menorValida P xs) xs)"
  show "list_all (x. ň P x) (take (menorValida P (a#xs)) (a#xs))"
  proof (cases)
    assume "P a"
    hence "menorValida P (a#xs) = 0" by simp
    hence "take (menorValida P (a#xs)) (a#xs) = []" by simp
    thus "list_all (x. ň P x) (take (menorValida P (a#xs)) (a#xs))"
      by simp
  next
    assume "ň(P a)"
    hence "menorValida P (a#xs) = 1 + menorValida P xs" by simp
    hence "take (menorValida P (a#xs)) (a#xs) =
           a#(take (menorValida P xs) xs)" by simp
    thus "list_all (x. ň P x) (take (menorValida P (a#xs)) (a#xs))"
      using HI `ň(P a)` by simp
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 4. £Cómo se puede relacionar
    "menorValida (x. P x  Q x) xs"
  con
    "menorValida P xs" y "menorValida Q xs"?
  £Se puede decir algo parecido con la conjunción de P y Q?
  Prueba tus conjeturas.
  ---------------------------------------------------------------------
*}

text {*
  La relación es la siguiente: La menor posición de los elementos que
  verifican la propiedad "P  Q" es el mínimo de la menor posición de
  los elementos que verifican P y de la menor posición de los elementos
  que verifican Q.
*}
```

```
-- "La demostración automática es"
lemma menorValida_diyuncion_auto:
  "menorValida (x. P x  Q x) xs =
   min (menorValida P xs) (menorValida Q xs)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma menorValida_diyuncion:
  "menorValida ( x. P x  Q x) xs =
   min (menorValida P xs) (menorValida Q xs)"
proof (induct xs)
  show "menorValida ( x. P x  Q x) [] =
    min (menorValida P []) (menorValida Q [])" by simp
next
  fix a xs
  assume HI: "menorValida ( x. P x  Q x) xs =
              min (menorValida P xs) (menorValida Q xs)"
  show "menorValida ( x. P x  Q x) (a#xs) =
        min (menorValida P (a#xs)) (menorValida Q (a#xs))"
  proof (cases)
    assume "P a"
    hence "menorValida ( x. P x  Q x) (a#xs) = 0" by simp
    also have " = min 0 (menorValida Q (a#xs))" by simp
    also have " = min (menorValida P (a#xs)) (menorValida Q (a#xs))"
      using ‘P a‘ by simp
    finally show ?thesis .
  next
    assume "ň P a"
    show "menorValida ( x. P x  Q x) (a#xs) =
      min (menorValida P (a#xs)) (menorValida Q (a#xs))"
    proof (cases)
      assume "Q a"
      hence "menorValida ( x. P x  Q x) (a#xs) = 0" by simp
      also have " = min (menorValida P (a#xs)) 0" by simp
      also have " = min (menorValida P (a#xs)) (menorValida Q (a#xs))"
        using ‘Q a‘ by simp
      finally show ?thesis .
    next
      assume "ň Q a"
      hence "menorValida ( x. P x  Q x) (a#xs) =
```

```
            1 + (menorValida ( x. P x  Q x) xs)" using ‘ňP a‘ by simp
      also have " = 1+min (menorValida P xs) (menorValida Q xs)"
        using HI by simp
      also have " = min (menorValida P (a#xs)) (menorValida Q (a#xs))"
        using ‘ň P a‘ ‘ň Q a‘ by simp
      finally show ?thesis .
    qed
  qed
qed

text {*
  La menor posición de los elementos que verifican la propiedad "P  Q"
  es mayor o igual que el máximo de la menor posición de los elementos
  que verifican P y de la menor posición de los elementos que verifican Q.
*}

-- "La demostración automática es"
lemma menorValida_conjuncion_auto:
  "max (menorValida P xs) (menorValida Q xs)  menorValida ( x. P x  Q x) xs"
by (induct xs) auto

-- "La demostración estructurada es"
lemma menorValida_conjuncion:
  "max (menorValida P xs) (menorValida Q xs)  menorValida ( x. P x  Q x) xs"
proof (induct xs)
  show "max (menorValida P []) (menorValida Q [])
        menorValida ( x. P x  Q x) []" by simp
next
  fix a xs
  assume HI: "max (menorValida P xs) (menorValida Q xs)
    menorValida ( x. P x  Q x) xs"
  show "max (menorValida P (a#xs)) (menorValida Q (a#xs))
        menorValida ( x. P x  Q x) (a#xs)"
  proof (cases)
    assume "P a"
    show "max (menorValida P (a#xs)) (menorValida Q (a#xs))
          menorValida ( x. P x  Q x) (a#xs)"
    proof -
      have "menorValida P (a#xs) = 0" using ‘P a‘ by simp
      hence f1: "max (menorValida P (a#xs)) (menorValida Q (a#xs)) =
```

```
                menorValida Q (a#xs)" by simp
    have "menorValida Q (a#xs)  menorValida ( x. P x  Q x) (a#xs)"
    proof (cases)
      assume "Q a"
      hence "menorValida Q (a#xs) = 0" by simp
      also have "  menorValida ( x. P x  Q x) (a#xs)" by simp
      finally show "menorValida Q (a#xs)
                    menorValida ( x. P x  Q x) (a#xs)" .
    next
      assume "ň Q a"
      hence "menorValida Q (a#xs) = 1+(menorValida Q xs)" by simp
      also have "  1+(max (menorValida P xs) (menorValida Q xs))" by simp
      also have "  1+(menorValida ( x. P x  Q x) xs)" using HI by simp
      also have " = menorValida ( x. P x  Q x) (a#xs)"
        using ‘ň Q a‘ by simp
      finally show "menorValida Q (a#xs)
                    menorValida ( x. P x  Q x) (a#xs)" .
    qed
    thus "max (menorValida P (a#xs)) (menorValida Q (a#xs))
          menorValida ( x. P x  Q x) (a#xs)" using f1 by simp
  qed
next
  assume "ň P a"
  show "max (menorValida P (a#xs)) (menorValida Q (a#xs))
        menorValida ( x. P x  Q x) (a#xs)"
  proof (cases)
    assume "Q a"
    hence "menorValida Q (a#xs) = 0" by simp
    hence "max (menorValida P (a#xs)) (menorValida Q (a#xs)) =
           menorValida P (a#xs)" by simp
    also have " = 1+(menorValida P xs)" using ‘ň P a‘ by simp
    also have "  1+(max (menorValida P xs) (menorValida Q xs))" by simp
    also have "  1+(menorValida ( x. P x  Q x) xs)" using HI by simp
    also have " = menorValida ( x. P x  Q x) (a#xs)"
      using ‘ň P a‘ by simp
    finally show "max (menorValida P (a#xs)) (menorValida Q (a#xs))
                  menorValida ( x. P x  Q x) (a#xs)" .
  next
    assume "ň Q a"
    hence "max (menorValida P (a#xs)) (menorValida Q (a#xs)) =
```

```
            max (1+(menorValida P xs)) (1+(menorValida Q xs))"
        using ‘ñ P a‘ by simp
      also have " = 1+(max (menorValida P xs) (menorValida Q xs))" by simp
      also have "  1+(menorValida ( x. P x  Q x) xs)" using HI by simp
      also have " = menorValida ( x. P x  Q x) (a#xs)"
        using ‘ñ P a‘ by simp
      finally show "max (menorValida P (a#xs)) (menorValida Q (a#xs))
                    menorValida ( x. P x  Q x) (a#xs)" .
    qed
  qed
qed

text {*
  La desigualdad complementaria no es válida.
*}

lemma
  "menorValida ( x. P x  Q x) xs  max (menorValida P xs) (menorValida Q xs)"
quickcheck
oops

text {*
  El contraejemplo encontrado es
     P = {a\<^isub>1}
     Q = {a\<^isub>2}
     xs = [a\<^isub>1, a\<^isub>2]
*}

text {*
  ---------------------------------------------------------------------
  Ejercicio 5.  Si P implica Q, £qué relación puede deducirse entre
  "menorValida P xs" y "menorValida Q xs"?
  ---------------------------------------------------------------------
*}

text {*
  Se deduce que
     "menorValida Q xs  menorValida P xs".
*}
```

```
-- "La demostración automática es"
lemma menorValida_si_implica_auto:
  "(x. P x  Q x)  menorValida Q xs  menorValida P xs"
by (induct xs) auto

-- "La demostración estructurada es"
lemma menorValida_si_implica:
  assumes "(x. P x  Q x)"
  shows "menorValida Q xs  menorValida P xs"
proof (induct xs)
  show "menorValida Q []  menorValida P []" by simp
next
  fix a xs
  assume HI: "menorValida Q xs  menorValida P xs"
  show "menorValida Q (a#xs)  menorValida P (a#xs)"
  proof (cases)
    assume "Q a"
    hence "menorValida Q (a#xs) = 0" by simp
    also have "  menorValida P (a#xs)" by simp
    finally show ?thesis .
  next
    assume "¬ Q a"
    hence "¬ P a" using assms by blast
    have "menorValida Q (a#xs) = 1+(menorValida Q xs)"
      using ‘¬ Q a‘ by simp
    also have "  1+(menorValida P xs)" using HI by simp
    also have " = menorValida P (a#xs)" using ‘¬ P a‘ by simp
    finally show ?thesis .
  qed
qed

end
```

6.1.4. Número de elementos válidos

```
chapter {* T6R1d: Número de elementos válidos *}

theory T6R1d
imports Main
begin
```

```
section {* Número de elementos válidos *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     cuentaP :: "('a  bool)  'a list  nat"
  tal que (cuentaP Q xs) es el número de elementos de la lista xs que
  satisfacen el predicado Q. Por ejemplo,
     cuentaP (x. 2<x) [1,3,4,0,5] = 3
  ----------------------------------------------------------------------
*}

fun cuentaP :: "('a  bool)  'a list  nat" where
  "cuentaP Q []     = 0"
| "cuentaP Q (x#xs) = (if Q x then (1 + cuentaP Q xs) else cuentaP Q xs)"

value "cuentaP (x. 2<x) [1::nat,3,4,0,5]" -- "= 3"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar o refutar:
     cuentaP P (xs @ ys) = cuentaP P xs + cuentaP P ys*
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma cuentaP_append_auto:
  "cuentaP P (xs @ ys) = cuentaP P xs + cuentaP P ys"
by (induct xs) auto

-- "La demostración estructurada es"
lemma cuentaP_append:
  "cuentaP P (xs @ ys) = cuentaP P xs + cuentaP P ys"
proof (induct xs)
  show "cuentaP P ([] @ ys) = cuentaP P [] + cuentaP P ys" by simp
next
  fix a xs
  assume HI: "cuentaP P (xs @ ys) = cuentaP P xs + cuentaP P ys"
  show "cuentaP P ((a#xs) @ ys) = cuentaP P (a#xs) + cuentaP P ys"
  proof (cases)
```

```
    assume "P a"
    hence "cuentaP P ((a#xs) @ ys) = 1 + cuentaP P (xs @ ys)" by simp
    also have " = 1 + cuentaP P xs + cuentaP P ys" using HI by simp
    also have " = cuentaP P (a#xs) + cuentaP P ys" using ‘P a‘ by simp
    finally show ?thesis .
  next
    assume "ň P a"
    hence "cuentaP P ((a#xs) @ ys) = cuentaP P (xs @ ys)" by simp
    also have " = cuentaP P xs + cuentaP P ys" using HI by simp
    also have " = cuentaP P (a#xs) + cuentaP P ys" using ‘ň P a‘ by simp
    finally show ?thesis .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Demostrar que el número de elementos de una lista que
  cumplen una determinada propiedad es el mismo que el de esa lista
  invertida.
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma cuentaP_rev_auto:
  "cuentaP P xs = cuentaP P (rev xs)"
by (induct xs) (simp_all add: cuentaP_append)

-- "La demostración estructurada es"
lemma cuentaP_rev:
  "cuentaP P xs = cuentaP P (rev xs)"
proof (induct xs)
  show "cuentaP P [] = cuentaP P (rev [])" by simp
next
  fix a xs
  assume HI: "cuentaP P xs = cuentaP P (rev xs)"
  show "cuentaP P (a#xs) = cuentaP P (rev (a#xs))"
  proof -
    have "cuentaP P (a#xs) = cuentaP P xs + cuentaP P [a]" by simp
    also have " = cuentaP P (rev xs) + cuentaP P [a]" using HI by simp
    also have " = cuentaP P ((rev xs) @ [a])" by (simp add: cuentaP_append)
```

```
    also have " = cuentaP P (rev (a#xs))" by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 4. Encontrar y demostrar una relación entre las funciones
  filter y  cuentaP.
  ---------------------------------------------------------------------
*}

text {*
  Solución: La relación existente es
     length (filter P xs) = cuentaP P xs
*}

-- "La demostración automática es"
lemma length_filter:
  "length (filter P xs) = cuentaP P xs"
by (induct xs) simp_all

-- "La demostración estructurada es"
lemma length_filter_2:
  "length (filter P xs) = cuentaP P xs"
proof (induct xs)
  show "length (filter P []) = cuentaP P []" by simp
next
  fix a xs
  assume HI: "length (filter P xs) = cuentaP P xs"
  show "length (filter P (a#xs)) = cuentaP P (a#xs)"
  proof (cases)
    assume "P a"
    hence "length (filter P (a#xs)) = length (a#(filter P xs))" by simp
    also have " = 1 + length (filter P xs)" by simp
    also have " = 1 + cuentaP P xs" using HI by simp
    also have " = cuentaP P (a#xs)" using ‘P a‘ by simp
    finally show ?thesis .
  next
    assume "ň P a"
```

```
    hence "length (filter P (a#xs)) = length (filter P xs)" by simp
    also have " = cuentaP P xs" using HI by simp
    also have " = cuentaP P (a#xs)" using ‘ň P a‘ by simp
    finally show ?thesis .
  qed
qed

end
```

6.1.5. Contador de occurrencias

```
chapter {* T6R1e: Contador de occurrencias *}

theory T6R1e
imports Main
begin

section {* Contador de ocurrencias *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     veces :: "'a  'a list  nat"
  tal que (veces x ys) es el número de occurrencias del elemento x en la
  lista ys. Por ejemplo,
    veces (2::nat) [2,1,2,5,2] = 3
  ----------------------------------------------------------------------
*}

fun veces :: "'a  'a list  nat" where
  "veces x []     = 0"
| "veces x (y#xs) = (if x=y then (1+(veces x xs)) else (veces x xs))"

value "veces (2::nat) [2,1,2,5,2]" -- "= 3"


text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar o refutar:
     veces a (xs @ ys) = veces a xs + veces a ys
  ----------------------------------------------------------------------
```

```
*}

-- "La demostración automática es"
lemma veces_append [simp]:
  "veces a (xs @ ys) = veces a xs + veces a ys"
by (induct xs) auto

-- "La demostración estructurada es"
lemma veces_append_2:
  "veces a (xs @ ys) = veces a xs + veces a ys"
proof (induct xs)
  show "veces a ([] @ ys) = veces a [] + veces a ys" by simp
next
  fix b xs
  assume HI: "veces a (xs @ ys) = veces a xs + veces a ys"
  show "veces a ((b#xs) @ ys) = veces a (b#xs) + veces a ys"
  proof (cases "a=b")
    assume "a=b"
    have "veces a ((b#xs) @ ys) = veces a (b#(xs@ys))" by simp
    also have " = 1 + veces a (xs@ys)" using `a=b` by simp
    also have " = 1 + veces a xs + veces a ys" using HI by simp
    also have " = veces a (b#xs) + veces a ys" using `a=b` by simp
    finally show ?thesis .
  next
    assume "ab"
    have "veces a ((b#xs) @ ys) = veces a (b#(xs@ys))" by simp
    also have " = veces a (xs@ys)" using `ab` by simp
    also have " = veces a xs + veces a ys" using HI by simp
    also have " = veces a (b#xs) + veces a ys" using `ab` by simp
    finally show ?thesis .
  qed
qed

text {*
  ------------------------------------------------------------------
  Ejercicio 3. Demostrar o refutar:
     veces a xs = veces a (rev xs)
  ------------------------------------------------------------------
*}
```

```
-- "La demostración automática es"
lemma veces_rev:
  "veces a xs = veces a (rev xs)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma veces_rev_2:
  "veces a xs = veces a (rev xs)"
proof (induct xs)
  show "veces a [] = veces a (rev [])" by simp
next
  fix b xs
  assume HI: "veces a xs = veces a (rev xs)"
  show "veces a (b#xs) = veces a (rev (b#xs))"
  proof -
    have "veces a (b#xs) = veces a [b] + veces a xs" by simp
    also have " = veces a (rev xs) + veces a [b]" using HI by simp
    also have " = veces a ((rev xs) @ [b])" by simp
    also have " = veces a (rev (b#xs))" by simp
    finally show ?thesis .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Demostrar o refutar:
    "veces a xs  length xs".
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma veces_le_length_auto:
  "veces a xs  length xs"
by (induct xs) auto

-- "La demostración estructurada es"
lemma veces_le_length:
  "veces a xs  length xs"
proof (induct xs)
  show "veces a []  length []" by simp
```

```
next
  fix b xs
  assume HI: "veces a xs  length xs"
  show "veces a (b#xs)  length (b#xs)"
  proof -
    have "veces a (b#xs)  1 + veces a xs" by simp
    also have "  1 + length xs" using HI by simp
    also have " = length (b#xs)" by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 5. Sabiendo que la función map aplica una función a todos
  los elementos de una lista:
     map f [x\<^isub>1,,x\<^isub>n] = [f x\<^isub>1,,f x\<^isub>n],
  demostrar o refutar
     veces a (map f xs) = veces (f a) xs
  ---------------------------------------------------------------------
*}

-- "La búsqueda de un contraejemplo con quickcheck es"
lemma veces_map:
  "veces a (map f xs) = veces (f a) xs"
quickcheck
oops

text {*
  El contraejemplo encontrado es
     a  = a\<^isub>2
     f  = (_. _)(a\<^isub>1 := a\<^isub>1, a\<^isub>2 := a\<^isub>1)
     xs = [a\<^isub>1]
  En efecto,
     veces a (map f xs) = veces a\<^isub>2 (map f [a\<^isub>1]) = veces a\<^isub>2 [a
     veces (f a\<^isub>2) [a\<^isub>1]  = veces (f a\<^isub>2) [a\<^isub>1]     = vec
*}

text {*
  ---------------------------------------------------------------------
```

```
  Ejercicio 6. La función
     filter :: "('a  bool)  'a list  'a list"
  está definida por
     filter P []       = []
     filter P (x # xs) = (if P x then x # filter P xs else filter P xs)
  Encontrar una expresión e que no contenga filter tal que se verifique
  la siguiente propiedad:
     veces a (filter P xs) = e
  ----------------------------------------------------------------------
*}

text {*
  Solución: "La expresión es
     (if (P a) then veces a xs else 0)
  como se prueba a continuación.
*}

-- "La demostración automática es"
lemma veces_filter:
  "veces a (filter P xs) = (if (P a) then veces a xs else 0)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma veces_filter_2:
  "veces a (filter P xs) = (if (P a) then veces a xs else 0)"
proof (induct xs)
  show "veces a (filter P []) = (if (P a) then veces a [] else 0)" by simp
next
  fix b xs
  assume HI: "veces a (filter P xs) = (if (P a) then veces a xs else 0)"
  show "veces a (filter P (b#xs)) = (if (P a) then veces a (b#xs) else 0)"
  proof (cases)
    assume "P b"
    hence "veces a (filter P (b#xs)) = veces a (b#(filter P xs))" by simp
    also have " = veces a ([b]@(filter P xs))" by simp
    also have " = (veces a [b]) + (veces a (filter P xs))" by simp
    also have " = (veces a [b]) + (if (P a) then veces a xs else 0)"
      using HI by simp
    also have " = (if (P a) then ((veces a [b])+(veces a xs)) else 0)"
      using `P b` by simp
```

```
    also have " = (if (P a) then (veces a ([b]@xs)) else 0)" by simp
    also have " = (if (P a) then (veces a (b#xs)) else 0)" by simp
    finally show ?thesis .
  next
    assume "¬ P b"
    hence "veces a (filter P (b#xs)) = veces a (filter P xs)" by simp
    also have " = (if (P a) then veces a xs else 0)" using HI by simp
    also have " = (if (P a) then (veces a (b#xs)) else 0)"
      using ‘¬ P b‘ by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 7. Usando veces, definir la función
     borraDups :: "'a list ⇒ bool"
  tal que (borraDups xs) es la lista obtenida eliminando los elementos
  duplicados de la lista xs. Por ejemplo,
     borraDups [1::nat,2,4,2,3] = [1,4,2,3]

  Nota: La función borraDups es equivalente a la predefinida remdups.
  ---------------------------------------------------------------------
*}

fun borraDups :: "'a list ⇒ 'a list" where
  "borraDups []     = []"
| "borraDups (x#xs) = (if 0<(veces x xs) then (borraDups xs)
                         else (x#borraDups xs))"

value "borraDups [1::nat,2,4,2,3]" -- "= [1,4,2,3]"

text {*
  ---------------------------------------------------------------------
  Ejercicio 8. Encontrar una expresión e que no contenga la función
  borraDups tal que se verifique
     veces x (borraDups xs) = e
  ---------------------------------------------------------------------
*}
```

```
text {*
  Solución: La expresión es
     (if 0<(veces x xs) then 1 else 0)
  comp se prueba a continuación.
*}

-- "La demostración automática es"
lemma veces_borraDups:
  "veces x (borraDups xs) = (if 0<veces x xs then 1 else 0)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma veces_borraDups_2:
  "veces x (borraDups xs) = (if 0<(veces x xs) then 1 else 0)"
proof (induct xs)
  show "veces x (borraDups []) = (if (0<veces x []) then 1 else 0)" by simp
next
  fix a xs
  assume HI: "veces x (borraDups xs) = (if (0<veces x xs) then 1 else 0)"
  show "veces x (borraDups (a#xs)) = (if (0<veces x (a#xs)) then 1 else 0)"
  proof (cases)
    assume c1: "0<(veces a xs)"
    hence "veces x (borraDups (a#xs)) = veces x (borraDups xs)" by simp
    also have " = (if (0<veces x xs) then 1 else 0)" using HI by simp
    also have " = (if (0<veces x (a#xs)) then 1 else 0)" using c1 by simp
    finally show ?thesis .
  next
    assume c2: "ň 0<(veces a xs)"
    hence "veces x (borraDups (a#xs)) = veces x (a#(borraDups xs))" by simp
    also have " = (veces x [a]) + (veces x (borraDups xs))" by simp
    also have " = (veces x [a]) + (if (0<veces x xs) then 1 else 0)"
      using HI by simp
    also have " = (if (0<veces x (a#xs)) then 1 else 0)" using c2 by simp
    finally show ?thesis .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Usando la función "veces", definir la función
```

```
    distintos :: "'a list  bool"
  tal que (distintos xs) se verifica si cada elemento de xs aparece tan
  solo una vez. Por ejemplo,
    distintos [1,4,3]
    ň distintos [1,4,1]

  Nota: La función "distintos" es equivalente a la predefinida "distinct".
  ----------------------------------------------------------------------
*}

fun distintos :: "'a list  bool" where
  "distintos []     = True"
| "distintos (x#xs) = ((veces x xs = 0)  distintos xs)"

value "distintos [1::nat,4,3]"   -- "= True"
value "ň distintos [1::nat,4,1]" -- "= True"

fun distintos' :: "'a list  bool" where
  "distintos' []     = True"
| "distintos' (x#xs) = (ň (elem x xs)  distintos' xs)"

value "distintos' [1::nat,4,3]"   -- "= True"
value "ň distintos' [1::nat,4,1]" -- "= True"

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Demostrar que el valor de "borraDups" verifica "distintos".
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma distintos_borraDups:
  "distintos (borraDups xs)"
by (induct xs) (auto simp add: veces_borraDups)

-- "La demostración estructurada es"
lemma distintos_borraDups_2:
  "distintos (borraDups xs)"
proof (induct xs)
  show "distintos (borraDups [])" by simp
```

```
next
  fix a xs
  assume HI: "distintos (borraDups xs)"
  show "distintos (borraDups (a#xs))"
  proof (cases)
    assume "0<veces a xs"
    hence "distintos (borraDups (a#xs)) = distintos (borraDups xs)" by simp
    thus ?thesis using HI by simp
  next
    assume "¬ 0<veces a xs"
    hence "veces a (borraDups xs) = 0" by (simp add:veces_borraDups)
    hence "(veces a (borraDups xs) = 0)  (distintos (borraDups xs))"
      using HI by simp
    hence "distintos (a#borraDups xs)" using ‘¬ 0<veces a xs‘ by simp
    thus "distintos (borraDups (a#xs))" using ‘¬ 0<veces a xs‘ by simp
  qed
qed

end
```

6.1.6. Suma y aplanamiento de listas

```
chapter {* T6R1f: Suma y aplanamiento de listas *}

theory T6R1f
imports Main T6R1_1-Cons_inverso_y_cuantificadores_sobre_listas
begin

section {* Suma y aplanamiento de listas *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     suma :: "nat list  nat"
  tal que (suma xs) es la suma de los elementos de la lista de números
  naturales xs. Por ejemplo,
     suma [3::nat,2,4] = 9
  ----------------------------------------------------------------------
*}

fun suma :: "nat list  nat" where
```

```
  "suma []     = 0"
| "suma (x#xs) = x + suma xs"

value "suma [3::nat,2,4]" -- "= 9"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Definir la función
     aplana :: "'a list list  'a list"
  tal que (aplana xss) es la obtenida concatenando los miembros de la
  lista de listas "xss". Por ejemplo,
     aplana [[2,3], [4,5], [7,9]] = [2,3,4,5,7,9]
  ----------------------------------------------------------------------
*}

fun aplana :: "'a list list  'a list" where
  "aplana []     = []"
| "aplana (x#xs) = x @ aplana xs"

value "aplana [[2::nat,3], [4,5], [7,9]]" -- "= [2,3,4,5,7,9]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Demostrar o refutar
     length (aplana xs) = suma (map length xs)
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma length_aplana:
  "length (aplana xs) = suma (map length xs)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma length_aplana_2:
  "length (aplana xs) = suma (map length xs)"
proof (induct xs)
  show "length (aplana []) = suma (map length [])" by simp
next
  fix a xs
```

```
  assume HI: "length (aplana xs) = suma (map length xs)"
  show "length (aplana (a#xs)) = suma (map length (a#xs))"
  proof -
    have "length (aplana (a#xs)) = length (a @ (aplana xs))" by simp
    also have " = length a + length (aplana xs)" by simp
    also have " = length a + suma (map length xs)" using HI by simp
    also have " = suma (map length (a#xs))" by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 4. Demostrar o refutar
     suma (xs @ ys) = suma xs + suma ys
  ---------------------------------------------------------------------
*}

-- "La demostración automática es:"
lemma suma_append:
  "suma (xs @ ys) = suma xs + suma ys"
by (induct xs) auto

-- "La demostración estructurada es"
lemma suma_append_2:
  "suma (xs @ ys) = suma xs + suma ys"
proof (induct xs)
  show "suma ([] @ ys) = suma [] + suma ys" by simp
next
  fix a xs
  assume HI: "suma (xs @ ys) = suma xs + suma ys"
  show "suma ((a#xs) @ ys) = suma (a#xs) + suma ys"
  proof -
    have "suma ((a#xs) @ ys) = suma (a#(xs@ys))" by simp
    also have " = a + suma (xs@ys)" by simp
    also have " = a + suma xs + suma ys" using HI by simp
    also have " = suma (a#xs) + suma ys" by simp
    finally show ?thesis .
  qed
qed
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Demostrar o refutar
     aplana (xs @ ys) = (aplana xs) @ (aplana ys)
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma aplana_append:
  "aplana (xs @ ys) = (aplana xs) @ (aplana ys)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma aplana_append_2:
  "aplana (xs @ ys) = (aplana xs) @ (aplana ys)"
proof (induct xs)
  show "aplana ([] @ ys) = (aplana []) @ (aplana ys)" by simp
next
  fix a xs
  assume HI: "aplana (xs @ ys) = (aplana xs) @ (aplana ys)"
  show "aplana ((a#xs) @ ys) = (aplana (a#xs)) @ (aplana ys)"
  proof -
    have "aplana ((a#xs) @ ys) = aplana (a#(xs@ys))" by simp
    also have " = a @ (aplana (xs@ys))" by simp
    also have " = a @ aplana xs @ aplana ys" using HI by simp
    also have " = aplana (a#xs) @ aplana ys" by simp
    finally show ?thesis .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 6. Demostrar o refutar
     aplana (map rev (rev xs)) = rev (aplana xs)
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma aplana_map_rev_rev:
```

```
  "aplana (map rev (rev xs)) = rev (aplana xs)"
by (induct xs) (auto simp add: aplana_append)

-- "La demostración estructurada es"
lemma aplana_map_rev_rev_2:
  "aplana (map rev (rev xs)) = rev (aplana xs)"
proof (induct xs)
  show "aplana (map rev (rev [])) = rev (aplana [])" by simp
next
  fix a xs
  assume HI: "aplana (map rev (rev xs)) = rev (aplana xs)"
  show "aplana (map rev (rev (a#xs))) = rev (aplana (a#xs))"
  proof -
    have "aplana (map rev (rev (a#xs))) = aplana (map rev ((rev xs)@[a]))"
      by simp
    also have " = aplana ((map rev (rev xs))@(map rev [a]))" by simp
    also have " = (aplana (map rev (rev xs)))@(aplana (map rev [a]))"
      by (simp add: aplana_append)
    also have " = (rev (aplana xs))@(aplana (map rev [a]))" using HI by simp
    also have " = (rev (aplana xs))@(rev (aplana [a]))" by simp
    also have " = rev ((aplana [a])@(aplana xs))" by simp
    also have " = rev (aplana (a#xs))" by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 7. Demostrar o refutar
     aplana (rev (map rev xs)) = rev (aplana xs)
  ---------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma aplana_rev_map_rev:
  "aplana (rev (map rev xs)) = rev (aplana xs)"
by (induct xs) (auto simp add: aplana_append)

text {*
  ---------------------------------------------------------------------
```

```
  Ejercicio 8. Demostrar o refutar
     list_all (list_all P) xs = list_all P (aplana xs)
  ------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma list_all_list_all:
  "list_all (list_all P) xs = list_all P (aplana xs)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma list_all_list_all_2:
  "list_all (list_all P) xs = list_all P (aplana xs)"
proof (induct xs)
  show "list_all (list_all P) [] = list_all P (aplana [])" by simp
next
  fix a xs
  assume HI: "list_all (list_all P) xs = list_all P (aplana xs)"
  show "list_all (list_all P) (a#xs) = list_all P (aplana (a#xs))"
  proof -
    have "list_all (list_all P) (a#xs) =
          ((list_all P a)  (list_all (list_all P) xs))" by simp
    also have " = ((list_all P a)  (list_all P (aplana xs)))"
      using HI by simp
    also have " = list_all P (a@(aplana xs))" by simp
    also have " = list_all P (aplana (a#xs))" by simp
    finally show ?thesis .
  qed
qed

text {*
  ------------------------------------------------------------------
  Ejercicio 9. Demostrar o refutar
     aplana (rev xs) = aplana xs
  ------------------------------------------------------------------
*}

-- "La búsqueda de un contraejemplo con QuickCheck es"
lemma aplana_rev:
  "aplana (rev xs) = aplana xs"
```

```
quickcheck
oops

text {*
  El contraejemplo encontrado es
     xs = [[a\<^isub>1], [a\<^isub>2]]
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Demostrar o refutar
     suma (rev xs) = suma xs
  ----------------------------------------------------------------------
*}

-- "La demostración automática es"
lemma suma_rev:
  "suma (rev xs) = suma xs"
by (induct xs) (auto simp add: suma_append)

-- "La demostración estructurada es"
lemma suma_rev_2:
  "suma (rev xs) = suma xs"
proof (induct xs)
  show "suma (rev []) = suma []" by simp
next
  fix a xs
  assume HI: "suma (rev xs) = suma xs"
  show "suma (rev (a#xs)) = suma (a#xs)"
  proof -
    have "suma (rev (a#xs)) = suma ((rev xs)@[a])" by simp
    also have " = (suma (rev xs)) + (suma [a])" by (simp add: suma_append)
    also have " = (suma xs) + (suma [a])" using HI by simp
    also have " = a + (suma xs)" by simp
    also have " = suma (a#xs)" by simp
    finally show ?thesis .
 qed
qed

text {*
```

```
  ----------------------------------------------------------------------
  Ejercicio 11. Buscar un predicado P para que se verifique la siguiente
  propiedad
     list_all P xs ⟷ length xs ≤ suma xs
  ----------------------------------------------------------------------
*}

text {*
  Solución: El predicado es "(x. 1 ≤ x)", como se demuestra a
  continuación
*}

-- "La demostración automática es"
lemma "list_all (x. 1 ≤ x) xs ⟷ length xs ≤ suma xs"
by (induct xs) auto

-- "La demostración estructurada es"
lemma "list_all (x. 1 ≤ x) xs ⟷ length xs ≤ suma xs"
proof (induct xs)
  show "list_all (x. 1 ≤ x) [] ⟷ length [] ≤ suma []" by simp
next
  fix a xs
  assume HI: "list_all (x. 1 ≤ x) xs ⟷ length xs ≤ suma xs"
  show "list_all (x. 1 ≤ x) (a#xs) ⟷ length (a#xs) ≤ suma (a#xs)"
  proof
    assume c1: "list_all (x. 1 ≤ x) (a#xs)"
    hence c2: "1 ≤ a" by simp
    have c3: "list_all (x. 1 ≤ x) xs" using c1 by simp
    have "length (a#xs) ≤ a + (length xs)" using c2 by simp
    also have " ≤ a + (suma xs)" using HI c3 by simp
    also have " = suma (a#xs)" by simp
    finally show "length (a#xs) ≤ suma (a#xs)" .
  qed
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Demostrar o refutar
     algunos (algunos P) xs = algunos P (aplana xs)
  ----------------------------------------------------------------------
```

```
*}

-- "La demostración automática es"
lemma algunos_algunos:
  "algunos (algunos P) xs = algunos P (aplana xs)"
by (induct xs) (auto simp add: algunos_append)

-- "La demostración estructurada es"
lemma algunos_algunos_2:
  "algunos (algunos P) xs = algunos P (aplana xs)"
proof (induct xs)
  show "algunos (algunos P) [] = algunos P (aplana [])" by simp
next
  fix a xs
  assume HI: "algunos (algunos P) xs = algunos P (aplana xs)"
  show "algunos (algunos P) (a#xs) = algunos P (aplana (a#xs))"
  proof -
    have "algunos (algunos P) (a#xs) =
          ((algunos P a)  (algunos (algunos P) xs))" by simp
    also have " = ((algunos P a)  (algunos P (aplana xs)))" using HI by simp
    also have " = algunos P (a @ (aplana xs))" by (simp add: algunos_append)
    also have " = algunos P (aplana (a#xs))" by simp
    finally show ?thesis .
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 13. Redefinir, usando la función list_all, la función
  algunos. Llamar la nueva función algunos2 y demostrar que es
  equivalente a algunos.
  ---------------------------------------------------------------------
*}

fun algunos2 :: "('a  bool)  ('a list  bool)" where
  "algunos2 P xs = (ň list_all (x. ň P x) xs)"

-- "La demostración automática es"
lemma algunos2_algunos:
  "algunos2 P xs = algunos P xs"
```

```
by (induct xs) auto

-- "La demostración estructurada es"
lemma algunos2_algunos_2:
  "algunos2 P xs = algunos P xs"
proof (induct xs)
  show "algunos2 P [] = algunos P []" by simp
next
  fix a xs
  assume HI: "algunos2 P xs = algunos P xs"
  show "algunos2 P (a#xs) = algunos P (a#xs)"
  proof -
    have "algunos2 P (a#xs) = (ň list_all (x. ň P x) (a#xs))"
      by simp
    also have " = (ň ((ň P a)  (list_all (x. ň P x) xs)))" by simp
    also have " = ((P a)  ň (list_all (x. ň P x) xs))" by simp
    also have " = ((P a)  (algunos2 P xs))" by simp
    also have " = ((P a)  (algunos P xs))" using HI by simp
    also have " = algunos P (a#xs)" by simp
    finally show ?thesis .
  qed
qed

end
```

6.1.7. Conjuntos mediante listas

```
chapter {* T6R1g: Conjuntos mediante listas *}

theory T6R1g
imports Main
begin

text {*
  Los conjuntos finitos se pueden representar mediante listas. En esta
  relación se define la unión y se demostran algunas de sus
  propiedades. Análogamente se puede hacer con la intersección y
  diferencia. *}

text {*
  ----------------------------------------------------------------------
```

```
  Ejercicio 1. Definir la función
     union_l :: "'a list  'a list  'a list"
  tal que (union_l xs ys) es la unión de las listas xs e ys. Por
  ejemplo,
     union_l [1,2] [2,3] = [1,2,3]
  ------------------------------------------------------------------
*}

fun union_l :: "'a list  'a list  'a list" where
  "union_l [] ys     = ys"
| "union_l (x#xs) ys = (if x  set ys
                          then (union_l xs ys)
                          else x#(union_l xs ys))"

value "union_l [1::nat,2] [2,3]" -- "= [1,2,3]"

text {*
  ------------------------------------------------------------------
  Ejercicio 2. Demostrar o refutar
     set (union_l xs ys) = set xs  set ys
  ------------------------------------------------------------------
*}

lemma set_union_l: "set (union_l xs ys) = set xs  set ys"
by (induct "xs") auto

text {*
  ------------------------------------------------------------------
  Ejercicio 3. Demostrar que si xs e ys no tienen elementos repetidos,
  entonces (union_l xs ys) tampoco los tiene.

  Indicación: Usar la función "distintc" de la teoría List.thy
  ------------------------------------------------------------------
*}

lemma [rule_format]:
  "distinct xs  distinct ys  (distinct (union_l xs ys))"
by (induct "xs") (auto simp add: set_union_l)

section {* Cuantificación sobre conjuntos *}
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Definir un conjunto S para que se verifique que
     (∀xA. P x) ⟷ (∀x ∈ B. P x) ⟷ (∀x∈S. P x)
  ----------------------------------------------------------------------
*}

text {* S = A ∪ B *}
lemma "(∀xA. P x) ⟷ (∀x ∈ B. P x) ⟷ (∀xAB. P x)"
by auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Definir una propiedad P para que se verifique que
     x ∈ A. P (f x) ⟷ y ∈ f ‘ A. Q y
  ----------------------------------------------------------------------
*}

text {* Q = P *}
lemma "∀xA. Q (f x) ⟷ ∀yf ‘A. Q y"
by auto

end
```

6.1.8. Suma de listas y recursión final

```
chapter {* T6R1h: Suma de listas y recursión final *}

theory T6R1h
imports Main
begin

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     suma :: "nat list ⇒ nat
  tal que (suma xs) es la suma de los elementos de xs. Por ejemplo,
     suma [2,3,5] = 10
  ----------------------------------------------------------------------
*}
```

```
fun suma :: "nat list  nat" where
  "suma []     = 0"
| "suma (x#xs) = x + suma xs"

value "suma [2,3,5]" -- "= 10"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar que la suma de la concatenación de dos listas
  es la suma de las sumas de cada lista.
  ----------------------------------------------------------------------
*}

lemma suma_append [simp]:
  "suma (xs @ ys) = suma xs + suma ys"
by (induct xs) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Demostrar que la suma de los n primeros números naturales
  es n*(n+1)/2; es decir,
     0 + 1 + 2 + 3 + ... + n = n*(n+1)/2
  Indicación: Usar la notación [0..<n+1] para la lista de los n primeros
  números naturales.
  ----------------------------------------------------------------------
*}

lemma "2 * suma [0..<n+1] = n * (n + 1)"
by (induct n) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Demostrar que la suma de la lista formada por n copias
  del número a es n*a.
  Indicación: Usar la expresión (replicate n a) para la lista formada
  por n copias del número a
  ----------------------------------------------------------------------
*}
```

```
lemma "suma (replicate n a) = n * a"
by (induct n) auto

text {*
  ---------------------------------------------------------------------
  Ejercicio 5. Definir, mediante recursión final, la función
     sumaF :: "nat list  nat
  tal que (sumaF xs) es la suma de los elementos de xs. Por ejemplo,
     sumaF [2,3,5] = 10
  ---------------------------------------------------------------------
*}

fun sumaFAux :: "nat list  nat  nat" where
  "sumaFAux []     n = n"
| "sumaFAux (x#xs) n = sumaFAux xs (x + n)"

definition sumaF :: "nat list  nat" where
  "sumaF xs  sumaFAux xs 0"

value "sumaF [2,3,5]" -- "= 10"

text {*
  ---------------------------------------------------------------------
  Ejercicio 6. Demostrar que las funciones suma y sumaF son
  equivalentes.
  ---------------------------------------------------------------------
*}

lemma sumaFAux_add:
  "sumaFAux xs (a+b) = a + sumaFAux xs b"
by (induct xs arbitrary: a b) auto

lemma [simp]: "sumaF [] = 0"
by (auto simp add: sumaF_def)

lemma [simp]: "sumaF (x#xs) = x + sumaF xs"
by (auto simp add: sumaF_def sumaFAux_add[THEN sym])

-- "Una demostración estructurada del lema anterior es"
lemma "sumaF (x#xs) = x + sumaF xs"
```

```
proof -
  have "sumaF (x#xs) = sumaFAux (x#xs) 0" by (simp add: sumaF_def)
  also have "... = sumaFAux xs (x + 0)" by simp
  also have "... = x + sumaFAux xs 0"
    using sumaFAux_add[THEN sym] by simp
  also have "... = x + sumaF xs" by (simp add: sumaF_def)
  finally show ?thesis .
qed

theorem "sumaF xs = suma xs"
by (induct xs) auto

end
```

6.1.9. Intercalación de listas

```
chapter {* T6R1i: Intercalación de listas *}

theory T6R1i
imports Main
begin

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     intercala :: "'a list  'a list  'a list"
  tal que (intercala xs ys) es la lista obtenida intercalando los
  elementos de xs e ys. Por   ejemplo,
     value "intercala [a,b] [u,v]"       -- "= [a, u, b, v]"
     value "intercala [a,b] [u,v,x,y,z]" -- "= [a, u, b, v, x, y, z]"
     value "intercala [a,b,c,d,e] [u,v]" -- "= [a, u, b, v, c, d, e]"
  ----------------------------------------------------------------------
*}

fun intercala :: "'a list  'a list  'a list" where
  "intercala [] ys = ys"
| "intercala xs [] = xs"
| "intercala (x#xs) (y#ys) = x # y # intercala xs ys"

value "intercala [a,b] [u,v]"       -- "= [a, u, b, v]"
value "intercala [a,b] [u,v,x,y,z]" -- "= [a, u, b, v, x, y, z]"
```

```
value "intercala [a,b,c,d,e] [u,v]" -- "= [a, u, b, v, c, d, e]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar la propiedad distributiva de la intercalación
  sobre la concatenación.
  ----------------------------------------------------------------------
*}

lemma
  assumes "length p = length u"
          "length q = length v"
  shows "intercala (p@q) (u@v) = intercala p u @ intercala q v"
using assms
by (induct p arbitrary: q u v) (simp, case_tac u, auto)

-- "Demostración estructurada del lema anterior"
lemma
  assumes "length p = length u"
          "length q = length v"
  shows "intercala (p@q) (u@v) = intercala p u @ intercala q v"
using assms
proof (induct p arbitrary: q u v)
  fix q u v :: "'a list"
  assume "length [] = length u" and "length q = length v"
  thus "intercala ([]@q) (u@v) = intercala [] u @ intercala q v"
    by auto
next
  fix a::"'a" and p q u v :: "'a list"
  assume
    "q u v. length p = length u; length q = length v
              intercala (p @ q) (u @ v) = intercala p u @ intercala q v"
    "length (a # p) = length u"
    "length q = length v"
  thus "intercala ((a#p)@q) (u@v) = intercala (a#p) u @ intercala q v"
  by (case_tac u, auto)
qed


end
```

6.1.10. Ordenación de listas por inserción

```
chapter {* T6R1j: Ordenación de listas por inserción *}

theory T6R1j_Ordenacion_de_listas_por_insercion
imports Main
begin

text {*
  En esta relación de ejercicios se define el algoritmo de ordenación de
  listas por inserción y se demuestra que es correcto.
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     inserta :: nat  nat list  nat list
  tal que (inserta a xs) es la lista obtenida insertando a delante del
  primer elemento de xs que es mayor o igual que a. Por ejemplo,
     inserta 3 [2,5,1,7] = [2,3,5,1,7]
  ---------------------------------------------------------------------- *}

fun inserta :: "nat  nat list  nat list" where
  "inserta a []     = [a]"
| "inserta a (x#xs) = (if a  x then a#x#xs else x # inserta a xs)"

value "inserta 3 [2,5,1,7]" -- "= [2,3,5,1,7]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Definir la función
     ordena :: nat list  nat list
  tal que (ordena xs) es la lista obtenida ordenando xs por inserción.
  Por ejemplo,

  ---------------------------------------------------------------------- *}

fun ordena :: "nat list  nat list" where
  "ordena []     = []"
| "ordena (x#xs) = inserta x (ordena xs)"
```

```
value "ordena [3,2,5,3]" -- "[2,3,3,5]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Definir la función
     menor :: nat  nat list  bool
  tal que (menor a xs) se verifica si a es menor o igual que todos los
  elementos de xs.Por ejemplo,
     menor 2 [3,2,5] = True
     menor 2 [3,0,5] = False
  ---------------------------------------------------------------------- *}

fun menor :: "nat  nat list  bool" where
  "menor a []     = True"
| "menor a (x#xs) = (a  x  menor a xs)"

value "menor 2 [3,2,5]" -- "= True"
value "menor 2 [3,0,5]" -- "= False"

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Definir la función
     ordenada :: nat list  bool
  tal que (ordenada xs) se verifica si xs es una lista ordenada de
  manera creciente. Por ejemplo,
     ordenada [2,3,3,5] = True
     ordenada [2,4,3,5] = False
  ---------------------------------------------------------------------- *}

fun ordenada :: "nat list  bool" where
  "ordenada []     = True"
| "ordenada (x#xs) = (menor x xs & ordenada xs)"

value "ordenada [2,3,3,5]" -- "= True"
value "ordenada [2,4,3,5]" -- "= False"

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Demostrar que si y es una cota inferior de xs y x  y,
  entonces x es una cota inferior de xs.
```

```
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma menor_menor:
  assumes "x  y"
  shows   "menor y xs  menor x xs"
using assms
by (induct xs) auto

-- "La demostración estructurada es"
lemma menor_menor_2:
  assumes "x  y"
  shows   "menor y xs  menor x xs"
proof (induct xs)
  show "menor y []  menor x []" by simp
next
  fix z zs
  assume HI: "menor y zs  menor x zs"
  show "menor y (z # zs)  menor x (z # zs)"
  proof
    assume sup: "menor y (z # zs)"
    show "menor x (z # zs)"
    proof (simp only: menor.simps(2))
      show "x  z  menor x zs"
      proof
          have "x  y" using assms .
          also have "y  z" using sup by simp
          finally show "x  z" by simp
      next
        have "menor y zs" using sup by simp
        with HI show "menor x zs" by simp
      qed
    qed
  qed
qed

text {*
  ------------------------------------------------------------------
  Ejercicio 6. Demostrar el siguiente teorema de corrección: x es una
  cota inferior de la lista obtenida insertando y en zs syss x  y y x
```

```
  es una cota inferior de zs.
  ------------------------------------------------------------- *}

-- "La demostración automática es"
lemma menor_inserta:
  "menor x (inserta y zs) = (x  y  menor x zs)"
by (induct zs) auto

-- "La demostración estructurada es"
lemma menor_inserta_2:
  "menor x (inserta y zs) = (x  y  menor x zs)"
proof (induct zs)
  show "menor x (inserta y []) = (x  y  menor x [])" by simp
next
  fix z zs
  assume HI: "menor x (inserta y zs) = (x  y  menor x zs)"
  show "menor x (inserta y (z#zs)) = (x  y  menor x (z#zs))"
  proof (cases "y  z")
    assume "y  z"
    hence "menor x (inserta y (z#zs)) = menor x (y#z#zs)" by simp
    also have "... = (x  y  menor x (z#zs))" by simp
    finally show ?thesis by simp
  next
    assume "ň(y  z)"
    hence "menor x (inserta y (z#zs)) =
                    menor x (z # inserta y zs)" by simp
    also have "... = (x  z  menor x (inserta y zs))" by simp
    also have "... = (x  z  x  y  menor x zs)" using HI by simp
    also have "... = (x  y  menor x (z#zs))" by auto
    finally show ?thesis by simp
  qed
qed

text {*
  ------------------------------------------------------------------------
  Ejercicio 6. Demostrar que al insertar un elemento la lista obtenida
  está ordenada syss lo estaba la original.
  ------------------------------------------------------------- *}

-- "La demostración automática es"
```

```
lemma ordenada_inserta:
  "ordenada (inserta a xs) = ordenada xs"
by (induct xs) (auto simp add: menor_menor menor_inserta)

-- "La demostración estructurada es"
lemma ordenada_inserta_2:
  "ordenada (inserta a xs) = ordenada xs"
proof (induct xs)
  show "ordenada (inserta a []) = ordenada []" by simp
next
  fix x xs
  assume HI: "ordenada (inserta a xs) = ordenada xs"
  show "ordenada (inserta a (x # xs)) = ordenada (x # xs)"
  proof (cases "a  x")
    assume "a  x"
    hence "ordenada (inserta a (x # xs)) =
           ordenada (a # x # xs)" by simp
    also have "... = (menor a (x#xs)  ordenada (x # xs))" by simp
    also have "... = ordenada (x # xs)"
      using ‘a  x‘  by (auto simp add: menor_menor)
    finally show "ordenada (inserta a (x # xs)) = ordenada (x # xs)"
      by simp
  next
    assume "ň(a  x)"
    hence "ordenada (inserta a (x # xs)) =
           ordenada (x # inserta a xs)" by simp
    also have "... = (menor x (inserta a xs)  ordenada (inserta a xs))"
      by simp
    also have "... = (menor x (inserta a xs)  ordenada xs)"
      using HI by simp
    also have "... = (menor x xs  ordenada xs)"
      using ‘ň(a  x)‘ by (simp add: menor_inserta)
    also have "... = ordenada (x # xs)" by simp
    finally show "ordenada (inserta a (x # xs)) = ordenada (x # xs)"
      by simp
  qed
qed

text {*
  ----------------------------------------------------------------------
```

```
  Ejercicio 7. Demostrar que, para toda lista xs, (ordena xs) está
  ordenada.
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
theorem ordenada_ordena:
  "ordenada (ordena xs)"
by (induct xs) (auto simp add: ordenada_inserta)

-- "La demostración estructurada es"
theorem ordenada_ordena_2:
  "ordenada (ordena xs)"
proof (induct xs)
  show "ordenada (ordena [])" by simp
next
  fix x xs
  assume "ordenada (ordena xs)"
  hence "ordenada (inserta x (ordena xs))"
    by (simp add: ordenada_inserta)
  thus "ordenada (ordena (x # xs))" by simp
qed

text {*
  ------------------------------------------------------------------------
  Nota. El teorema anterior no garantiza que ordena sea correcta, ya que
  puede que (ordena xs) no tenga los mismos elementos que xs. Por
  ejemplo, si se define (ordena xs) como [] se tiene que (ordena xs)
  está ordenada pero no es una ordenación de xs. Para ello, definimos la
  función cuenta.
  ------------------------------------------------------------------ *}

text {*
  ------------------------------------------------------------------------
  Ejercicio 8. Definir la función
     cuenta :: nat list => nat => nat
  tal que (cuenta xs y) es el número de veces que aparece el elemento y
  en la lista xs. Por ejemplo,
     cuenta [1,3,4,3,5] 3 = 2
  ------------------------------------------------------------------ *}
```

```
fun cuenta :: "nat list => nat => nat" where
  "cuenta []     y = 0"
| "cuenta (x#xs) y = (if x=y then Suc(cuenta xs y) else cuenta xs y)"

value "cuenta [1,3,4,3,5] 3" -- "= 2"

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Demostrar que el número de veces que aparece y en
  (inserta x xs) es
  * uno más el número de veces que aparece en xs, si y = x;
  * el número de veces que aparece en xs, si y  x;
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma cuenta_inserta:
  "cuenta (inserta x xs) y =
   (if x=y then Suc (cuenta xs y) else cuenta xs y)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma cuenta_inserta_2:
  "cuenta (inserta x xs) y =
   (if x=y then Suc (cuenta xs y) else cuenta xs y)"
  (is "?P x y xs")
proof (induct xs)
  show "?P x y []" by simp
next
  fix a xs
  assume "?P x y xs"
  thus "?P x y (a#xs)" by auto
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Demostrar que el número de veces que aparece y en
  (ordena xs) es el número de veces que aparece en xs.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
```

```
theorem cuenta_ordena:
  "cuenta (ordena xs) y = cuenta xs y"
by (induct xs) (auto simp add: cuenta_inserta)

-- "La demostración estructurada es"
theorem cuenta_ordena_2:
  "cuenta (ordena xs) y = cuenta xs y"
proof (induct xs)
  show "cuenta (ordena []) y = cuenta [] y" by simp
next
  fix x xs
  assume HI: "cuenta (ordena xs) y = cuenta xs y"
  show "cuenta (ordena (x # xs)) y = cuenta (x # xs) y"
  proof (cases "x = y")
    assume "x = y"
    have "cuenta (ordena (x # xs)) y = cuenta (inserta x (ordena xs)) y"
      by simp
    also have "... = Suc (cuenta (ordena xs) y)"
      using `x = y` by (simp add: cuenta_inserta)
    also have "... = Suc (cuenta xs y)" using HI by simp
    also have "... = cuenta (x # xs) y" using `x = y` by simp
    finally show "cuenta (ordena (x # xs)) y = cuenta (x # xs) y"
      by simp
  next
    assume "x ≠ y"
    have "cuenta (ordena (x # xs)) y = cuenta (inserta x (ordena xs)) y"
      by simp
    also have "... = cuenta (ordena xs) y"
      using `x ≠ y` by (simp add: cuenta_inserta)
    also have "... = cuenta xs y" using HI by simp
    also have "... = cuenta (x # xs) y" using `x ≠ y` by simp
    finally show "cuenta (ordena (x # xs)) y = cuenta (x # xs) y"
      by simp
  qed
qed

end
```

6.1.11. Ordenación de listas por mezcla

```
chapter {* T6R1k: Ordenación de listas por mezcla *}

theory T6R1k_Ordenacion_de_listas_por_mezcla
imports Main
begin

text {*
  En esta relación de ejercicios se define el algoritmo de ordenación de
  listas por mezcla y se demuestra que es correcto.
*}

section {* Ordenación de listas *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     menor :: nat  nat list  bool
  tal que (menor a xs) se verifica si a es menor o igual que todos los
  elementos de xs.Por ejemplo,
     menor 2 [3,2,5] = True
     menor 2 [3,0,5] = False
  ---------------------------------------------------------------------- *}

fun menor :: "nat  nat list  bool" where
  "menor a []     = True"
| "menor a (x#xs) = (a  x  menor a xs)"

value "menor 2 [3,2,5]" -- "= True"
value "menor 2 [3,0,5]" -- "= False"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Definir la función
     ordenada :: nat list  bool
  tal que (ordenada xs) se verifica si xs es una lista ordenada de
  manera creciente. Por ejemplo,
     ordenada [2,3,3,5] = True
     ordenada [2,4,3,5] = False
  ---------------------------------------------------------------------- *}
```

```
fun ordenada :: "nat list  bool" where
  "ordenada []     = True"
| "ordenada (x#xs) = (menor x xs & ordenada xs)"

value "ordenada [2,3,3,5]" -- "= True"
value "ordenada [2,4,3,5]" -- "= False"

text {*
  ---------------------------------------------------------------------
  Ejercicio 3. Definir la función
     cuenta :: nat list => nat => nat
  tal que (cuenta xs y) es el número de veces que aparece el elemento y
  en la lista xs. Por ejemplo,
     cuenta [1,3,4,3,5] 3 = 2
  --------------------------------------------------------------------- *}

fun cuenta :: "nat list => nat => nat" where
  "cuenta []     y = 0"
| "cuenta (x#xs) y = (if x=y then Suc(cuenta xs y) else cuenta xs y)"

value "cuenta [1,3,4,3,5] 3" -- "= 2"

section {* Ordenación por mezcla *}

text {*
  ---------------------------------------------------------------------
  Ejercicio 4. Definir la función
     mezcla :: nat list  nat list  nat list
  tal que (mezcla xs ys) es la lista obtenida mezclando las listas
  ordenadas xs e ys. Por ejemplo,
     mezcla [1,2,5] [3,5,7] = [1,2,3,5,5,7]
  --------------------------------------------------------------------- *}

fun mezcla :: "nat list  nat list  nat list" where
  "mezcla [] ys = ys"
| "mezcla xs [] = xs"
| "mezcla (x # xs) (y # ys) = (if x  y
                               then x # mezcla xs (y # ys)
                               else y # mezcla (x # xs) ys)"
```

```
value "mezcla [1,2,5] [3,5,7]" -- "= [1,2,3,5,5,7]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Definir la función
     ordenaM :: nat list  nat list
  tal que (ordenaM xs) es la lista obtenida ordenando la lista xs
  mediante mezclas; es decir, la divide en dos mitades, las ordena y las
  mezcla. Por ejemplo,

  ---------------------------------------------------------------------- *}

fun ordenaM :: "nat list  nat list" where
  "ordenaM [] = []"
| "ordenaM [x] = [x]"
| "ordenaM xs =
     (let mitad = length xs div 2 in
      mezcla (ordenaM (take mitad xs))
             (ordenaM (drop mitad xs)))"

value "ordenaM [3,2,5,2]" -- "= [2,2,3,5]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 6. Sea x  y. Si y es menor o igual que todos los elementos
  de xs, entonces x es menor o igual que todos los elementos de xs
  ---------------------------------------------------------------------- *}

lemma menor_menor:
  "x  y  menor y xs  menor x xs"
by (induct xs) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Demostrar que el número de veces que aparece n en la
  mezcla de dos listas es igual a la suma del número de apariciones en
  cada una de las listas
  ---------------------------------------------------------------------- *}
```

```
lemma cuenta_mezcla:
  "cuenta (mezcla xs ys) n = cuenta xs n + cuenta ys n"
by (induct xs ys rule: mezcla.induct) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 8. Demostrar que x es menor que todos los elementos de ys y
  de zs, entonces también lo es de su mezcla.
  ---------------------------------------------------------------------- *}

lemma menor_mezcla:
  assumes "menor x ys"
          "menor x zs"
  shows   "menor x (mezcla ys zs)"
using assms
by (induct ys zs rule: mezcla.induct) simp_all



text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Demostrar que la mezcla de dos listas ordenadas es una
  lista ordenada.
  Indicación: Usar los siguientes lemas
  ů linorder_not_le: (ň x  y) = (y < x)
  ů order_less_le:   (x < y) = (x  y  x  y)
  ---------------------------------------------------------------------- *}

lemma ordenada_mezcla:
  assumes "ordenada xs"
          "ordenada ys"
  shows   "ordenada (mezcla xs ys)"
using assms
by (induct xs ys rule: mezcla.induct)
   (auto simp add: menor_mezcla
                   menor_menor
                   linorder_not_le
                   order_less_le)

text {*
```

```
  ------------------------------------------------------------------
  Ejercicio 10. Demostrar que si x es mayor que 1, entonces el mínimo de
  x y su mitad es menor que x.
  Indicación: Usar los siguientes lemas
  ů min_def:          min a b = (if a  b then a else b)
  ů linorder_not_le: (ň x  y) = (y < x)
  ------------------------------------------------------------------ *}

lemma min_mitad:
  "1 < x  min x (x div 2::nat) < x"
by (simp add: min_def linorder_not_le)

text {*
  ------------------------------------------------------------------
  Ejercicio 11. Demostrar que si x es mayor que 1, entonces x menos su
  mitad es menor que x.
  ------------------------------------------------------------------ *}

lemma menos_mitad:
  "1 < x  x - x div (2::nat) < x"
by arith

text {*
  ------------------------------------------------------------------
  Ejercicio 11. Demostrar que (ordenaM xs) está ordenada.
  ------------------------------------------------------------------ *}

theorem ordenada_ordenaM:
  "ordenada (ordenaM xs)"
by (induct xs rule: ordenaM.induct)
   (auto simp add: ordenada_mezcla)

text {*
  ------------------------------------------------------------------
  Ejercicio 12. Demostrar que el número de apariciones de un elemento en
  la concatenación de dos listas es la suma del número de apariciones en
  cada una.
  ------------------------------------------------------------------ *}

lemma cuenta_conc:
```

```
  "cuenta (xs @ ys) x = cuenta xs x + cuenta ys x"
by (induct xs) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 13. Demostrar que las listas xs y (ordenaM xs) tienen los
  mismos elementos.
  ---------------------------------------------------------------------- *}

theorem cuenta_ordenaM:
  "cuenta (ordenaM xs) x = cuenta xs x"
by (induct xs rule: ordenaM.induct)
   (auto simp add: cuenta_mezcla
                   cuenta_conc [symmetric])

end
```

6.2. Ejercicios sobre árboles y otros tipos de datos inductivos

6.2.1. Recorridos de árboles

```
chapter {* T6R2a: Recorridos de árboles *}

theory T6R2a
imports Main
begin

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir el tipo de datos arbol para representar los
  árboles binarios que tiene información en los nodos y en las hojas.
  Por ejemplo, el árbol
          e
         / \
        /   \
       c     g
      / \   / \
     a   d f   h
  se representa por "N e (N c (H a) (H d)) (N g (H f) (H h))".
```

```
  ----------------------------------------------------------------------
*}

datatype 'a arbol = H "'a" | N "'a" "'a arbol" "'a arbol"

value "N e (N c (H a) (H d)) (N g (H f) (H h))"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Definir la función
     preOrden :: "'a arbol  'a list"
  tal que (preOrden a) es el recorrido pre orden del árbol a. Por
  ejemplo,
     preOrden (N e (N c (H a) (H d)) (N g (H f) (H h)))
     = [e,c,a,d,g,f,h]
  ----------------------------------------------------------------------
*}

fun preOrden :: "'a arbol  'a list" where
  "preOrden (H x)     = [x]"
| "preOrden (N x i d) = x#((preOrden i)@(preOrden d))"

value "preOrden (N e (N c (H a) (H d)) (N g (H f) (H h)))"
-- "= [e,c,a,d,g,f,h]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Definir la función
     postOrden :: "'a arbol  'a list"
  tal que (postOrden a) es el recorrido post orden del árbol a. Por
  ejemplo,
     postOrden (N e (N c (H a) (H d)) (N g (H f) (H h)))
     = [e,c,a,d,g,f,h]
  ----------------------------------------------------------------------
*}

fun postOrden :: "'a arbol  'a list" where
  "postOrden (H x)     = [x]"
| "postOrden (N x i d) = (postOrden i)@(postOrden d)@[x]"
```

```
value "postOrden (N e (N c (H a) (H d)) (N g (H f) (H h)))"
-- "[a,d,c,f,h,g,e]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Definir la función
     inOrden :: "'a arbol  'a list"
  tal que (inOrden a) es el recorrido in orden del árbol a. Por
  ejemplo,
     inOrden (N e (N c (H a) (H d)) (N g (H f) (H h)))
     = [a,c,d,e,f,g,h]
  ----------------------------------------------------------------------
*}

fun inOrden :: "'a arbol  'a list" where
  "inOrden (H x)     = [x]"
| "inOrden (N x i d) = (inOrden i)@[x]@(inOrden d)"

value "inOrden (N e (N c (H a) (H d)) (N g (H f) (H h)))"
-- "[a,c,d,e,f,g,h]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Definir la función
     espejo :: "'a arbol  'a arbol"
  tal que (espejo a) es la imagen especular del árbol a. Por ejemplo,
     espejo (N e (N c (H a) (H d)) (N g (H f) (H h)))
     = N e (N g (H h) (H f)) (N c (H d) (H a))
  ----------------------------------------------------------------------
*}

fun espejo :: "'a arbol  'a arbol" where
  "espejo (H x)     = (H x)"
| "espejo (N x i d) = (N x (espejo d) (espejo i))"

value "espejo (N e (N c (H a) (H d)) (N g (H f) (H h)))"
-- "N e (N g (H h) (H f)) (N c (H d) (H a))"

text {*
  ----------------------------------------------------------------------
```

```
  Ejercicio 6. Demostrar que
     preOrden (espejo a) = rev (postOrden a)
  ----------------------------------------------------------------------
*}

lemma  "preOrden (espejo a) = rev (postOrden a)"
by (induct a) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Demostrar que
     postOrden (espejo a) = rev (preOrden a)
  ----------------------------------------------------------------------
*}

lemma "postOrden (espejo a) = rev (preOrden a)"
by (induct a) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 8. Demostrar que
     inOrden (espejo a) = rev (inOrden a)
  ----------------------------------------------------------------------
*}

theorem "inOrden (espejo a) = rev (inOrden a)"
by (induct a) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Definir la función
     raiz :: "'a arbol  'a"
  tal que (raiz a) es la raiz del árbol a. Por ejemplo,
     raiz (N e (N c (H a) (H d)) (N g (H f) (H h))) = e
  ----------------------------------------------------------------------
*}

fun raiz :: "'a arbol  'a" where
  "raiz (H x)     = x"
| "raiz (N x i d) = x"
```

```
value "raiz (N e (N c (H a) (H d)) (N g (H f) (H h)))" -- "= e"

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Definir la función
     extremo_izquierda :: "'a arbol  'a"
  tal que (extremo_izquierda a) es el nodo más a la izquierda del árbol
  a. Por ejemplo,
     extremo_izquierda (N e (N c (H a) (H d)) (N g (H f) (H h))) = a
  ----------------------------------------------------------------------
*}

fun extremo_izquierda :: "'a arbol  'a" where
  "extremo_izquierda (H a)      = a"
| "extremo_izquierda (N f x y) = (extremo_izquierda x)"

value "extremo_izquierda (N e (N c (H a) (H d)) (N g (H f) (H h)))" -- "= a"

text {*
  ----------------------------------------------------------------------
  Ejercicio 11. Definir la función
     extremo_derecha :: "'a arbol  'a"
  tal que (extremo_derecha a) es el nodo más a la derecha del árbol
  a. Por ejemplo,
     extremo_derecha (N e (N c (H a) (H d)) (N g (H f) (H h))) = h
  ----------------------------------------------------------------------
*}

fun extremo_derecha :: "'a arbol  'a" where
  "extremo_derecha (H x)     = x"
| "extremo_derecha (N x i d) = (extremo_derecha d)"

value "extremo_derecha (N e (N c (H a) (H d)) (N g (H f) (H h)))" -- "= h"

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Demostrar o refutar
     last (inOrden a) = extremo_derecha a
  ----------------------------------------------------------------------
```

```
*}

lemma [simp]: "inOrden a  []"
by (induct a) auto

lemma [simp]: "ys  []  last (xs@ys) = last ys"
by (induct xs) auto

theorem "last (inOrden a) = extremo_derecha a"
by (induct a) auto

text {*
  ---------------------------------------------------------------------
  Ejercicio 13. Demostrar o refutar
     hd (inOrden a) = extremo_izquierda a
  ---------------------------------------------------------------------
*}

theorem "hd (inOrden a) = extremo_izquierda a"
by (induct a) auto

text {*
  ---------------------------------------------------------------------
  Ejercicio 14. Demostrar o refutar
     hd (preOrden a) = last (postOrden a)
  ---------------------------------------------------------------------
*}

theorem "hd (preOrden a) = last (postOrden a)"
by (induct a) auto

text {*
  ---------------------------------------------------------------------
  Ejercicio 15. Demostrar o refutar
     hd (preOrden a) = raiz a
  ---------------------------------------------------------------------
*}

theorem "hd (preOrden a) = raiz a"
by (induct a) auto
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 16. Demostrar o refutar
     hd (inOrden a) = raiz a
  ----------------------------------------------------------------------
*}

theorem "hd (inOrden a) = raiz a"
quickcheck
oops

text {*
  Quickcheck found a counterexample:
  a = N a\<^isub>1 (H a\<^isub>2) (H a\<^isub>1)

  Evaluated terms:
  hd (inOrden a) = a\<^isub>2
  raiz a = a\<^isub>1
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 17. Demostrar o refutar
     last (postOrden a) = raiz a
  ----------------------------------------------------------------------
*}

theorem "last (postOrden a) = raiz a"
by (induct a) auto

end
```

6.2.2. Plegados de listas y de árboles

```
chapter {* T6R2b: Plegados de listas y de árboles *}

theory T6R2b
imports Main
begin
```

```
section {* Nuevas funciones sobre listas *}

text {*
  Nota. En esta relación se usará la función suma tal que (suma xs) es
  la suma de los elementos de xs, definida por
*}

fun suma :: "nat list  nat" where
  "suma []       = 0"
| "suma (x # xs) = x + suma xs"

text {*
  Las funciones de plegado, foldr y foldl, están definidas en la teoría
  List.thy por
       foldr :: "('a  'b  'b)  'a list  'b  'b"
       foldr f []       = id
       foldr f (x # xs) = f x  foldr f xs

       foldl :: "('b  'a  'b)  'b  'a list  'b"
       foldl f a []       = a
       foldl f a (x # xs) = foldl f (f a x) xs"

  Por ejemplo,
     foldr (op +) [a,b,c] d        = a + (b + (c + d))
     foldl (op +) d [a,b,c]        = ((d + a) + b) + c
     foldr (op -) [9,4,2] (0::int) = 7
     foldl (op -) (0::int) [9,4,2] = -15
*}

value "foldr (op +) [a,b,c] d"        -- "= a + (b + (c + d))"
value "foldl (op +) d [a,b,c]"        -- "= ((d + a) + b) + c"
value "foldr (op -) [9,4,2] (0::int)" -- "= 7"
value "foldl (op -) (0::int) [9,4,2]" -- "= -15"

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Demostrar que
     suma xs = foldr (op +) xs 0
  ----------------------------------------------------------------------
*}
```

```
lemma suma_foldr: "suma xs = foldr (op +) xs 0"
by (induct xs) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar que
     length xs = foldr ( x res. 1 + res) xs 0
  ----------------------------------------------------------------------
*}

lemma length_foldr: "length xs = foldr ( x res. 1 + res) xs 0"
by (induct xs) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. La aplicación repetida de foldr y map tiene el
  inconveniente de que la lista se recorre varias veces. Sin embargo, es
  suficiente recorrerla una vez como se muestra en el siguiente ejemplo,
     suma (map (x. x + 3) xs) = foldr h xs b
  Determinar los valores de h y b para que se verifique la igualdad
  anterior y demostrarla.
  ----------------------------------------------------------------------
*}

text {* Basta tomar ( x y. x + y + 3) como h y 0 como b *}
lemma "suma (map (x. x + 3) xs) = foldr ( x y. x + y + 3) xs 0"
by (induct xs) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Generalizar el resultado anterior; es decir determinar
  los valores de h y b para que se verifique la igualdad
     foldr g (map f xs) a = foldr h xs b
  y demostrarla.
  ----------------------------------------------------------------------
*}

text {* Basta tomar (x acc. g (f x) acc) como h y a como b *}
lemma "foldr g (map f xs) a = foldr (x acc. g (f x) acc) xs a"
```

```
by (induct xs) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. La siguiente función invierte una lista en tiempo lineal
     fun inversa_ac :: "['a list, 'a list]  'a list" where
       "inversa_ac [] ys = ys"
     | "inversa_ac (x#xs) ys = (inversa_ac xs (x#ys))"

     definition inversa_ac :: "'a list  'a list" where
       "inversa_ac xs  inversa_ac_aux xs []"
  Por ejemplo,
     inversa_ac [a,d,b,c] = [c, b, d, a]
  Demostrar que inversa_ac se puede definir unsando foldl.
  ----------------------------------------------------------------------
*}

fun inversa_ac_aux :: "['a list, 'a list]  'a list" where
  "inversa_ac_aux [] ys = ys"
| "inversa_ac_aux (x#xs) ys = (inversa_ac_aux xs (x#ys))"

definition inversa_ac :: "'a list  'a list" where
  "inversa_ac xs  inversa_ac_aux xs []"

value "inversa_ac [a,d,b,c]" -- "= [c, b, d, a]"

lemma inversa_ac_aux_foldl_aux:
  "inversa_ac_aux xs a = foldl ( ys x. x # ys) a xs"
by (induct xs arbitrary: a) auto

corollary inversa_ac_aux_foldl:
  "inversa_ac_aux xs a = foldl ( ys x. x # ys) a xs"
by (rule inversa_ac_aux_foldl_aux)

text {*
  ----------------------------------------------------------------------
  Ejercicio 6. Demostrar la siguiente propiedad distributiva de la suma
  sobre la concatenación:
     suma (xs @ ys) = suma xs + suma ys
  ----------------------------------------------------------------------
```

```
*}

lemma suma_append [simp]:
  "suma (xs @ ys) = suma xs + suma ys"
by (induct xs) auto

text {*
  ---------------------------------------------------------------------
  Ejercicio 7. Demostrar una propiedad similar para foldr
     foldr f (xs @ ys) a = f (foldr f xs a) (foldr f ys a)
  En este caso, hay que restringir el resultado teniendo en cuenta
  propiedades algebraicas de f y a.
  ---------------------------------------------------------------------
*}

text {* Definición de a es neutro por la izquiera de f *}
definition neutro_izquierda :: "['a  'b  'b, 'a]  bool" where
  "neutro_izquierda f a  (x. (f a x = x))"

text {* Definición de f es asociativa *}
definition asociativa :: "['a  'a  'a]  bool" where
  "asociativa f  ( x y z. f (f x y) z = f x (f y z))"

lemma foldr_append:
  assumes "neutro_izquierda f a"
          "asociativa f"
  shows "foldr f (xs @ ys) a = f (foldr f xs a) (foldr f ys a)"
using assms
by (induct xs) (auto simp add: neutro_izquierda_def asociativa_def)

text {*
  ---------------------------------------------------------------------
  Ejercicio 8. Definir, usando foldr, la función
     prod :: "nat list  nat"
  tal que (prod xs) es el producto de los elementos de xs. Por ejemplo,
  ---------------------------------------------------------------------
*}

definition prod :: "nat list  nat" where
  "prod xs  foldr (op *) xs 1"
```

```
value "prod [2::nat,3,5]" -- "= 30"

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Demostrar directamente (es decir, sin inducción) que
     prod (xs @ ys) = prod xs * prod ys
  ----------------------------------------------------------------------
*}

lemma "prod (xs @ ys) = prod xs * prod ys"
unfolding prod_def
by (rule foldr_append,
    simp add: neutro_izquierda_def,
    simp add: asociativa_def)

-- "Una demostración aplicativa del lema anterior es"
lemma "prod (xs @ ys) = prod xs * prod ys"
  apply (simp only: prod_def)
  apply (rule foldr_append)
    apply (simp add: neutro_izquierda_def)
  apply (simp add: asociativa_def)
done

section {* Functions on Trees *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Definir el tipo de datos arbol para representar los
  árboles binarios que tiene información sólo en los nodos. Por ejemplo,
  el árbol
           e
          / \
         /   \
        c     g
       / \   / \
      ů   ů ů   ů
  se representa por "N (N H c H) e (N H g H)"
  ----------------------------------------------------------------------
*}
```

```
datatype 'a arbol = H | N "'a arbol" "'a" "'a arbol"

value "N (N H c H) e (N H g H)"

text {*
  ----------------------------------------------------------------------
  Ejercicio 11. Definir la función
     preOrden :: "'a arbol  'a list"
  tal que (preOrden a) es el recorrido pre orden del árbol a. Por
  ejemplo,
     preOrden (N (N H c H) e (N H g H))
     = [e,c,g]
  ----------------------------------------------------------------------
*}

fun preOrden :: "'a arbol  'a list" where
  "preOrden H         = []"
| "preOrden (N i x d) = x#((preOrden i)@(preOrden d))"

value "preOrden (N (N H c H) e (N H g H))"
-- "= [e,c,g]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Definir la función
     postOrden :: "'a arbol  'a list"
  tal que (postOrden a) es el recorrido post orden del árbol a. Por
  ejemplo,
     postOrden (N (N H c H) e (N H g H))
     = [c,g,e]
  ----------------------------------------------------------------------
*}

fun postOrden :: "'a arbol  'a list" where
  "postOrden H         = []"
| "postOrden (N i x d) = (postOrden i)@(postOrden d)@[x]"

value "postOrden (N (N H c H) e (N H g H))"
-- "= [c,g,e]"
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 13. Definir, usando un acumulador, la función
     postOrdenA :: "'a arbol  'a list"
  tal que (postOrdenA a) es el recorrido post orden del árbol a. Por
  ejemplo,
     postOrdenA (N (N H c H) e (N H g H))
     = [c,g,e]
  ----------------------------------------------------------------------
*}

fun postOrdenAaux :: "['a arbol, 'a list]  'a list" where
  "postOrdenAaux H          xs = xs"
| "postOrdenAaux (N i x d) xs = (postOrdenAaux i (postOrdenAaux d (x#xs)))"

definition postOrdenA :: "'a arbol  'a list" where
  "postOrdenA a  postOrdenAaux a []"

value "postOrdenA (N (N H c H) e (N H g H))"
-- "= [c,g,e]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 14. Demostrar que
     postOrdenAaux a xs = (postOrden a) @ xs
  ----------------------------------------------------------------------
*}


lemma postOrdenA:
  "postOrdenAaux a xs = (postOrden a) @ xs"
by (induct a arbitrary: xs) auto

corollary "postOrdenAaux a xs = (postOrden a) @ xs"
by (rule postOrdenA)

text {*
  ----------------------------------------------------------------------
  Ejercicio 15. Definir la función
```

```
    foldl_arbol :: "('b => 'a => 'b)  'b  'a arbol  'b" where
  tal que (foldl_arbol f b a) es el plegado izquierdo del árbol a con la
  operación f y elemento inicial b.
  ------------------------------------------------------------------------
*}

fun foldl_arbol :: "('b => 'a => 'b)  'b  'a arbol  'b" where
  "foldl_arbol f b H         = b"
| "foldl_arbol f b (N i x d) = (foldl_arbol f (foldl_arbol f (f b x) d) i)"

text {*
  ------------------------------------------------------------------------
  Ejercicio 16. Demostrar que
    postOrdenAaux t a = foldl_arbol ( xs x. Cons x xs) a t
  ------------------------------------------------------------------------
*}

lemma "postOrdenAaux t a = foldl_arbol ( xs x. Cons x xs) a t"
by (induct t arbitrary: a) auto

text {*
  ------------------------------------------------------------------------
  Ejercicio 17. Definir la función
    suma_arbol :: "nat arbol  nat"
  tal que (suma_arbol a) es la suma de los elementos del árbol de
  números naturales a.
  ------------------------------------------------------------------------
*}

fun suma_arbol :: "nat arbol  nat" where
  "suma_arbol H          = 0"
| "suma_arbol (N i x d) = (suma_arbol i) + x + (suma_arbol d)"

text {*
  ------------------------------------------------------------------------
  Ejercicio 18. Demostrar que
    suma_arbol a = suma (preOrden a)"
  ------------------------------------------------------------------------
*}
```

```
lemma "suma_arbol a = suma (preOrden a)"
by (induct a) auto

end
```

6.2.3. Árboles binarios completos

```
chapter {* T6R2c: Árboles binarios completos *}

theory T6R2c
imports Main
begin

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir el tipo de datos arbol para representar los
  árboles binarios que no tienen información ni en los nodos y ni en las
  hojas. Por ejemplo, el árbol
          ů
         / \
        /   \
       ů     ů
      / \   / \
     ů   ů ů   ů
  se representa por "N (N H H) (N H H)".
  ----------------------------------------------------------------------
*}

datatype arbol = H | N arbol arbol

value "N (N H H) (N H H)"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Definir la función
     hojas :: "arbol => nat"
  tal que (hojas a) es el número de hojas del árbol a. Por ejemplo,
     hojas (N (N H H) (N H H)) = 4
  ----------------------------------------------------------------------
*}
```

```
fun hojas :: "arbol => nat" where
  "hojas H       = 1"
| "hojas (N i d) = hojas i + hojas d"

value "hojas (N (N H H) (N H H))" -- "= 4"

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Definir la función
     profundidad :: "arbol => nat"
  tal que (profundidad a) es la profundidad del árbol a. Por ejemplo,
     profundidad (N (N H H) (N H H)) = 2
  ----------------------------------------------------------------------
*}

fun profundidad :: "arbol => nat" where
  "profundidad H       = 0"
| "profundidad (N i d) = 1 + max (profundidad i) (profundidad d)"

value "profundidad (N (N H H) (N H H))" -- "= 2"

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Definir la función
     abc :: "nat  arbol"
  tal que (abc n) es el árbol binario completo de profundidad n. Por
  ejemplo,
     abc 3 = N (N (N H H) (N H H)) (N (N H H) (N H H))
  ----------------------------------------------------------------------
*}

fun abc :: "nat  arbol" where
  "abc 0       = H"
| "abc (Suc n) = N (abc n) (abc n)"

value "abc 3" -- "= N (N (N H H) (N H H)) (N (N H H) (N H H))"

text {*
  ----------------------------------------------------------------------
```

```
  Ejercicio 6. Un árbol binario a es completo respecto de la medida f si
  a es una hoja o bien a es de la forma (N i d) y se cumple que tanto i
  como d son árboles binarios completos respecto de f y, además,
  f(i) = f(r).

  Definir la función
     es_abc :: "(arbol => 'a) => arbol => bool
  tal que (es_abc f a) se verifica si a es un árbol binario completo
  respecto de f.
  ----------------------------------------------------------------------
*}

fun es_abc :: "(arbol => 'a) => arbol => bool" where
  "es_abc f H       = True"
| "es_abc f (N i d) = (es_abc f i  es_abc f d  f i = f d)"

text {*
  ----------------------------------------------------------------------
  Nota. (size a) es el número de nodos del árbol a. Por ejemplo,
     size (N (N H H) (N H H)) = 3
  ----------------------------------------------------------------------
*}

value "size (N (N H H) (N H H))"
value "size (N (N (N H H) (N H H)) (N (N H H) (N H H)))"

text {*
  ----------------------------------------------------------------------
  Nota. Tenemos 3 funciones de medida sobre los árboles: número de
  hojas, número de nodos y profundidad. A cada una le corresponde un
  concepto de completitud. En los siguientes ejercicios demostraremos
  que los tres conceptos de completitud son iguales.
  ----------------------------------------------------------------------
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Demostrar que un árbol binario a es completo respecto de
  la profundidad syss es completo respecto del número de hojas.
  ----------------------------------------------------------------------
```

```
*}

text {* Si a es un árbol completo respecto de la profundidad, entonces
  el número de hojas de a es 2 elevado a la profundidad de a. *}
lemma [simp]: "es_abc profundidad a  hojas a = 2 ^ (profundidad a)"
by (induct a) auto

theorem es_abc_profundidad_hojas: "es_abc profundidad a = es_abc hojas a"
by (induct a) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 8. Demostrar que un árbol binario a es completo respecto del
  número de hojas syss es completo respecto del número de nodos
  ----------------------------------------------------------------------
*}

text {* El número de hojas de un árbol binario a es igual al número de
  nodos da a más 1. *}
lemma [simp]: "hojas a = size a + 1"
by (induct a) auto

theorem es_abc_hojas_size: "es_abc hojas a = es_abc size a"
by (induct a) auto


text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Demostrar que un árbol binario a es completo respecto de
  la profundidad syss es completo respecto del número de nodos
  ----------------------------------------------------------------------
*}

corollary es_abc_size_profundidad: "es_abc size a = es_abc profundidad a"
by (simp add: es_abc_profundidad_hojas es_abc_hojas_size)

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Demostrar que (abc n) es un árbol binario completo.
  ----------------------------------------------------------------------
```

```
*}

lemma "es_abc f (abc n)"
by (induct n) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 11. Demostrar que si a es un árbolo binario completo
  respecto de la profundidad, entonces a es (abc (profundidad a)).
  ----------------------------------------------------------------------
*}

theorem "es_abc profundidad a  a = abc (profundidad a)"
by (induct a) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Encontrar una medida f tal que (es_abc f) es distinto de
  (es_abc size).
  ----------------------------------------------------------------------
*}

text {* Basta tomar como f la función constante 0. *}

value "es_abc size (N H (N H H))"     -- "= False"
value "es_abc (t. 0) (N H (N H H))" -- "= True"

lemma "es_abc (t. 0::nat)  es_abc size"
proof
  assume "es_abc (t. 0::nat) = es_abc size"
  hence "(es_abc (t. 0::nat) (N H (N H H))) = (es_abc size (N H (N H H)))"
    by (simp add: fun_eq_iff)
  thus False by simp
qed

text {*
  Referencia: Este ejercicio es una adaptación del de Tobias Nipkow
  "Complete Binary Trees" que se encuentra en
  http://isabelle.in.tum.de/exercises/trees/complete/ex.pdf
*}
```

```
end
```

6.2.4. Diagramas de decisión binarios

```
chapter {* T6R2d: Diagramas de decisión binarios *}

theory T6R2d
imports Main
begin

text {*
  Las funciones booleanas se pueden representar mediante diagramas de
  decisión binarios (DDB). Por ejemplo, la función f definida por la
  tabla de la izquierda se representa por el DDB de la derecha
     +---+---+---+----------+            p
     | p | q | r | f(p,q,r) |           / \
     +---+---+---+----------+          /   \
     | F | F | * | V        |         q     q
     | F | V | * | F        |        / \   / \
     | V | F | * | F        |       V   F F   r
     | V | V | F | F        |                / \
     | V | V | V | V        |               F   V
     +---+---+---+----------+
  Para cada variable, si su valor es falso se evalúa su hijo izquierdo y
  si es verdadero se evalúa su hijo derecho.
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir el tipo de datos ddb para representar los
  diagramas de decisión binarios. Por ejemplo, el DDB anterior se
  representa por
     N (N (H True) (H False)) (N (H False) (N (H False) (H True)))
  ----------------------------------------------------------------------
*}

datatype ddb = H bool | N ddb ddb

value "N (N (H True) (H False)) (N (H False) (N (H False) (H True)))"
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Definir ddb1 para representar el DDB del ejercicio 1.
  ----------------------------------------------------------------------
*}

abbreviation ddb1 :: ddb where
  "ddb1  N (N (H True) (H False)) (N (H False) (N (H False) (H True)))"

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Definir int1,..., int8 para representar las
  interpretaciones del ejercicio 1.
  ----------------------------------------------------------------------
*}

abbreviation int1 :: "nat  bool" where
  "int1 x  False"
abbreviation int2 :: "nat  bool" where
  "int2  int1 (2 := True)"
abbreviation int3 :: "nat  bool" where
  "int3  int1 (1 := True)"
abbreviation int4 :: "nat  bool" where
  "int4  int1 (1 := True, 2 := True)"
abbreviation int5 :: "nat  bool" where
  "int5  int1 (0 := True)"
abbreviation int6 :: "nat  bool" where
  "int6  int1 (0 := True, 2 := True)"
abbreviation int7 :: "nat  bool" where
  "int7  int1 (0 := True, 1 := True)"
abbreviation int8 :: "nat  bool" where
  "int8  int1 (0 := True, 1 := True, 2 := True)"

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Definir la función
     valor :: "(nat  bool)  nat  ddb  bool"
  tal que (valor i n d) es el valor del DDB d en la interpretación i a
  partir de la variable de índice n. Por ejemplo,
     valor int1 0 ddb1 = True
```

```
     valor int2 0 ddb1 = True
     valor int3 0 ddb1 = False
     valor int4 0 ddb1 = False
     valor int5 0 ddb1 = False
     valor int6 0 ddb1 = False
     valor int7 0 ddb1 = False
     valor int8 0 ddb1 = True
  ------------------------------------------------------------------------
*}

fun valor :: "(nat  bool)  nat  ddb  bool" where
  "valor i n (H x) = x"
| "valor i n (N b1 b2) =
     (if i n then valor i (Suc n) b2 else valor i (Suc n) b1)"

value "valor int1 0 ddb1" -- "= True"
value "valor int2 0 ddb1" -- "= True"
value "valor int3 0 ddb1" -- "= False"
value "valor int4 0 ddb1" -- "= False"
value "valor int5 0 ddb1" -- "= False"
value "valor int6 0 ddb1" -- "= False"
value "valor int7 0 ddb1" -- "= False"
value "valor int8 0 ddb1" -- "= True"

text {*
  ------------------------------------------------------------------------
  Ejercicio 5. Definir la función
     ddb_op_un :: "(bool  bool)  ddb  ddb
  tal que (ddb_op_un f d) es el diagrama obtenido aplicando el operador
  unitario f a cada hoja de DDB d de forma que se conserve el valor; es
  decir,
     valor i n (ddb_op_un f d) = f (valor i n d)"
  Por ejemplo,
     value "ddb_op_un (x. ňx) ddb1"
     = N (N (H False) (H True)) (N (H True) (N (H True) (H False)))
  ------------------------------------------------------------------------
*}

fun ddb_op_un :: "(bool  bool)  ddb  ddb" where
  "ddb_op_un f (H x) = H (f x)"
```

```
| "ddb_op_un f (N b1 b2) = N (ddb_op_un f b1) (ddb_op_un f b2)"

value "ddb_op_un (x. ňx) ddb1"
-- "= N (N (H False) (H True)) (N (H True) (N (H True) (H False)))"

text {*
  ---------------------------------------------------------------------
  Ejercicio 6. Demostrar que la definición de ddb_op_un es correcta; es
  decir,
     valor i n (ddb_op_un f d) = f (valor i n d)"
  ---------------------------------------------------------------------
*}

theorem ddb_op_un_correcto:
  "valor i n (ddb_op_un f d) = f (valor i n d)"
by (induct d arbitrary: i n) auto


text {*
  ---------------------------------------------------------------------
  Ejercicio 7. Definir la función
     ddb_op_bin :: "(bool  bool  bool)  ddb  ddb  ddb"
  tal que (ddb_op_bin f d1 d2) es el diagrama obtenido aplicando el
  operador binario f a los DDB d1 y d2 de forma que se conserve el
  valor; es decir,
     valor i n (ddb_op_bin f d1 d2) = f (valor i n d1) (valor i n d2)
  Por ejemplo,
     ddb_op_bin (op ) ddb1 (N (H True) (H False))
     = N (N (H True) (H False)) (N (H False) (N (H False) (H False)))
     ddb_op_bin (op ) ddb1 (N (H False) (H True))
     = N (N (H False) (H False)) (N (H False) (N (H False) (H True)))
     ddb_op_bin (op ) ddb1 (N (H True) (H False))
     = N (N (H True) (H True)) (N (H False) (N (H False) (H True)))
     ddb_op_bin (op ) ddb1 (N (H False) (H True))
     = N (N (H True) (H False)) (N (H True) (N (H True) (H True)))
  ---------------------------------------------------------------------
*}

fun ddb_op_bin :: "(bool  bool  bool)  ddb  ddb  ddb" where
  "ddb_op_bin f (H x) d = ddb_op_un (f x) d"
```

```
| "ddb_op_bin f (N d1 d2) d = (case d of
       H x        N (ddb_op_bin f d1 (H x)) (ddb_op_bin f d2 (H x))
     | N d1' d2'  N (ddb_op_bin f d1 d1') (ddb_op_bin f d2 d2'))"

value "ddb_op_bin (op ) ddb1 (N (H True) (H False))"
-- "N (N (H True) (H False)) (N (H False) (N (H False) (H False)))"
value "ddb_op_bin (op ) ddb1 (N (H False) (H True))"
-- "N (N (H False) (H False)) (N (H False) (N (H False) (H True)))"
value "ddb_op_bin (op ) ddb1 (N (H True) (H False))"
-- "N (N (H True) (H True)) (N (H False) (N (H False) (H True)))"
value "ddb_op_bin (op ) ddb1 (N (H False) (H True))"
-- "N (N (H True) (H False)) (N (H True) (N (H True) (H True)))"

text {*
  ---------------------------------------------------------------------
  Ejercicio 8. Demostrar que la definición de ddb_op_bin es correcta;
  es decir,
     valor i n (ddb_op_bin f d1 d2) = f (valor i n d1) (valor i n d2)
  ---------------------------------------------------------------------
*}

theorem ddb_op_bin_correcto:
  "valor i n (ddb_op_bin f d1 d2) = f (valor i n d1) (valor i n d2)"
proof (induct d1 arbitrary: n d2)
  fix b n b2
  show "valor i n (ddb_op_bin f (H b) b2) =
        f (valor i n (H b)) (valor i n b2)"
    by (auto simp add: ddb_op_un_correcto)
next
  fix d11 d12 n d2
  assume "n d2. valor i n (ddb_op_bin f d11 d2) =
                f (valor i n d11) (valor i n d2)"
     and "n d2. valor i n (ddb_op_bin f d12 d2) =
                f (valor i n d12) (valor i n d2)"
  thus "valor i n (ddb_op_bin f (N d11 d12) d2) =
        f (valor i n (N d11 d12)) (valor i n d2)"
    by (cases d2) auto
qed

-- "Una demostración más detallada es"
```

```
theorem ddb_op_bin_correcto_2:
  "valor i n (ddb_op_bin f d1 d2) = f (valor i n d1) (valor i n d2)"
proof (induct d1 arbitrary: n d2)
  fix b n b2
  show "valor i n (ddb_op_bin f (H b) b2) =
        f (valor i n (H b)) (valor i n b2)"
    by (auto simp add: ddb_op_un_correcto)
next
  fix d11 d12 n d2
  assume HI1: "n d2. valor i n (ddb_op_bin f d11 d2) =
                     f (valor i n d11) (valor i n d2)"
     and HI2: "n d2. valor i n (ddb_op_bin f d12 d2) =
                     f (valor i n d12) (valor i n d2)"
  show "valor i n (ddb_op_bin f (N d11 d12) d2) =
        f (valor i n (N d11 d12)) (valor i n d2)"
  proof (cases d2)
    fix b
    assume "d2 = H b"
    thus "valor i n (ddb_op_bin f (N d11 d12) d2) =
          f (valor i n (N d11 d12)) (valor i n d2)"
      using HI1 HI2 by auto
  next
    fix d21 d22
    assume "d2 = N d21 d22"
    thus "valor i n (ddb_op_bin f (N d11 d12) d2) =
          f (valor i n (N d11 d12)) (valor i n d2)"
      using HI1 HI2 by auto
  qed
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 9. Definir la función
     ddb_and :: "ddb  ddb  ddb"
  tal que (ddb_and d1 d2) es el diagrama correspondiente a la conjunción
  de los DDB d1 y d2 de forma que se conserva el valor; es decir,
     valor i n (ddb_and d1 d2) = (valor i n d1  valor i n d2)
  Por ejemplo,
     ddb_and ddb1 (N (H True) (H False))
     = N (N (H True) (H False)) (N (H False) (N (H False) (H False)))
```

```
     ddb_and ddb1 (N (H False) (H True))
     = N (N (H False) (H False)) (N (H False) (N (H False) (H True)))
  ------------------------------------------------------------------
*}

definition ddb_and :: "ddb  ddb  ddb" where
  "ddb_and  ddb_op_bin (op )"

value "ddb_and ddb1 (N (H True) (H False))"
-- "= N (N (H True) (H False)) (N (H False) (N (H False) (H False)))"
value "ddb_and ddb1 (N (H False) (H True))"
-- "= N (N (H False) (H False)) (N (H False) (N (H False) (H True)))"

text {*
  ------------------------------------------------------------------
  Ejercicio 10. Demostrar que la definición de ddb_and es correcta;
  es decir,
     valor i n (ddb_and d1 d2) = (valor i n d1  valor i n d2)
  ------------------------------------------------------------------
*}

theorem ddb_and_correcta:
  "valor i n (ddb_and d1 d2) = (valor i n d1  valor i n d2)"
by (auto simp add: ddb_and_def ddb_op_bin_correcto)

text {*
  ------------------------------------------------------------------
  Ejercicio 11. Definir la función
     ddb_or :: "ddb  ddb  ddb"
  tal que (ddb_or d1 d2) es el diagrama correspondiente a la disyunción
  de los DDB d1 y d2 de forma que se conserva el valor; es decir,
     valor i n (ddb_or d1 d2) = (valor i n d1  valor i n d2)
  Por ejemplo,
     ddb_or ddb1 (N (H True) (H False))
     = N (N (H True) (H True)) (N (H False) (N (H False) (H True)))
     ddb_or ddb1 (N (H False) (H True))
     = N (N (H True) (H False)) (N (H True) (N (H True) (H True)))
  ------------------------------------------------------------------
*}
```

```
definition ddb_or :: "ddb  ddb  ddb" where
  "ddb_or  ddb_op_bin (op )"

value "ddb_or ddb1 (N (H True) (H False))"
-- "= N (N (H True) (H True)) (N (H False) (N (H False) (H True)))"
value "ddb_or ddb1 (N (H False) (H True))"
-- "= N (N (H True) (H False)) (N (H True) (N (H True) (H True)))"

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Demostrar que la definición de ddb_or es correcta;
  es decir,
     valor i n (ddb_or d1 d2) = (valor i n d1  valor i n d2)
  ----------------------------------------------------------------------
*}

theorem ddb_or_correcta:
  "valor i n (ddb_or d1 d2) = (valor i n d1  valor i n d2)"
by (auto simp add: ddb_or_def ddb_op_bin_correcto)

text {*
  ----------------------------------------------------------------------
  Ejercicio 13. Definir la función
     ddb_not :: "ddb  ddb"
  tal que (ddb_not d) es el diagrama correspondiente a la negación
  del DDB d de forma que se conserva el valor; es decir,
     valor i n (ddb_or d1 d2) = (valor i n d1  valor i n d2)
  Por ejemplo,
     ddb_not ddb1
     = N (N (H False) (H True)) (N (H True) (N (H True) (H False)))
  ----------------------------------------------------------------------
*}

definition ddb_not :: "ddb  ddb" where
  "ddb_not  ddb_op_un Not"

value "ddb_not ddb1"
-- "= N (N (H False) (H True)) (N (H True) (N (H True) (H False)))"

text {*
```

```
  ------------------------------------------------------------------
  Ejercicio 14. Demostrar que la definición de ddb_not es correcta;
  es decir,
     valor i n (ddb_not d) = (ň valor i n d)
  ------------------------------------------------------------------
*}

theorem ddb_not_correcta:
  "valor i n (ddb_not d) = (ň valor i n d)"
by (auto simp add: ddb_not_def ddb_op_un_correcto)

text {*
  ------------------------------------------------------------------
  Ejercicio 15. Definir la función
     xor :: "bool  bool  bool"
  tal que (xor x y) es la disyunción excluyente de x e y. Por ejemplo,
     xor True False = True
     xor True True  = False
  ------------------------------------------------------------------
*}

definition xor :: "bool  bool  bool" where
  "xor x y  (x  ňy)  (ňx  y)"

value "xor True False" -- "= True"
value "xor True True"  -- "= False"

text {*
  ------------------------------------------------------------------
  Ejercicio 16. Definir la función
     ddb_xor :: "ddb  ddb  ddb"
  tal que (ddb_xor d1 d2) es el diagrama correspondiente a la disyunción
  excluyente de los DDB d1 y d2. Por ejemplo,
     ddb_xor ddb1 (N (H True) (H False))
     = N (N (H True) (H True)) (N (H False) (N (H False) (H True)))
     ddb_xor ddb1 (N (H False) (H True))
     = N (N (H True) (H False)) (N (H True) (N (H True) (H True)))
  ------------------------------------------------------------------
*}
```

```
definition ddb_xor :: "ddb  ddb  ddb" where
  "ddb_xor  ddb_op_bin xor"

value "ddb_xor ddb1 (N (H True) (H False))"
-- "= N (N (H False) (H True)) (N (H False) (N (H False) (H True)))"
value "ddb_xor ddb1 (N (H False) (H True))"
-- "= N (N (H True) (H False)) (N (H True) (N (H True) (H False)))"

text {*
  ---------------------------------------------------------------------
  Ejercicio 17. Demostrar que la definición de ddb_xor es correcta;
  es decir,
     valor i n (ddb_xor d1 d2) = xor (valor i n d1) (valor i n d2)
  ---------------------------------------------------------------------
*}

theorem ddb_xor_correcta:
  "valor i n (ddb_xor d1 d2) = xor (valor i n d1) (valor i n d2)"
by (auto simp add: ddb_xor_def ddb_op_bin_correcto)

text {*
  ---------------------------------------------------------------------
  Ejercicio 18. Definir la función
     ddb_var :: "nat  ddb" where
  tal que (ddb_var n) es el diagrama equivalente a la variable p(n). Por
  ejemplo,
     ddb_var 0
     = N (H False) (H True)
     ddb_var 1
     = N (N (H False) (H True)) (N (H False) (H True))
  ---------------------------------------------------------------------
*}

fun ddb_var :: "nat  ddb" where
  "ddb_var 0       = N (H False) (H True)"
| "ddb_var (Suc i) = N (ddb_var i) (ddb_var i)"

value "ddb_var 0"
-- "= N (H False) (H True)"
value "ddb_var 1"
```

```
-- "= N (N (H False) (H True)) (N (H False) (H True))"

text {*
  ----------------------------------------------------------------------
  Ejercicio 19. Demostrar que la definición de ddb_var es correcta;
  es decir,
     "valor i 0 (ddb_var n) = i n
  ----------------------------------------------------------------------
*}

-- "Para probarlo, primero hay que generalizarlo"
theorem ddb_var_correcta:
  "valor i m (ddb_var n) = i (n+m)"
by (induct n arbitrary: m) auto

corollary ddb_var_correcta_2:
  "valor i 0 (ddb_var n) = i n"
by (simp add: ddb_var_correcta)

text {*
  ----------------------------------------------------------------------
  Ejercicio 20. Definir el tipo de las fórmulas proposicionales
  contruidas con la constante T, las variables (Var n) y las conectivas
  Not, And, Or y Xor.
  ----------------------------------------------------------------------
*}

datatype form = T
              | Var nat
              | Not form
              | And form form
              | Or  form form
              | Xor form form

text {*
  ----------------------------------------------------------------------
  Ejercicio 21. Definir la función
     valor_fla :: "(nat  bool)  form  bool"
  tal que (valor_fla i f) es el valor de la fórmula f en la
  interpretación i. Por ejemplo,
```

```
    valor_fla (n. True) (Xor T T)        = False
    valor_fla (n. False) (Xor T (Var 1)) = True
  ----------------------------------------------------------------------
*}

fun valor_fla :: "(nat  bool)  form  bool" where
  "valor_fla i T           = True"
| "valor_fla i (Var n)     = i n"
| "valor_fla i (Not f)     = (ñ(valor_fla i f))"
| "valor_fla i (And f1 f2) = (valor_fla i f1  valor_fla i f2)"
| "valor_fla i (Or f1 f2)  = (valor_fla i f1  valor_fla i f2)"
| "valor_fla i (Xor f1 f2) = xor (valor_fla i f1) (valor_fla i f2)"

value "valor_fla (n. True) (Xor T T)"        -- "= False"
value "valor_fla (n. False) (Xor T (Var 1))" -- "= True"

text {*
  ----------------------------------------------------------------------
  Ejercicio 22. Definir la función
    ddb_fla :: "form  ddb"
  tal que (ddb_fla f) es el DDB equivalente a la fórmula f; es decir,
    valor i 0 (ddb_fla f) = valor_fla i f
  ----------------------------------------------------------------------
*}

fun ddb_fla :: "form  ddb" where
  "ddb_fla T           = H True"
| "ddb_fla (Var i)     = ddb_var i"
| "ddb_fla (Not f)     = ddb_not (ddb_fla f)"
| "ddb_fla (And f1 f2) = ddb_and (ddb_fla f1) (ddb_fla f2)"
| "ddb_fla (Or  f1 f2) = ddb_or  (ddb_fla f1) (ddb_fla f2)"
| "ddb_fla (Xor f1 f2) = ddb_xor (ddb_fla f1) (ddb_fla f2)"

text {*
  ----------------------------------------------------------------------
  Ejercicio 23. Demostrar que la definición de ddb_fla es correcta; es
  decir,
    valor i 0 (ddb_fla f) = valor_fla i f
  ----------------------------------------------------------------------
*}
```

```
theorem ddb_fla_correcta:
  "valor e 0 (ddb_fla f) = valor_fla e f"
by (induct f) (auto simp add: ddb_var_correcta
                              ddb_and_correcta
                              ddb_or_correcta
                              ddb_not_correcta
                              ddb_xor_correcta)

text {*
  Referencias:
  ů S. Berghofer "Binary decision diagrams". En
       http://isabelle.in.tum.de/exercises/trees/bdd/ex.pdf
  ů J.A. Alonso, F.J. Martín y J.L. Ruiz "Diagramas de decisión
    binarios". En
       http://www.cs.us.es/cursos/lp-2005/temas/tema-07.pdf
  ů Wikipedia "Binary decision diagram". En
       http://en.wikipedia.org/wiki/Binary_decision_diagram
*}

end
```

6.2.5. Representación de fórmulas proposicionales mediante polinomios

```
chapter {* T6R2e: Representación de fórmulas proposicionales mediante
  polinomios *}

theory T6R2e
imports Main
begin

text {*
  El objetivo de esta relación es definir un procedimiento para
  transformar fórmulas proposicionales (construidas con ,  y ) en
  polinomios de la forma
     (p_1    p_n)    (q_1    q_m)
  y demostrar que, para cualquier interpretación I, el valor de las
  fórmulas coincide con la de su correspondiente polinomio. *}
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Las fórmulas proposicionales pueden definirse mediante
  las siguientes reglas:
  ů  es una fórmula proposicional
  ů Las variables proposicionales p_1, p_2,  son fórmulas
    proposicionales,
  ů Si F y G son fórmulas proposicionales, entonces (F  G) y (F  G)
    también lo son.
  donde  es una fórmula que siempre es verdadera,  es la conjunción y
   es la disyunción exclusiva.

  Definir el tipo de datos form para representar las fórmulas
  proposicionales usando
  ů T en lugar de ,
  ů (Var i) en lugar de p_i,
  ů (And F G) en lugar de (F  G) y
  ů (Xor F G) en lugar de (F  G).
  ---------------------------------------------------------------------- *}

datatype form = T | Var nat | And form form | Xor form form

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Los siguientes ejemplos de fórmulas
     form1 = p0
     form2 = (p0  p1)  (p0  p1)
  ---------------------------------------------------------------------- *}

abbreviation form1 :: "form" where
  "form1  Xor (Var 0) T"
abbreviation form2 :: "form" where
  "form2  Xor (Xor (Var 0) (Var 1)) (And (Var 0) (Var 1))"

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Definir la función
     xor :: bool  bool  bool
  tal que (xor p q) es el valor de la disyunción exclusiva de p y q. Por
  ejemplo,
```

```
    xor False True = True
  ---------------------------------------------------------------- *}

definition xor :: "bool ⇒ bool ⇒ bool" where
  "xor x y ≡ (x ∧ ¬y) ∨ (¬x ∧ y)"

value "xor False True" -- "= True"

text {*
  ---------------------------------------------------------------------
  Ejercicio 4. Una interpretación es una aplicación de los naturales en
  los booleanos. Definir las siguientes interpretaciones
         | p0 | p1 | p2 | p3 | ...
    int1 | F  | F  | F  | F  | ...
    int2 | F  | V  | F  | F  | ...
    int3 | V  | F  | F  | F  | ...
    int3 | V  | V  | F  | F  | ...
  ---------------------------------------------------------------- *}

abbreviation int1 :: "nat ⇒ bool" where
  "int1 x ≡ False"
abbreviation int2 :: "nat ⇒ bool" where
  "int2 ≡ int1 (1 := True)"
abbreviation int3 :: "nat ⇒ bool" where
  "int3 ≡ int1 (0 := True)"
abbreviation int4 :: "nat ⇒ bool" where
  "int4 ≡ int1 (0 := True, 1 := True)"

text {*
  ---------------------------------------------------------------------
  Ejercicio 5. Dada una interpretación I, el valor de de una fórmula F
  repecto de I, I(F), se define por
  ∙ T, si F es ;
  ∙ I(n), si F es p_n;
  ∙ I(G) ∧ I(H), si F es (G ∧ H);
  ∙ I(G) ∨ I(H), si F es (G ∨ H).

  Definir la función
     valorF :: (nat ⇒ bool) ⇒ form ⇒ bool
  tal que (valorF i f) es el valor de la fórmula f respecto de la
```

```
  interpretación i. Por ejemplo,
     valorF int1 form1 = True
     valorF int3 form1 = False
     valorF int1 form2 = False
     valorF int2 form2 = True
     valorF int3 form2 = True
     valorF int4 form2 = True
  ---------------------------------------------------------------------- *}

fun valorF :: "(nat  bool)  form  bool" where
  "valorF i T = True"
| "valorF i (Var n) = i n"
| "valorF i (And f g) = (valorF i f  valorF i g)"
| "valorF i (Xor f g) = xor (valorF i f) (valorF i g)"

value "valorF int1 form1" -- "= True"
value "valorF int3 form1" -- "= False"
value "valorF int1 form2" -- "= False"
value "valorF int2 form2" -- "= True"
value "valorF int3 form2" -- "= True"
value "valorF int4 form2" -- "= True"

text {*
  ----------------------------------------------------------------------
  Ejercicio 6. Un monomio es una lista de números naturales y se puede
  interpretar como la conjunción de variables proposionales cuyos
  índices son los números de la lista. Por ejemplo, el monomio [0,2,1]
  se interpreta como la fórmula (p0  p2  p1).

  Definir la función
     formM :: nat list  form
  tal que (formM m) es la fórmula correspondiente al monomio. Por
  ejemplo,
     formM [0,2,1] = And (Var 0) (And (Var 2) (And (Var 1) T))
  ---------------------------------------------------------------------- *}

fun formM :: "nat list  form" where
  "formM []     = T"
| "formM (n#ns) = And (Var n) (formM ns)"
```

```
value "formM [0,2,1]" -- "= And (Var 0) (And (Var 2) (And (Var 1) T)"

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Definir, por recursión, la función
     valorM :: (nat  bool)  nat list  bool
  tal que (valorM i m) es el valor de la fórmula representada por el
  monomio m en la interpretación i. Por ejemplo,
     valorM int1 [0,2,1]                              = False
     valorM (int1(0:=True,1:=True,2:=True)) [0,2,1] = True
  Demostrar que, para toda interpretación i y todo monomio m, se tiene
  que
     valorM i m = valorF i (formM m)
  ---------------------------------------------------------------------- *}

fun valorM :: "(nat  bool)  nat list  bool" where
  "valorM i []       = True"
| "valorM i (x # xs) = (i x  valorM i xs)"

value "valorM int1 [0,2,1]"                              -- "= False"
value "valorM (int1(0:=True,1:=True,2:=True)) [0,2,1]" -- "= True"

-- "La demostración automática es"
lemma correccion_valorM:
  "valorM i m = valorF i (formM m)"
by (induct m) auto

-- "La demostración estructurada es"
lemma correccion_valorM2:
  "valorM i m = valorF i (formM m)"
proof (induct m)
  show "valorM i [] = valorF i (formM [])" by simp
next
  fix x xs assume HI: "valorM i xs = valorF i (formM xs)"
  have "valorM i (x#xs) = (i x  valorM i xs)" by simp
  also have "... = (i x  (valorF i (formM xs)))" using HI by simp
  also have "... = ((valorF i (Var x))  (valorF i (formM xs)))" by simp
  also have "... = valorF i (And (Var x) (formM xs))" by simp
  also have "... = valorF i (formM (x#xs))" by simp
  finally show "valorM i (x#xs) = valorF i (formM (x#xs))" by simp
```

```
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 8. Un polinomio es una lista de monomios y se puede
  interpretar como la disyunción exclusiva de los monomios. Por ejemplo,
  el polinomio [[0,2,1],[1,3]] se interpreta como la fórmula
  (p0  p2  p1)  (p1  p3).

  Definir la función
     formP :: nat list list  form
  tal que (formP p) es la fórmula correspondiente al polinomio p. Por
  ejemplo,
     formP [[1,2],[3]]
     = Xor (And (Var 1) (And (Var 2) T)) (Xor (And (Var 3) T) (Xor T T))
  ---------------------------------------------------------------------- *}

fun formP :: "nat list list  form" where
  "formP []     = (Xor T T)"
| "formP (m#ms) = (Xor (formM m) (formP ms))"

value "formP [[1,2],[3]]"
-- "= Xor (And (Var 1) (And (Var 2) T)) (Xor (And (Var 3) T) (Xor T T))"

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Definir la función
     valorP :: (nat  bool)  nat list list  bool
  tal que (valorP i p) es el valor de la fórmula representada por el
  polinomio p en la interpretación i. Por ejemplo,
     valorP (int1(1:=True,3:=True)) [[0,2,1],[1,3]] = True
  Demostrar que, para toda interpretación i y todo polinomio p, se tiene
  que
     valorM i p = valorF i (formP p)
  ---------------------------------------------------------------------- *}

fun valorP :: "(nat  bool)  nat list list  bool" where
  "valorP i []       = False"
| "valorP i (m # ms) = xor (valorM i m) (valorP i ms)"
```

```
value "valorP (int1(1:=True,3:=True)) [[0,2,1],[1,3]]" -- "= True"

-- "La demostración automática es"
lemma correccion_valorP:
  "valorP i p = valorF i (formP p)"
by (induct p) (auto simp add: xor_def correccion_valorM)

-- "La demostración estructurada es"
lemma correccion_valorP2:
  "valorP i p = valorF i (formP p)"
proof (induct p)
  show "valorP i [] = valorF i (formP [])" by (simp add: xor_def)
next
  fix m ms assume HI: "valorP i ms = valorF i (formP ms)"
  have "valorP i (m#ms) = xor (valorM i m) (valorP i ms)" by simp
  also have "... = xor (valorM i m) (valorF i (formP ms))" using HI by simp
  also have "... = xor (valorF i (formM m)) (valorF i (formP ms))"
    by (simp add: correccion_valorM)
  also have "... = valorF i (Xor (formM m) (formP ms))" by simp
  also have "... = valorF i (formP (m#ms))" by simp
  finally show "valorP i (m#ms) = valorF i (formP (m#ms))" by simp
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 10. Definir la función
     productoM :: nat list  nat list list  nat list list
  tal que (productoM m p) es el producto del monomio p por el polinomio
  p. Por  ejemplo,
     productoM [1,3] [[1,2,4],[7],[4,1]]
     = [[1,3,1,2,4],[1,3,7],[1,3,4,1]]
  --------------------------------------------------------------------- *}

fun productoM :: "nat list  nat list list  nat list list" where
  "productoM m []     = []"
| "productoM m (x#xs) = (m@x)#(productoM m xs)"

value "productoM [1,3] [[1,2,4],[7],[4,1]]"
-- "= [[1,3,1,2,4],[1,3,7],[1,3,4,1]]"
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 11. Demostrar que, en cualquier interpretación i, el valor
  de la concatenación de dos monomios es la conjunción de sus valores.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma valorM_conc:
  "valorM i (xs @ ys) = (valorM i xs  valorM i ys)"
by (induct xs) auto

-- "La demostración estructurada es"
lemma valorM_conc2:
  "valorM i (xs @ ys) = (valorM i xs  valorM i ys)"
proof (induct xs)
  show "valorM i ([] @ ys) = (valorM i []  valorM i ys)" by simp
next
  fix x xs assume HI: "valorM i (xs @ ys) = (valorM i xs  valorM i ys)"
  have "valorM i ((x#xs) @ ys) = valorM i (x#(xs@ys))" by simp
  also have "... = (i x  (valorM i (xs@ys)))" by simp
  also have "... = (i x  (valorM i xs  valorM i ys))" using HI by simp
  also have "... = ((i x  valorM i xs)  valorM i ys)" by simp
  also have "... = ((valorM i (x#xs))  valorM i ys)" by simp
  finally show "valorM i ((x#xs) @ ys) =
                ((valorM i (x#xs))  valorM i ys)" by simp
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Demostrar que, en cualquier interpretación i, el valor
  del producto de un monomio por un polinomio es la conjunción de sus
  valores.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma correccion_productoM:
  "valorP i (productoM m p) = (valorM i m  valorP i p)"
by (induct p) (auto simp add: xor_def valorM_conc)

-- "La demostración estructurada es"
```

```
lemma correccion_productoM2:
  "valorP i (productoM m p) = (valorM i m  valorP i p)"
proof (induct p)
  show "valorP i (productoM m []) = (valorM i m  valorP i [])" by simp
next
  fix xs p
  assume HI: "valorP i (productoM m p) = (valorM i m  valorP i p)"
  have "valorP i (productoM m (xs#p)) = valorP i ((m@xs)#(productoM m p))"
    by simp
  also have "... = xor (valorM i (m@xs)) (valorP i (productoM m p))"
    by simp
  also have "... = xor (valorM i (m@xs)) (valorM i m  valorP i p)"
    using HI by simp
  also have "... = xor (valorM i m  valorM i xs)
                       (valorM i m  valorP i p)"
    by (simp add: valorM_conc)
  also have "... = (valorM i m  (xor (valorM i xs) (valorP i p)))"
    by (auto simp add: xor_def)
  also have "... = (valorM i m  valorP i (xs#p))" by simp
  finally show "valorP i (productoM m (xs#p)) =
                (valorM i m  valorP i (xs#p))" by simp
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 13. Definir la función
     producto :: nat list list  nat list list  nat list list
  tal que (producto p q) es el producto de los polinomios p y q. Por
  ejemplo,
     producto [[1,3],[2]] [[1,2,4],[7],[4,1]]
     = [[1,3,1,2,4],[1,3,7],[1,3,4,1],[2,1,2,4],[2,7],[2,4,1]]
  --------------------------------------------------------------------- *}

fun producto :: "nat list list  nat list list  nat list list" where
  "producto [] q      = []"
| "producto (m # p) q = (productoM m q) @ (producto p q)"

value "producto [[1,3],[2]] [[1,2,4],[7],[4,1]]"
-- "= [[1,3,1,2,4],[1,3,7],[1,3,4,1],[2,1,2,4],[2,7],[2,4,1]]"
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 14. Demostrar que, en cualquier interpretación i, el valor
  de la concatenación de dos polinomios es la disyunción exclusiva de
  sus valores.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma valorP_conc:
  "valorP i (xs @ ys) = (xor (valorP i xs) (valorP i ys))"
by (induct xs) (auto simp add: xor_def)

-- "La demostración estructurada es"
lemma valorP_conc2:
  "valorP i (xs @ ys) = (xor (valorP i xs) (valorP i ys))"
proof (induct xs)
  show "valorP i ([]@ys) = xor (valorP i []) (valorP i ys)"
    by (simp add: xor_def)
next
  fix x xs
  assume HI: "valorP i (xs@ys) = xor (valorP i xs) (valorP i ys)"
  have "valorP i ((x#xs)@ys) = valorP i (x#(xs@ys))" by simp
  also have "... = xor (valorM i x) (valorP i (xs@ys))" by simp
  also have "... = xor (valorM i x) (xor (valorP i xs) (valorP i ys))"
    using HI by simp
  also have "... = xor (xor (valorM i x) (valorP i xs)) (valorP i ys)"
    by (auto simp add: xor_def)
  also have "... = xor (valorP i (x#xs)) (valorP i ys)"
    by simp
  finally show "valorP i ((x#xs)@ys) = xor (valorP i (x#xs)) (valorP i ys)"
    by simp
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 15. Demostrar que, en cualquier interpretación i, el valor
  del producto de dos polinomios es la conjunción de sus valores.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
```

```
lemma correccion_producto:
  "valorP i (producto p q) = (valorP i p  valorP i q)"
by (induct p)
   (auto simp add: xor_def correccion_productoM valorP_conc)

-- "La demostración estructurada es"
lemma correccion_producto2:
  "valorP i (producto p q) = (valorP i p  valorP i q)"
proof (induct p)
  show "valorP i (producto [] q) = (valorP i []  valorP i q)"
    by simp
next
  fix m p
  assume HI: "valorP i (producto p q) = (valorP i p  valorP i q)"
  have "valorP i (producto (m # p) q) =
        valorP i ((productoM m q) @ (producto p q))" by simp
  also have "... = xor (valorP i (productoM m q))
                       (valorP i (producto p q))"
    by (simp add: valorP_conc)
  also have "... = xor (valorM i m  valorP i q)
                       (valorP i (producto p q))"
    by (simp add: correccion_productoM)
  also have "... = xor (valorM i m  valorP i q)
                       (valorP i p  valorP i q)"
    by (simp add: correccion_producto)
  also have "... = (xor (valorM i m) (valorP i p)  valorP i q)"
    by (auto simp add: xor_def)
  also have "... = (valorP i (m # p)  valorP i q)"
    by simp
  finally show "valorP i (producto (m # p) q) =
                (valorP i (m # p)  valorP i q)" by simp
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 16. Definir la función
     polinomio :: form  nat list list
  tal que (polinomio f) es el polinomio que representa la fórmula f. Por
  ejemplo,
     polinomio (Xor (Var 1) (Var 2))               = [[1],[2]]
```

```
    polinomio (And (Var 1) (Var 2))                = [[1,2]]
    polinomio (Xor (Var 1) T)                      = [[1],[]]
    polinomio (And (Var 1) T)                      = [[1]]]
    polinomio (And (Xor (Var 1) (Var 2)) (Var 3)) = [[1,3],[2,3]]
    polinomio (Xor (And (Var 1) (Var 2)) (Var 3)) = [[1,2],[3]]
  ------------------------------------------------------------------- *}

fun polinomio :: "form  nat list list" where
  "polinomio T           = [[]]"
| "polinomio (Var i)     = [[i]]"
| "polinomio (Xor f1 f2) = polinomio f1 @ polinomio f2"
| "polinomio (And f1 f2) = producto (polinomio f1) (polinomio f2)"

value "polinomio (Xor (Var 1) (Var 2))"                -- "= [[1],[2]]"
value "polinomio (And (Var 1) (Var 2))"                -- "= [[1,2]]"
value "polinomio (Xor (Var 1) T)"                      -- "= [[1],[]]"
value "polinomio (And (Var 1) T)"                      -- "= [[1]]]"
value "polinomio (And (Xor (Var 1) (Var 2)) (Var 3))" -- "= [[1,3],[2,3]]"
value "polinomio (Xor (And (Var 1) (Var 2)) (Var 3))" -- "= [[1,2],[3]]"

text {*
  ---------------------------------------------------------------------
  Ejercicio 17. Demostrar que, en cualquier interpretación i, el valor
  de f es igual que el de su polinomio.
  ------------------------------------------------------------------- *}

-- "La demostración automática es"
theorem correccion_polinomio:
  "valorF i f = valorP i (polinomio f)"
by (induct f)
   (auto simp add: xor_def correccion_producto valorP_conc)

-- "La demostración estructurada es"
theorem correccion_polinomio2:
  "valorF i f = valorP i (polinomio f)"
proof (induct f)
  show "valorF i T = valorP i (polinomio T)" by (simp add: xor_def)
next
  fix n
  show "valorF i (Var n) = valorP i (polinomio (Var n))"
```

```
    by (simp add: xor_def)
next
  fix f g
  assume HI1: "valorF i f = valorP i (polinomio f)" and
         HI2: "valorF i g = valorP i (polinomio g)"
  have "valorF i (And f g) = (valorF i f  valorF i g)" by simp
  also have "... = (valorP i (polinomio f)  valorP i (polinomio g))"
    using HI1 HI2 by simp
  also have "... = valorP i (producto (polinomio f) (polinomio g))"
    by (simp add: correccion_producto)
  also have "... = valorP i (polinomio (And f g))"
    by simp
  finally show "valorF i (And f g) = valorP i (polinomio (And f g))"
    by simp
next
  fix f g
  assume HI1: "valorF i f = valorP i (polinomio f)" and
         HI2: "valorF i g = valorP i (polinomio g)"
  have "valorF i (Xor f g) = xor (valorF i f) (valorF i g)"
    by simp
  also have "... = xor (valorP i (polinomio f)) (valorP i (polinomio g))"
    using HI1 HI2 by simp
  also have "... = valorP i ((polinomio f) @ (polinomio g))"
    by (simp add: valorP_conc)
  also have "... = valorP i (polinomio (Xor f g))"
    by simp
  finally show "valorF i (Xor f g) = valorP i (polinomio (Xor f g))"
    by simp
qed

end
```

6.2.6. Ordenación de listas mediante árboles ordenados

```
chapter {* T6R2f: Ordenación de listas mediante árboles ordenados *}

theory T6R2f_Ordenacion_de_listas_mediante_arboles_ordenados
imports Main T6R1j_Ordenacion_de_listas_por_insercion
begin

text {*
```

```
  En esta relación de ejercicios se definen los árboles ordenados y se
  muestra cómo se puede ordenar listas mediante árboles. Para ello, se
  definen dos funciones
  * arbol_de que transforma lista en árboles ordenados y
  * lista_de que transforma árboles en listas
  Se demuestra que (lista_de (arbol_de xs)) es una lista ordenada con
  los mismos elementos que xs.
*}

section {* Árboles ordenados *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir el tipo de datos arbol para representar los
  árboles binarios que tiene en los nodos números naturales. Por
  ejemplo, el árbol
          4
         / \
        /   \
       1     6
      / \   / \
     H   H H   H
  se representa por "Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja)".
  ----------------------------------------------------------------------
*}

datatype arbol = Hoja | Nodo nat arbol arbol

value "Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja)"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Definir la función
     menorA :: nat  arbol  bool
  tal que (menorA x a) se verifica si x es menor o igual que todos los
  elementos del árbol a. Por ejemplo,
     menorA 2 (Nodo 4 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja)) = True
     menorA 5 (Nodo 4 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja)) = False
  ---------------------------------------------------------------------- *}
```

```
fun menorA :: "nat  arbol  bool" where
  "menorA x Hoja           = True"
| "menorA x (Nodo n a1 a2) = (x  n  menorA x a1  menorA x a2)"

value "menorA 2 (Nodo 4 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja))"
-- "= True"
value "menorA 5 (Nodo 4 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja))"
-- "= False"

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Definir la función
     mayorA :: nat  arbol  bool
  tal que (mayorA x a) se verifica si x es mayor o igual que todos los
  elementos del árbol a. Por ejemplo,
     mayorA 7 (Nodo 4 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja)) = True
     mayorA 5 (Nodo 4 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja)) = False
  ---------------------------------------------------------------------- *}

fun mayorA :: "nat  arbol  bool" where
  "mayorA x Hoja           = True"
| "mayorA x (Nodo n a1 a2) = (n  x  mayorA x a1  mayorA x a2)"

value "mayorA 7 (Nodo 4 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja))"
-- "= True"
value "mayorA 5 (Nodo 4 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja))"
-- "= False"

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Un árbol está ordenado si cada nodo es mayayor igual que
  todos los nodos de su subárbol izquierdo y menor o igual que todos los
  nodos de su subárbol derecho.

  Definir la función
     ordenadoA :: arbol  bool
  tal que (ordenadoA x a) se verifica si a es es un árbol ordenado. Por
  ejemplo,
     ordenadoA (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja))
     = True
```

```
     ordenadoA (Nodo 4 (Nodo 5 Hoja Hoja) (Nodo 6 Hoja Hoja))
     = False
     ordenadoA (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 3 Hoja Hoja))
     = False
  ------------------------------------------------------------------ *}

fun ordenadoA :: "arbol  bool" where
  "ordenadoA Hoja           = True"
| "ordenadoA (Nodo n a1 a2) = (ordenadoA a1  ordenadoA a2
                               mayorA n a1  menorA n a2)"

value "ordenadoA (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja))"
-- "= True"
value "ordenadoA (Nodo 4 (Nodo 5 Hoja Hoja) (Nodo 6 Hoja Hoja))"
-- "= False"
value "ordenadoA (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 3 Hoja Hoja))"
-- "= False"

text {*
  ------------------------------------------------------------------------
  Ejercicio 6. Definir la función
     insertaA :: nat  arbol  arbol
  tal que (inserta n a) es el árbol obtenido insertando n delante del
  primer subárbol de a cuya raiz es mayor o igual que a. Por ejemplo,
     insertaA 4 Hoja
     = Nodo 4 Hoja Hoja
     insertaA 1 (Nodo 4 Hoja Hoja)
     = Nodo 4 (Nodo 1 Hoja Hoja) Hoja
     insertaA 6 (Nodo 4 (Nodo 1 Hoja Hoja) Hoja)
     = Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja)
     insertaA 3 (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja))
     = Nodo 4 (Nodo 1 Hoja (Nodo 3 Hoja Hoja)) (Nodo 6 Hoja Hoja)
     insertaA 5 (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja))
     = Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 (Nodo 5 Hoja Hoja) Hoja)
  ------------------------------------------------------------------ *}

fun insertaA :: "nat  arbol  arbol" where
  "insertaA x Hoja           = Nodo x Hoja Hoja"
| "insertaA x (Nodo n a1 a2) = (if x  n
                                then Nodo n (insertaA x a1) a2
```

```
                                else Nodo n a1 (insertaA x a2))"

value "insertaA 4 Hoja"
-- "= Nodo 4 Hoja Hoja"
value "insertaA 1 (Nodo 4 Hoja Hoja)"
-- "= Nodo 4 (Nodo 1 Hoja Hoja) Hoja"
value "insertaA 6 (Nodo 4 (Nodo 1 Hoja Hoja) Hoja)"
-- "= Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja)"
value "insertaA 3 (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja))"
-- "= Nodo 4 (Nodo 1 Hoja (Nodo 3 Hoja Hoja)) (Nodo 6 Hoja Hoja)"
value "insertaA 5 (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja))"
-- "= Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 (Nodo 5 Hoja Hoja) Hoja)"

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Definir la función
     arbol_de :: nat list  arbol
  tal que (arbol_de xs) es el árbol obtenido insertando n los elementos
  de la lista xs. Por ejemplo,
     arbol_de [6,1,4]
     = Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja)
  ---------------------------------------------------------------------- *}

fun arbol_de :: "nat list  arbol" where
  "arbol_de []     = Hoja"
| "arbol_de (x#xs) = insertaA x (arbol_de xs)"

value "arbol_de [6,1,4]"
-- "= Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja)"

text {*
  ----------------------------------------------------------------------
  Ejercicio 8. Demostrar que x es mayor o igual que los nodos del árbol
  obtenido insertando y en el árbol a syss x es mayor o igual que y y x
  es mayor o igual que los nodos del árbol a.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma mayorA_insertaA:
  "mayorA x (insertaA y a) = (y  x  mayorA x a)"
```

```
by (induct a) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Demostrar que x es menor o igual que los nodos del árbol
  obtenido insertando y en el árbol a syss x es menor o igual que y y x
  es menor o igual que los nodos del árbol a.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma menorA_insertaA:
  "menorA x (insertaA y a) = (x  y  menorA x a)"
by (induct a) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Demostrar que el árbol obtenido insertando x en el árbol
  a es ordenado syss el árbol a es ordenado.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma ordenadoA_insertaA:
  "ordenadoA (insertaA x a) = ordenadoA a"
by (induct a) (auto simp add: mayorA_insertaA menorA_insertaA)

text {*
  ----------------------------------------------------------------------
  Ejercicio 11. Demostrar que para cualquier lista xs, (arbol_de xs) es
  un árbol ordenado.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
theorem ordenadoA_arbol_de:
  "ordenadoA (arbol_de xs)"
by (induct xs) (auto simp add: ordenadoA_insertaA)

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Definir la función
     cuentaA :: arbol => nat => nat
```

```
  tal que (cuentaA a y) es el número de veces que aparece y en el árbol
  a. Por ejemplo,
     cuentaA (Nodo 6 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja)) 6 = 2
  ----------------------------------------------------------------- *}

fun cuentaA :: "arbol => nat => nat" where
  "cuentaA Hoja           y = 0"
| "cuentaA (Nodo x a1 a2) y = (if x=y
                               then Suc (cuentaA a1 y + cuentaA a2 y)
                               else cuentaA a1 y + cuentaA a2 y)"

value "cuentaA (Nodo 6 (Nodo 3 Hoja Hoja) (Nodo 6 Hoja Hoja)) 6"
-- "= 2"

text {*
  ------------------------------------------------------------------
  Ejercicio 13. Demostrar que el número de veces que aparece y en el
  árbol obtenido insertando x en el árbol a es
  * uno más el número de veces que aparece y en a, si y = x,
  * el número de veces que aparece y en a, si y  x,
  ----------------------------------------------------------------- *}

-- "La demostración automática es"
lemma cuentaA_insertaA:
  "cuentaA (insertaA x a) y =
   (if x=y then Suc (cuentaA a y) else cuentaA a y)"
by (induct a) auto

text {*
  ------------------------------------------------------------------
  Ejercicio 14. Demostrar que el número de veces que aparece y en
  (arbol_de xs) es el mismo que el número de veces que aparece y en xs.
  ----------------------------------------------------------------- *}

-- "La demostración automática es"
theorem cuenta_arbol_de:
  "cuentaA (arbol_de xs) x = cuenta xs x"
by (induct xs) (auto simp add: cuentaA_insertaA)

section {* Ordenación de lista mediante árboles *}
```

```
text {*
  La función arbol_de transforma una lista en un árbol ordenado. Vamos a
  definir la función lista_de que transforme un árbol ordenado en una
  lista ordenada. Con ambas, tendremos un algoritmo de ordenación de
  listas.
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 15. Definir la función
     lista_de :: arbol  nat list
  tal que (lista_de a) es la lista obtenida recorriendo el árbol a de
  forma prefija. Por ejemplo,
     lista_de (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja))
     = [1,4,6]
  ---------------------------------------------------------------------- *}

fun lista_de :: "arbol  nat list" where
  "lista_de Hoja           = []"
| "lista_de (Nodo n a1 a2) = lista_de a1 @ [n] @ lista_de a2"

value "lista_de (Nodo 4 (Nodo 1 Hoja Hoja) (Nodo 6 Hoja Hoja))"
-- "= [1,4,6]"

text {*
  ----------------------------------------------------------------------
  Ejercicio 16. Demostrar que x es menor o igual que todos elmentos de
  la concatenación de dos sistas syss es menor o igual que todos los
  elementos de cada una de las listas.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma menor_conc:
  "menor x (a@b) = (menor x a  menor x b)"
by (induct a) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 17. Definir la función
```

```
    mayor :: nat ⇒ nat list ⇒ bool
  tal que (mayor x ys) se verifica si x es mayor o igual que todos los
  elementos de ys. Por ejemplo,
  ------------------------------------------------------------------ *}

fun mayor :: "nat ⇒ nat list ⇒ bool" where
  "mayor a []     = True"
| "mayor a (x#xs) = (x ≤ a ∧ mayor a xs)"

value "mayor 7 [1,5,2]" --"= True"
value "mayor 7 [1,9,2]" --"= False"

text {*
  ------------------------------------------------------------------
  Ejercicio 18. Demostrar que x es mayor o igual que todos elmentos de
  la concatenación de dos sistas syss es mayor o igual que todos los
  elementos de cada una de las listas.
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma mayor_conc:
  "mayor x (a@b) = (mayor x a ∧ mayor x b)"
by (induct a) auto

text {*
  ------------------------------------------------------------------
  Ejercicio 18. Demostrar que la lista (a@(x#b)) está ordenada syss se
  a y b está ordenada, x es mayor o igual que los elementos de a y es
  menor o igual que los elementos de b.
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma ordenada_conc:
  "ordenada (a@(x#b)) = (ordenada a ∧ ordenada b ∧ mayor x a ∧ menor x b)"
by (induct a) (auto simp add: menor_conc mayor_conc menor_menor)

text {*
  ------------------------------------------------------------------
  Ejercicio 19. Demostrar que n es mayor o igual que todos los elementos
  de la lista de elementos del árbol a syss es mayor o igual que todos
```

```
  los elementos de a.
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma mayor_lista_de:
  "mayor n (lista_de a) = mayorA n a"
by (induct a) (auto simp add: mayorA_insertaA mayor_conc)

text {*
  ------------------------------------------------------------------------
  Ejercicio 20. Demostrar que n es menor o igual que todos los elementos
  de la lista de elementos del árbol a syss es menor o igual que todos
  los elementos de a.
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma menor_lista_de:
  "menor n (lista_de a) = menorA n a"
by (induct a) (auto simp add: menorA_insertaA menor_conc)

text {*
  ------------------------------------------------------------------------
  Ejercicio 21. Demostrar que (lista_de a) está ordenada syss a es un
  árbol ordenado.
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma ordenada_lista_de:
  "ordenada (lista_de a) = ordenadoA a"
by (induct a) (auto simp add: ordenada_conc mayor_lista_de menor_lista_de)

text {*
  ------------------------------------------------------------------------
  Ejercicio 22. Demostrar que (lista_de (arbol_de xs)) está ordenada.
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
theorem ordenada_lista_de_arbol_de:
  "ordenada (lista_de (arbol_de xs))"
by (auto simp add: ordenada_lista_de ordenadoA_arbol_de)
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 23. Demostrar que el número de veces que aparece un elemento
  en la concatenación de dos listas es la suma del número de veces que
  aparece en cada una.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma cuenta_conc:
  "cuenta (a@b) n = cuenta a n + cuenta b n"
by (induct a) auto

text {*
  ----------------------------------------------------------------------
  Ejercicio 24. Demostrar que el número de veces que aparece n en
  (lista_de a) es el mismo que el número de veces que aprece n en el
  árbol a.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma cuenta_lista_de:
  "cuenta (lista_de a) n = cuentaA a n"
by (induct a) (auto simp add: cuenta_conc)

text {*
  ----------------------------------------------------------------------
  Ejercicio 25. Demostrar que el número de veces que aparece n
  (lista_de (arbol_de xs)) es el mismo número de veces que aparece n en
  xs.
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
theorem cuenta_lista_de_arbol_de:
  "cuenta (lista_de (arbol_de xs)) n = cuenta xs n"
by (induct xs) (auto simp add: cuenta_lista_de cuentaA_insertaA)

end
```

6.3. Ejercicios sobre aritmética

6.3.1. Potencias y sumatorios

```
chapter {* T6R3a: Potencias y sumatorios *}

theory T6R3a
imports Main
begin

text {*
  En esta relación se definen las potencias y sumatorios y se demuestran
  algunas de sus propiedades. *}

section {* Potencia *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     potencia :: nat => nat => nat
  tal que (potencia x n) es x elvado a n. Por ejemplo,
     potencia 2 3 = 8
  ---------------------------------------------------------------------- *}

fun potencia :: "nat => nat => nat" where
  "potencia x 0       = Suc 0"
| "potencia x (Suc n) = x * potencia x n"

value "potencia 2 3" -- "= 8"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar que
     x\<^bsup>mn\<^esup> = (x\<^bsup>m\<^esup>)\<^bsup>n\<^esup>
  Indicación: Demostrar antes un lema.
  ---------------------------------------------------------------------- *}

-- "La demostración automática del lema es"
lemma potencia_suma:
  "potencia x (m + n) = potencia x m * potencia x n"
by (induct n) auto
```

```
-- "La demostración estructurada del lema es"
lemma potencia_suma2:
  "potencia x (m + n) = potencia x m * potencia x n"
proof (induct n)
  show "potencia x (m + 0) = potencia x m * potencia x 0" by simp
next
  fix n
  assume HI: "potencia x (m + n) = potencia x m * potencia x n"
  have "potencia x (m + Suc n) = potencia x (Suc (m + n))" by simp
  also have "... = x * potencia x (m + n)" by simp
  also have "... = x * (potencia x m * potencia x n)" using HI by simp
  also have "... = potencia x m * (x * potencia x n)" by simp
  also have "... = potencia x m * potencia x (Suc n)" by simp
  finally show "potencia x (m + Suc n) = potencia x m * potencia x (Suc n)"
    by simp
qed

-- "La demostración automática del teorema es"
theorem potencia_mult:
  "potencia x (m * n) = potencia (potencia x m) n"
by (induct n) (auto simp add: potencia_suma)

-- "La demostración estructurada del teorema es"
theorem potencia_mult2:
  "potencia x (m * n) = potencia (potencia x m) n"
proof (induct n)
  show "potencia x (m * 0) = potencia (potencia x m) 0" by simp
next
  fix n
  assume HI: "potencia x (m * n) = potencia (potencia x m) n"
  have "potencia x (m * Suc n) = potencia x (m + m * n)" by simp
  also have "... = potencia x m * potencia x (m * n)"
    by (simp add: potencia_suma)
  also have "... = potencia x m * potencia (potencia x m) n"
    using HI by simp
  also have "... = potencia (potencia x m) (Suc n)"
    by simp
  finally show "potencia x (m * Suc n) = potencia (potencia x m) (Suc n)"
    by simp
```

```
qed

section {* Sumatorios *}

text {*
  ---------------------------------------------------------------------
  Ejercicio 3. Definir la función
     suma :: nat list => nat
  tal que (suma ns) es la suma de los elementos de ns. Por ejemplo,
     suma [3,2,5] = 10
  --------------------------------------------------------------------- *}

fun suma :: "nat list => nat" where
  "suma []     = 0"
| "suma (n#ns) = n + suma ns"

value "suma [3,2,5]" -- "= 10"

text {*
  ---------------------------------------------------------------------
  Ejercicio 4. Demostrar que
     suma (xs @ ys) = suma xs + suma ys
  Indicación: Conviene demostrar un lema previo.
  --------------------------------------------------------------------- *}

-- "La demostración automática del lema es"
lemma suma_append:
  "suma (xs @ ys) = suma xs + suma ys"
by (induct xs) auto

-- "La demostración estructurada del lema es"
lemma suma_append2:
  "suma (xs @ ys) = suma xs + suma ys"
proof (induct xs)
  show "suma ([] @ ys) = suma [] + suma ys" by simp
next
  fix x xs
  assume HI: "suma (xs @ ys) = suma xs + suma ys"
  have "suma ((x#xs) @ ys) = suma (x # (xs @ ys))" by simp
  also have "... = x + suma (xs @ ys)" by simp
```

```
  also have "... = x + (suma xs + suma ys)" using HI by simp
  also have "... = (x + suma xs) + suma ys" by simp
  also have "... = suma (x#xs) + suma ys" by simp
  finally show "suma ((x#xs) @ ys) = suma (x#xs) + suma ys" by simp
qed

-- "La demostración automática del teorema es"
theorem suma_rev: "
  suma (rev ns) = suma ns"
by (induct ns) (auto simp add: suma_append)

-- "La demostración estructurada del teorema es"
theorem suma_rev2: "
  suma (rev ns) = suma ns"
proof (induct ns)
  show "suma (rev []) = suma []" by simp
next
  fix n ns
  assume HI: "suma (rev ns) = suma ns"
  have "suma (rev (n#ns)) =  suma (rev ns @ [n])" by simp
  also have "... = suma (rev ns) + suma [n]" by (simp add: suma_append)
  also have "... = suma ns + suma [n]" using HI by simp
  also have "... = n + suma ns" by simp
  also have "... = suma (n#ns)" by simp
  finally show "suma (rev (n#ns)) = suma (n#ns)" by simp
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Definir la función
     Suma :: (nat => nat) => nat => nat
  tal que (Suma f n) es la suma f(0) +  + f(n-1). Por ejemplo,
     Suma (x. x+2) 3 = 9
  ---------------------------------------------------------------------- *}

fun Suma :: "(nat => nat) => nat => nat" where
  "Suma f 0       = 0"
| "Suma f (Suc n) = Suma f n + f n"

value "Suma (x. x+2) 3" -- "= 9"
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 6. Demostrar que
     Suma (i. f i + g i) k = Suma f k + Suma g k
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma "Suma (i. f i + g i) k = Suma f k + Suma g k"
by (induct k) auto

-- "La demostración estructurada es"
lemma "Suma (i. f i + g i) k = Suma f k + Suma g k"
proof (induct k)
  show "Suma (i. f i + g i) 0 = Suma f 0 + Suma g 0" by simp
next
  fix k
  assume HI: "Suma (i. f i + g i) k = Suma f k + Suma g k"
  have "Suma (i. f i + g i) (Suc k) =
        Suma (i. f i + g i) k + (f k + g k)" by simp
  also have "... = (Suma f k + Suma g k) + (f k + g k)"
    using HI by simp
  also have "... = (Suma f k + f k) + (Suma g k + g k)" by simp
  also have "... = Suma f (Suc k) + Suma g (Suc k)" by simp
  finally show "Suma (i. f i + g i) (Suc k) =
                Suma f (Suc k) + Suma g (Suc k)" by simp
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Determinar el valor de indefinida para que se verifique
  la siguiente igualdad
     Suma f (k + l) = Suma f k + Suma indefinida l
  y demostrarla.
  ---------------------------------------------------------------------- *}

-- "El valor de indefinida es (i. f (k + i))"

-- "La demostración automática es"
lemma "Suma f (k + l) = Suma f k + Suma (i. f (k + i)) l"
```

```
by (induct l) auto

-- "La demostración estructurada es"
lemma "Suma f (k + l) = Suma f k + Suma (i. f (k + i)) l"
proof (induct l)
  show "Suma f (k + 0) = Suma f k + Suma (i. f (k + i)) 0" by simp
next
  fix l
  assume HI: "Suma f (k + l) = Suma f k + Suma (i. f (k + i)) l"
  have "Suma f (k + Suc l) = Suma f (Suc (k + l))" by simp
  also have "... = Suma f (k + l) + f (k + l)" by simp
  also have "... = Suma f k + Suma (i. f (k + i)) l + f (k + l)"
    using HI by simp
  also have "... = Suma f k + Suma (i. f (k + i)) (Suc l)" by simp
  finally show "Suma f (k + Suc l) =
                Suma f k + Suma (i. f (k + i)) (Suc l)" by simp
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 8. Determinar el valor de indefinida para que se verifique
  la siguiente igualdad
     Suma f k = suma indefinida
  y demostrarla.
  ---------------------------------------------------------------------- *}

-- "El valor de indefinida es (map f [0..<k])"

-- "La demostración automática es"
lemma "Suma f k = suma (map f [0..<k])"
by (induct k) (auto simp add: suma_append)

-- "La demostración estructurada es"
lemma "Suma f k = suma (map f [0..<k])" (is "?P k")
proof (induct k)
  show "?P 0" by simp
next
  fix k assume HI: "?P k"
  have "Suma f (Suc k) = Suma f k + f k" by simp
  also have "... = suma (map f [0..<k]) + f k" using HI by simp
```

```
  also have "... = suma (map f [0..<k]) + suma [f k]" by simp
  also have "... = suma (map f [0..<k] @ [f k])"
    by (simp add: suma_append)
  also have "... = suma (map f [0..<Suc k])" by simp
  finally show "?P (Suc k)" by simp
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Demostrar la fórmula de la suma de naturales:
     1 + 2 +  + n = (n(n+1))/2
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma suma_de_naturales:
  fixes n :: "nat"
  shows "2 * (i=1..n. i) = n * (n + 1)"
by (induct n) simp_all

-- "La demostración estructurada es"
lemma suma_de_naturales_2:
  fixes n :: "nat"
  shows "2 * (i=1..n. i) = n * (n + 1)" (is "?P n")
proof (induct n)
  show "?P 0" by simp
next
  fix n assume HI: "?P n"
  have "2 * {1..Suc n} = 2 * ({1..n} + Suc n)" by simp
  also have "... = 2 * {1..n} + 2 * Suc n" by simp
  also have "... = n * (n + 1) + 2 * Suc n" using HI by simp
  also have "... = Suc n * (Suc n + 1)" by simp
  finally show "?P (Suc n)" by simp
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 11. Demostrar la fórmula de la suma de impares:
     1 + 3 +  + (2*(n-1)+1) = n\<^sup>2
  ---------------------------------------------------------------------- *}
```

```
-- "La demostración automática es"
theorem suma_impares:
  "(i::nat=0..<n. 2 * i + 1) = n^Suc (Suc 0)"
by (induct n) simp_all

-- "La demostración estructurada es"
lemma suma_impares_2:
  "(i::nat=0..<n. 2 * i + 1) = n^Suc (Suc 0)"
   (is "?P n")
proof (induct n)
  show "?P 0" by simp
next
  fix n assume HI: "?P n"
  have "(i = 0..<Suc n. 2 * i + 1) =
        (i = 0..<n. 2 * i + 1) + 2 * n + 1" by simp
  also have "... = n^Suc (Suc 0) + 2 * n + 1" using HI by simp
  also have "... = Suc n ^ Suc (Suc 0)" by simp
  finally show "?P (Suc n)" by simp
qed

text {*
  Nota: En los siguientes ejercicios se usarán las siguientes
  propiedades distributivas:
  ů add_mult_distrib:  (m + n) * k = m * k + n * k
  ů add_mult_distrib2: k * (m + n) = k * m + k * n
  que agruparemos en el lema distrib
*}

lemmas distrib = add_mult_distrib add_mult_distrib2

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Demostrar la fórmula de la suma de cuadrados
     1 + 2\<^sup>2 +  + n\<^sup>2 = (n*(n+1)*(2*n+1))/6
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma suma_cuadrados:
  "6 * (i::nat=0..n. i^Suc (Suc 0)) = n * (n + 1) * (2 * n + 1)"
by (induct n) (auto simp add: distrib)
```

```
-- "La demostración estructurada es"
lemma suma_cuadrados_2:
  "6 * (i::nat=0..n. i^Suc (Suc 0)) = n * (n + 1) * (2 * n + 1)"
  (is "?P n")
proof (induct n)
  show "?P 0" by simp
next
  fix n :: "nat"
  assume HI: "6 * (i = 0..n. i ^ Suc (Suc 0)) = n * (n + 1) * (2 * n + 1)"
  have "6 * (i = 0..Suc n. i ^ Suc (Suc 0)) =
        6 * ((i = 0..n. i ^ Suc (Suc 0)) + (Suc n)^Suc (Suc 0))"
    by simp
  also have "... = 6 * (i = 0..n. i ^ Suc (Suc 0)) + 6 * (Suc n)^Suc (Suc 0)"
    by (simp add: distrib)
  also have "... = n * (n + 1) * (2 * n + 1) + 6 * (Suc n)^Suc (Suc 0)"
    using HI by simp
  also have "... = Suc n * (Suc n + 1) * (2 * Suc n + 1)"
    by (simp add: distrib)
  finally show "?P (Suc n)" by simp
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Demostrar la fórmula de la suma de cubos
     1 + 2\<^bsup>3\<^esup> +  + n\<^sup>3 = (n*(n+1))\<^sup>2/4
  Indicación: Se pueden usar, además de distrib, el lema
  ů power_eq_if: p ^ m = (if m = 0 then 1 else p * p ^ (m - 1))
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma suma_de_cubos:
  "4 * (i::nat=0..n. i^3) = (n * (n + 1))^Suc (Suc 0)"
by (induct n) (auto simp add: distrib power_eq_if)

-- "La demostración estructurada es"
lemma suma_de_cubos_2:
  "4 * (i::nat=0..n. i^3) = (n * (n + 1))^Suc (Suc 0)"
  (is "?P n" is "?S n = _")
proof (induct n)
```

```
  show "?P 0" by simp
next
  fix n
  assume HI: "?S n = (n * (n + 1))^Suc (Suc 0)"
  have "?S (n + 1) = ?S n + 4 * (n + 1)^3" by simp
  also have "... = (n * (n + 1))^Suc (Suc 0) + 4 * (n + 1)^3"
    using HI by simp
  also have "... = ((n + 1) * ((n + 1) + 1))^Suc (Suc 0)"
    by (simp add: power_eq_if distrib)
  finally show "?P (Suc n)" by simp
qed

end
```

6.3.2. Métodos de cálculo de cuadrados

```
chapter {* T6R3b: Métodos de cálculo de cuadrados *}

theory T6R3b
imports Main
begin

text {*
  En esta relación demostraremos la corrección de tres métodos para
  calcular el cuadrado de números naturales que aparece en un libro de
  matemáticas védicas.

  La relación está basada en el trabajo de Farhad Mehta titulado
  "Magical methods (Computing with natural numbers)". *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. El primer método sirve para calcular el cuadrado cuyo
  predecesor tenga un cuadrado fácilmente calculable, Por ejemplo, para
  el 61 se tiene
     61\<^bsup>2\<^esup> = 60\<^bsup>2 \<^esup>+ 60 + 61 = 3600 + 121 = 3721
  Basado en la idea anterior, definir la función
     cuadrado :: nat  nat
  ---------------------------------------------------------------------- *}

fun cuadrado :: "nat  nat" where
```

```
  "cuadrado 0       = 0"
| "cuadrado (Suc n) = (cuadrado n) + n + (Suc n)"

value "cuadrado 61"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar que cuadrado es correcta; es decir, que
     cuadrado n = n * n
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
theorem cuadrado_correcta [simp]:
  "cuadrado n = n * n"
by (induct n) auto

-- "La demostración estructurada es"
theorem cuadrado_correcta2:
  "cuadrado n = n * n"
proof (induct n)
  show "cuadrado 0 = 0 * 0" by simp
next
  fix n assume HI: "cuadrado n = n * n"
  have "cuadrado (Suc n) = cuadrado n + n + Suc n"by simp
  also have "... = n * n + n + Suc n" using HI by simp
  also have "... = Suc n * Suc n" by simp
  finally show "cuadrado (Suc n) = Suc n * Suc n" by simp
qed

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. El segundo método se aplica a los números mayores que 100
  que están próximos a 100. Por ejemplo,
     102\<^bsup>2\<^esup> = (102 + (102 - 100)) * 100 + (102 - 100)\<^sup>2
          = (102 + 2) * 100 + 4
          = 10404
  En general, si n  100,
     n\<^sup>2 = (n + (n - 100)) * 100 + (n - 100)\<^sup>2
  que se puede generalizar para todo m, si n  m
     n\<^sup>2 = (n + (n - m)) * m + (n - m)\<^sup>2
```

```
  Demostrar la última propiedad.
  ------------------------------------------------------------------ *}

-- "La demostración automática es"
lemma cuadrado2_correcta_aux [rule_format]:
  "mn. cuadrado n = (n + (n - m)) * m + cuadrado (n - m)"
by (induct n) (auto, case_tac m, auto)

-- "La demostración estructurada es"
lemma cuadrado2_correcta_aux2 [rule_format]:
  "mn. cuadrado n = (n+(n-m))*m + cuadrado (n-m)" (is "mn. ?P m n")
proof (induct n)
  show "m0. ?P m 0" by simp
next
  fix n
  assume HI: "mn. ?P m n"
  show "m  Suc n. ?P m (Suc n)"
  proof
    fix m
    show "m  Suc n  ?P m (Suc n)"
    proof
      assume "m  Suc n"
      show "?P m (Suc n)"
      proof (cases "m")
        assume "m = 0"
        thus "?P m (Suc n)" by simp
      next
        fix x
        assume "m = Suc x"
        thus "?P m (Suc n)" using HI `m  Suc n` by auto
      qed
    qed
  qed
qed

-- "La demostración estructurada más detallada es"
lemma cuadrado2_correcta_aux3 [rule_format]:
  "mn. cuadrado n = (n+(n-m))*m + cuadrado (n-m)" (is "mn. ?P m n")
proof (induct n)
  show "m0. ?P m 0"
```

```
  proof
    fix m
    show "m  0  ?P m 0"
    proof
      assume "m  0"
      hence "m = 0" by auto
      thus "?P m 0" by auto
    qed
  qed
next
  fix n
  assume HI: "mn. ?P m n"
  show "m  Suc n. ?P m (Suc n)"
  proof
    fix m
    show "m  Suc n  ?P m (Suc n)"
    proof
      assume "m  Suc n"
      show "?P m (Suc n)"
      proof (cases "m")
        assume "m = 0"
        thus "?P m (Suc n)" by simp
      next
        fix x
        assume "m = Suc x"
        show "?P m (Suc n)"
        proof -
          have "x  n" using `m  Suc n` `m = Suc x` by auto
          have "cuadrado (Suc n) = cuadrado n + n + Suc n" by simp
          also have "... = (n+(n-x))*x + cuadrado (n-x) + n + Suc n"
            using HI `x  n` by simp
          also have "... = (Suc n + (Suc n - m)) * m + cuadrado (Suc n - m)"
            using `m = Suc x` `m  Suc n` by auto
          finally show "?P m (Suc n)" by simp
        qed
      qed
    qed
  qed
qed
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Demostrar que, si n  100,
     n\<^sup>2 = (n + (n - 100)) * 100 + (n - 100)\<^sup>2
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma cuadrado2_correcta:
  "100  n  cuadrado n = (n + (n - 100)) * 100 + cuadrado (n - 100)"
by (rule cuadrado2_correcta_aux)

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. El tercer método se aplica a los números mayores que
  terminan en 5. Por ejemplo,
      85\<^bsup>2\<^esup> = ( 8  9)*100 + 25 = 7225
     995\<^bsup>2\<^esup> = (99*100)*100 + 25 = 990025
  Demostrar que
     ((10*n)+5)\<^sup>2 = (n*(n+1))*100 + 25
  ---------------------------------------------------------------------- *}

-- "La demostración automática es"
lemma cuadrado3_correcta:
  "cuadrado((10 * n) + 5) = ((n * (Suc n)) * 100) + 25"
by (auto simp add: add_mult_distrib add_mult_distrib2)

-- "La demostración estructurada es"
lemma cuadrado3_correcta2:
  "cuadrado((10 * n) + 5) = ((n * (Suc n)) * 100) + 25"
proof -
  have "cuadrado((10 * n) + 5) = ((10 * n) + 5) * ((10 * n) + 5)"
    by (rule cuadrado_correcta)
  also have "... = ((n * (Suc n)) * 100) + 25"
    by (auto simp add: add_mult_distrib add_mult_distrib2)
  finally show "cuadrado((10 * n) + 5) = ((n * (Suc n)) * 100) + 25"
    by simp
qed

end
```

Capítulo 7

Caso de estudio: Compilación de expresiones

```
chapter {* Tema 7: Caso de estudio: Compilación de expresiones *}

theory T7
imports Main
begin

text {*
  El objetivo de este tema es contruir un compilador de expresiones
  genéricas (construidas con variables, constantes y operaciones
  binarias) a una máquina de pila y demostrar su corrección.
*}

section {* Las expresiones y el intérprete *}

text {*
  Definición. Las expresiones son las constantes, las variables
  (representadas por números naturales) y las aplicaciones de operadores
  binarios a dos expresiones.
*}

type_synonym 'v binop = "'v  'v  'v"

datatype 'v expr =
  Const 'v
| Var nat
| App "'v binop" "'v expr" "'v expr"
```

```
text {*
  Definición. [Intérprete]
  La función "valor" toma como argumentos una expresión y un entorno
  (i.e. una aplicación de las variables en elementos del lenguaje) y
  devuelve el valor de la expresión en el entorno.
*}

fun valor :: "'v expr  (nat  'v)  'v" where
  "valor (Const b) ent = b"
| "valor (Var x) ent = ent x"
| "valor (App f e1 e2) ent = (f (valor e1 ent) (valor e2 ent))"

text {*
  Ejemplo. A continuación mostramos algunos ejemplos de evaluación con
  el intérprete.
*}

lemma
  "valor (Const 3) id = 3
   valor (Var 2) id = 2
   valor (Var 2) (x. x+1) = 3
   valor (App (op +) (Const 3) (Var 2)) (x. x+1) = 6
   valor (App (op +) (Const 3) (Var 2)) (x. x+4) = 9"
by simp

section {* La máquina de pila *}

text {*
  Nota. La máquina de pila tiene tres clases de intrucciones:
  ů cargar en la pila una constante,
  ů cargar en la pila el contenido de una dirección y
  ů aplicar un operador binario a los dos elementos superiores de la pila.
*}

datatype 'v instr =
  IConst 'v
| ILoad nat
| IApp "'v binop"
```

```
text {*
  Definición. [Ejecución]
  La ejecución de la máquina de pila se modeliza mediante la función
  "ejec" que toma una lista de intrucciones, una memoria (representada
  como una función de las direcciones a los valores, análogamente a los
  entornos) y una pila (representada como una lista) y devuelve la pila
  al final de la ejecución.
*}

fun ejec :: "'v instr list  (nat'v)  'v list  'v list" where
  "ejec [] ent vs = vs"
| "ejec (i#is) ent vs =
     (case i of
        IConst v  ejec is ent (v#vs)
      | ILoad x  ejec is ent ((ent x)#vs)
      | IApp f  ejec is ent ((f (hd vs) (hd (tl vs)))#(tl(tl vs))))"

text {*
  A continuación se muestran ejemplos de ejecución.
*}

lemma
  "ejec [IConst 3] id [7] = [3,7]
   ejec [ILoad 2, IConst 3] id [7] = [3,2,7]
   ejec [ILoad 2, IConst 3] (x. x+4) [7] = [3,6,7]
   ejec [ILoad 2, IConst 3, IApp (op +)] (x. x+4) [7] = [9,7]"
by simp

section {* El compilador *}

text {*
  Definición. El compilador "comp" traduce una expresión en una lista de
  instrucciones.
*}

fun comp :: "'v expr  'v instr list" where
  "comp (Const v) = [IConst v]"
| "comp (Var x) = [ILoad x]"
| "comp (App f e1 e2) = (comp e2) @ (comp e1) @ [IApp f]"
```

```
text {*
  A continuación se muestran ejemplos de compilación.
*}

lemma
  "comp (Const 3) = [IConst 3]
   comp (Var 2) = [ILoad 2]
   comp (App (op +) (Const 3) (Var 2)) = [ILoad 2, IConst 3, IApp (op +)]"
by simp

section {* Corrección del compilador *}

text {*
  Para demostrar que el compilador es correcto, probamos que el
  resultado de compilar una expresión y a continuación ejecutarla es lo
  mismo que interpretarla; es decir,
*}

theorem "ejec (comp e) ent [] = [valor e ent]"
oops

text {*
  El teorema anterior no puede demostrarse por inducción en e. Para
  demostrarlo, lo generalizamos a
*}

theorem "vs. ejec (comp e) ent vs = (valor e ent)#vs"
oops

text {*
  En la demostración del teorema anterior usaremos el siguiente lema.
*}

lemma ejec_append:
  " vs. ejec (xs@ys) ent vs = ejec ys ent (ejec xs ent vs)" (is "?P xs")
proof (induct xs)
  show "?P []" by simp
next
  fix a xs
  assume "?P xs"
```

```
  thus "?P (a#xs)" by (cases "a", auto)
qed

-- "La demostración detallada es"
lemma ejec_append_1:
  " vs. ejec (xs@ys) ent vs = ejec ys ent (ejec xs ent vs)" (is "?P xs")
proof (induct xs)
  show "?P []" by simp
next
  fix a xs
  assume HI: "?P xs"
  thus "?P (a#xs)"
  proof (cases "a")
    case IConst thus ?thesis using HI by simp
  next
    case ILoad thus ?thesis using HI by simp
  next
    case IApp thus ?thesis using HI by simp
  qed
qed

text {*
  Una demostración más detallada del lema es la siguiente:
*}

lemma ejec_append_2:
  "vs. ejec (xs@ys) ent vs = ejec ys ent (ejec xs ent vs)" (is "?P xs")
proof (induct xs)
  show "?P []" by simp
next
  fix a xs
  assume HI: "?P xs"
  thus "?P (a#xs)"
  proof (cases "a")
    fix v assume C1: "a=IConst v"
    show " vs. ejec ((a#xs)@ys) ent vs = ejec ys ent (ejec (a#xs) ent vs)"
    proof
      fix vs
      have "ejec ((a#xs)@ys) ent vs = ejec (((IConst v)#xs)@ys) ent vs"
        using C1 by simp
```

```
    also have " = ejec (xs@ys) ent (v#vs)" by simp
    also have " = ejec ys ent (ejec xs ent (v#vs))" using HI by simp
    also have " = ejec ys ent (ejec ((IConst v)#xs) ent vs)" by simp
    also have " = ejec ys ent (ejec (a#xs) ent vs)" using C1 by simp
    finally show "ejec ((a#xs)@ys) ent vs =
                  ejec ys ent (ejec (a#xs) ent vs)" .
  qed
next
  fix n assume C2: "a=ILoad n"
  show " vs. ejec ((a#xs)@ys) ent vs = ejec ys ent (ejec (a#xs) ent vs)"
  proof
    fix vs
    have "ejec ((a#xs)@ys) ent vs = ejec (((ILoad n)#xs)@ys) ent vs"
      using C2 by simp
    also have " = ejec (xs@ys) ent ((ent n)#vs)" by simp
    also have " = ejec ys ent (ejec xs ent ((ent n)#vs))" using HI by simp
    also have " = ejec ys ent (ejec ((ILoad n)#xs) ent vs)" by simp
    also have " = ejec ys ent (ejec (a#xs) ent vs)" using C2 by simp
    finally show "ejec ((a#xs)@ys) ent vs =
                  ejec ys ent (ejec (a#xs) ent vs)" .
  qed
next
  fix f assume C3: "a=IApp f"
  show "vs. ejec ((a#xs)@ys) ent vs = ejec ys ent (ejec (a#xs) ent vs)"
  proof
    fix vs
    have "ejec ((a#xs)@ys) ent vs = ejec (((IApp f)#xs)@ys) ent vs"
      using C3 by simp
    also have " = ejec (xs@ys) ent ((f (hd vs) (hd (tl vs)))#(tl(tl vs)))"
      by simp
    also have " = ejec ys
                       ent
                       (ejec xs ent ((f (hd vs) (hd (tl vs)))#(tl(tl vs))))"
      using HI by simp
    also have " = ejec ys ent (ejec ((IApp f)#xs) ent vs)" by simp
    also have " = ejec ys ent (ejec (a#xs) ent vs)" using C3 by simp
    finally show "ejec ((a#xs)@ys) ent vs =
                  ejec ys ent (ejec (a#xs) ent vs)" .
  qed
qed
```

```
qed

text {*
  La demostración automática del teorema es
*}

theorem "vs. ejec (comp e) ent vs = (valor e ent)#vs"
by (induct e) (auto simp add:ejec_append)

text {*
  La demostración estructurada del teorema es
*}

theorem "vs. ejec (comp e) ent vs = (valor e ent)#vs"
proof (induct e)
  fix v
  show "vs. ejec (comp (Const v)) ent vs = (valor (Const v) ent)#vs" by simp
next
  fix x
  show "vs. ejec (comp (Var x)) ent vs = (valor (Var x) ent) # vs" by simp
next
  fix f e1 e2
  assume HI1: "vs. ejec (comp e1) ent vs = (valor e1 ent) # vs"
    and HI2: "vs. ejec (comp e2) ent vs = (valor e2 ent) # vs"
  show "vs. ejec (comp (App f e1 e2)) ent vs = (valor (App f e1 e2) ent) # vs"
  proof
    fix vs
    have "ejec (comp (App f e1 e2)) ent vs
          = ejec ((comp e2) @ (comp e1) @ [IApp f]) ent vs" by simp
    also have " = ejec ((comp e1) @ [IApp f]) ent (ejec (comp e2) ent vs)"
      using ejec_append by blast
    also have " = ejec [IApp f]
                       ent
                       (ejec (comp e1) ent (ejec (comp e2) ent vs))"
      using ejec_append by blast
    also have " =  ejec [IApp f] ent (ejec (comp e1) ent ((valor e2 ent)#vs))"
      using HI2 by simp
    also have " = ejec [IApp f] ent ((valor e1 ent)#((valor e2 ent)#vs))"
      using HI1 by simp
    also have " = (f (valor e1 ent) (valor e2 ent))#vs" by simp
```

```
    also have " = (valor (App f e1 e2) ent) # vs" by simp
    finally
    show "ejec (comp (App f e1 e2)) ent vs = (valor (App f e1 e2) ent) # vs"
      by blast
  qed
qed

end
```

Capítulo 8

Conjuntos, funciones y relaciones

```
chapter {* Tema 8: Conjuntos, funciones y relaciones *}

theory T8
imports Main
begin

section {* Conjuntos *}

subsection {* Operaciones con conjuntos *}

text {*
  Nota. La teoría elemental de conjuntos es HOL/Set.thy.

  Nota. En un conjunto todos los elemento son del mismo tipo (por
  ejemplo, del tipo ) y el conjunto tiene tipo (en el ejemplo, " set").

  Reglas de la intersección:
  ů IntI:  c  A; c  B  c  A  B
  ů IntD1: c  A  B  c  A
  ů IntD2: c  A  B  c  B

  Nota. Propiedades del complementario:
  ů Compl_iff: (c  - A) = (c  A)
  ů Compl_Un:  - (A  B) = - A  - B

  Nota. El conjunto vacío se representa por {} y el universal por UNIV.

  Nota. Propiedades de la diferencia y del complementario:
```

```
  ů Diff_disjoint:   A  (B - A) = {}
  ů Compl_partition: A  - A = UNIV

  Nota. Reglas de la relación de subconjunto:
  ů subsetI: (x. x  A  x  B)  A  B
  ů subsetD: A  B; c  A  c  B

  Nota. Ejemplo trivial.
*}

lemma "(A  B  C) = (A  C  B  C)"
by blast

text {*
  Nota. Otro ejemplo trivial.
*}

lemma "(A  -B) = (B  -A)"
by blast

text {*
  Principio de extensionalidad de conjuntos:
  ů set_ext: (x. (x  A) = (x  B))  A = B

  Reglas de la igualdad de conjuntos:
  ů equalityI: A  B; B  A  A = B
  ů equalityE: A = B; A  B; B  A  P  P

  Lema. [Analogía entre intersección y conjunción]
  "x  A  B" syss "x  A" y "x  B".
*}

lemma "(x  A  B) = (x  A  x  B)"
by simp

text {*
  Lema. [Analogía entre unión y disyunción]
  "x  A  B" syss "x  A" ó "x  B".
*}
```

```
lemma "(x  A  B) = (x  A  x  B)"
by simp

text {*
  Lema. [Analogía entre subconjunto e implicación]
  "(A  B)" syss para todo "x", si "x  A" entonces "x  B".
*}

lemma "(A  B) = ( x. x  A  x  B)"
by auto

text {*
  Lema. [Analogía entre complementario y negación]
  x pertenece al complementario de A syss x no pertenece a A.
*}

lemma "(x  -A) = (x  A)"
by simp

subsection {* Notación de conjuntos finitos *}

text {*
  Nota. La teoría de conjuntos finitos es HOL/Finite_Set.thy.

  Nota. Los conjuntos finitos se definen por inducción a partir de las
  siguientes reglas inductivas:
  ů El conjunto vacío es un conjunto finito.
    ů emptyI: "finite {}"
  ů Si se le añade un elemento a un conjunto finito se obtiene otro
    conjunto finito.
    ů insertI: "finite A  finite (insert a A)"

  A continuación se muestran ejemplos de conjuntos finitos.
*}

lemma
  "insert 2 {} = {2}
   insert 3 {2} = {2,3}
   insert 2 {2,3} = {2,3}
   {2,3} = {3,2,3,2,2}"
```

```
by auto

text {*
  Nota. Los conjuntos finitos se representan con la notación conjuntista
  habitual: los elementos entre llaves y separados por comas.

  Nota. Lema trivial.
*}

lemma "{a,b}  {c,d} = {a,b,c,d}"
by blast

text {*
  Nota. Conjetura falsa.
*}

lemma "{a,b}  {b,c} = {b}"
refute
oops

text {*
  Nota. Conjetura corregida.
*}

lemma "{a,b}  {b,c} = (if a=c then {a,b} else {b})"
by auto

text {*
  Sumas y productos de conjuntos finitos:
  · (setsum f A) es la suma de la aplicación de f a los elementos del
    conjunto finito A,
  · (setprod f A) es producto de la aplicación de f a los elementos del
    conjunto finito A,
  · A es la suma de los elementos del conjunto finito A,
  · A es el producto de los elementos del conjunto finito A.

  Ejemplos de definiciones recursivas sobre conjuntos finitos:
  Sea A un conjunto finito de números naturales.
  · sumaConj A es la suma de los elementos A.
  · productoConj A es el producto de los elementos de A.
```

```
  ů sumaCuadradosconj A es la suma de los cuadrados de los elementos A.
*}

definition sumaConj :: "nat set  nat" where
  "sumaConj S  S"

definition productoConj :: "nat set  nat" where
  "productoConj S  S"

definition sumaCuadradosConj :: "nat set  nat" where
  "sumaCuadradosConj S  setsum (x. x*x) S"

text {*
  Nota. Para simplificar lo que sigue, declaramos las anteriores
  definiciones como reglas de simplificación.
*}

declare sumaConj_def[simp]
declare productoConj_def[simp]
declare sumaCuadradosConj_def[simp]

text {*
  Ejemplos de evaluación de las anteriores definiciones recursivas.
*}

lemma
  "sumaConj {1,2,3,4} = 10
   productoConj {1,2,3} = productoConj {3,2}
   sumaCuadradosConj {1,2,3,4} = 30"
by simp

text {*
  Inducción sobre conjuntos finitos: Para demostrar que todos los
  conjuntos finitos tienen una propiedad P basta probar que
  ů El conjunto vacío tiene la propiedad P.
  ů Si a un conjunto finito que tiene la propiedad P se le añade un
    nuevo elemento, el conjunto obtenido sigue teniendo la propiedad P.
  En forma de regla
  ů finite_induct: finite F;
                   P {};
```

```
                    x F. finite F; x  F; P F  P ({x}  F)
                    P F
```

Lema. [Ejemplo de inducción sobre conjuntos finitos]
Sea S un conjunto finito de números naturales. Entonces todos los
elementos de S son menores o iguales que la suma de los elementos de S.

Demostración automática:
}

```
lemma "finite S  xS. x  sumaConj S"
by (induct rule: finite_induct) auto

text {*
  Demostración estructurada:
*}

lemma sumaConj_acota: "finite S  xS. x  sumaConj S"
proof (induct rule: finite_induct)
  show "x  {}. x  sumaConj {}" by simp
next
  fix x and F
  assume fF: "finite F"
     and xF: "x  F"
     and HI: " xF. x  sumaConj F"
  show "y  insert x F. y  sumaConj (insert x F)"
  proof
    fix y
    assume "y  insert x F"
    show "y  sumaConj (insert x F)"
    proof (cases "y = x")
      assume "y = x"
      hence "y  x + (sumaConj F)" by simp
      also have " = sumaConj (insert x F)" using fF xF by simp
      finally show ?thesis .
    next
      assume "y  x"
      hence "y  F" using `y  insert x F` by simp
      hence "y  sumaConj F" using HI by blast
      also have "  x + (sumaConj F)" by simp
```

```
      also have " = sumaConj (insert x F)" using fF xF by simp
      finally show ?thesis .
    qed
  qed
qed

subsection {* Definiciones por comprensión *}

text {*
  El conjunto de los elementos que cumple la propiedad P se representa
  por {x. P}.

  Reglas de comprensión (relación entre colección y pertenencia):
  ů mem_Collect_eq: (a ∈ {x. P x}) = P a
  ů Collect_mem_eq: {x. x ∈ A} = A

  Dos lemas triviales.
*}

lemma "{x. P x ∧ x ∈ A} = {x. P x} ∩ A"
by blast

lemma "{x. P x ∨ Q x} = -{x. P x} ∩ {x. Q x}"
by blast

text {*
  Nota. Ejemplo con la sintaxis general de comprensión.
*}

lemma
  "{p*q | p q. p ∈ prime ∧ q ∈ prime} =
   {z. ∃p q. z = p*q ∧ p ∈ prime ∧ q ∈ prime}"
by blast

text {*
   En HOL, la notación conjuntista es azúcar sintáctica:
   ů x ∈ A es equivalente a A(x).
   ů {x. P} es equivalente a x. P.
*}
```

```
text {*
  Definición. [Ejemplo de definición por comprensión]
  El conjunto de los pares es el de los números n para los que existe un
  m tal que n = 2*m.
*}

definition Pares :: "nat set" where
  "Pares  {n. m. n = 2*m }"

text {*
  Ejemplo. Los números 2 y 34 son pares.
*}

lemma
  "2  Pares
   34  Pares"
by (simp add: Pares_def)

text {*
  Definición. El conjunto de los impares es el de los números n para los
  que existe un m tal que n = 2*m + 1.
*}

definition Impares :: "nat set" where
  "Impares  {n. m. n = 2*m + 1 }"

text {*
  Lema. [Ejemplo con las reglas de intersección y comprensión]
  El conjunto de los pares es disjunto con el de los impares.
*}

lemma "x  (Pares  Impares)"
proof
  fix x assume S: "x  (Pares  Impares)"
  hence "x  Pares" by (rule IntD1)
  hence "m. x = 2 * m" by (simp only: Pares_def mem_Collect_eq)
  then obtain p where p: "x = 2 * p" ..
  from S have "x  Impares" by (rule IntD2)
  hence " m. x = 2 * m + 1" by (simp only: Impares_def mem_Collect_eq)
  then obtain q where q: "x = 2 * q + 1" ..
```

```
  from p and q show "False" by arith
qed

subsection {* Cuantificadores acotados *}

text {*
  Reglas de cuantificador universal acotado ("bounded"):
  ů ballI: (x. x  A  P x)  xA. P x
  ů bspec: xA. P x; x  A  P x

  Reglas de cuantificador existencial acotado ("bounded"):
  ů bexI: P x; x  A  xA. P x
  ů bexE: xA. P x; x. x  A; P x  Q  Q

  Reglas de la unión indexada:
  ů UN_iff: (b  (xA. B x)) = (xA. b  B x)
  ů UN_I:   a  A; b  B a  b  (xA. B x)
  ů UN_E:   b  (xA. B x); x. x  A; b  B x  R  R

  Reglas de la unión de una familia:
  ů Union_def: S = (xS. x)
  ů Union_iff: (A  C) = (XC. A  X)

  Reglas de la intersección indexada:
  ů INT_iff: (b  (xA. B x)) = (xA. b  B x)
  ů INT_I:   (x. x  A  b  B x)  b  (xA. B x)
  ů INT_E:   b  (xA. B x); b  B a  R; a  A  R  R

  Reglas de la intersección de una familia:
  ů Inter_def: S = (xS. x)
  ů Inter_iff: (A  C) = (XC. A  X)

  Abreviaturas:
  ů "Collect P" es lo mismo que "{x. P}".
  ů "All P"     es lo mismo que "x. P x".
  ů "Ex P"      es lo mismo que "x. P x".
  ů "Ball A P"  es lo mismo que "xA. P x".
  ů "Bex A P"   es lo mismo que "xA. P x".
  *}
```

```
subsection {* Conjuntos finitos y cardinalidad *}

text {*
  El número de elementos de un conjunto finito A es el cardinal de A y
  se representa por "card A".

  Ejemplos de cardinales de conjuntos finitos.
*}

lemma
  "card {} = 0
   card {4} = 1
   card {4,1} = 2
   x  y  card {x,y} = 2"
by simp

text {*
  Propiedades de cardinales:
  ů Cardinal de la unión de conjuntos finitos:
    card_Un_Int: finite A; finite B
                  card A + card B = card (A  B) + card (A  B)"
  ů Cardinal del conjunto potencia:
    card_Pow: finite A  card (Pow A) = 2 ^ card A
*}

section {* Funciones *}

text {*
  La teoría de funciones es HOL/Fun.thy.
*}

subsection {* Nociones básicas de funciones *}

text {*
  Principio de extensionalidad para funciones:
  ů ext: (x. f x = g x)  f = g

  Actualización de funciones
  ů fun_upd_apply: (f(x := y)) z = (if z = x then y else f z)
  ů fun_upd_upd:   f(x := y, x := z) = f(x := z)
```

```
  Función identidad
  ů id_def: id  x. x

  Composición de funciones:
  ů o_def: f  g = (x. f (g x))

  Asociatividad de la composición:
  ů o_assoc: f  (g  h) = (f  g)  h
*}

subsection {* Funciones inyectivas, suprayectivas y biyectivas *}

text {*
  Función inyectiva sobre A:
  ů inj_on_def: inj_on f A  xA. yA. f x = f y  x = y

  Nota. "inj f" es una abreviatura de "inj_on f UNIV".

  Función suprayectiva:
  ů surj_def: surj f  y. x. y = f x

  Función biyectiva:
  ů bij_def: bij f  inj f  surj f

  Propiedades de las funciones inversas:
  ů inv_f_f:      inj f  inv f (f x) = x
  ů surj_f_inv_f: surj f  f (inv f y) = y
  ů inv_inv_eq:   bij f  inv (inv f) = f

  Igualdad de funciones (por extensionalidad):
  ů fun_eq_iff: (f = g) = (x. f x = g x)

  Lema. Una función inyectiva puede cancelarse en el lado izquierdo de
  la composición de funciones.
*}

lemma
  assumes "inj f"
  shows "(f  g = f  h) = (g = h)"
```

```
proof
  assume "f  g = f  h"
  thus "g = h" using ‘inj f‘ by (simp add:fun_eq_iff inj_on_def)
next
  assume "g = h"
  thus "f  g = f  h" by auto
qed

text {*
  Una demostración más detallada es la siguiente
*}

lemma
  assumes "inj f"
  shows "(f  g = f  h) = (g = h)"
proof
  assume "f  g = f  h"
  show "g = h"
  proof
    fix x
    have "(f  g)(x) = (f  h)(x)" using ‘f  g = f  h‘ by simp
    hence "f(g(x)) = f(h(x))" by simp
    thus "g(x) = h(x)" using ‘inj f‘ by (simp add:inj_on_def)
  qed
next
  assume "g = h"
  show "f  g = f  h"
  proof
    fix x
    have "(f  g) x = f(g(x))" by simp
    also have " = f(h(x))" using ‘g = h‘ by simp
    also have " = (f  h) x" by simp
    finally show "(f  g) x = (f  h) x" by simp
  qed
qed

subsubsection {* Función imagen *}

text {*
  Imagen de un conjunto mediante una función:
```

```
  ů image_def: f ` A = {y. (xA. y = f x)

  Propiedades de la imagen:
  ů image_compose: (f  g)`r = f`g`r
  ů image_Un:      f`(A  B) = f`A  f`B
  ů image_Int:     inj f  f`(A  B) = f`A  f`B"

  Ejemplos de demostraciones triviales de propiedades de la imagen.
*}

lemma "f`A  g`A = (xA. {f x, g x})"
by auto

lemma "f`{(x,y). P x y} = {f(x,y) | x y. P x y}"
by auto

text {*
  El rango de una función ("range f") es la imagen del universo ("f`UNIV").

  Imagen inversa de un conjunto:
  ů vimage_def: f -` B  {x. f x : B}

  Propiedad de la imagen inversa de un conjunto:
  ů vimage_Compl: f -` (-A) = -(f -` A)
*}

section {* Relaciones *}

subsection {* Relaciones básicas *}

text {*
  La teoría de relaciones es HOL/Relation.thy.

  Las relaciones son conjuntos de pares.

  Relación identidad:
  ů Id_def: Id  {p. x. p = (x,x)}

  Composición de relaciones:
  ů rel_comp_def: r O s  {(x,z). y. (x, y)  r  (y, z)  s}
```

```
  Propiedades:
  ů R_O_Id:         R O Id = R
  ů rel_comp_mono: r'  r; s'  s  (r' O s')  (r O s)

  Imagen inversa de una relación:
  ů converse_iff: ((a,b)  r\<^bsup>\<^sup>-1\<^esup>) = ((b,a)  r)

  Propiedad de la imagen inversa de una relación:
  ů converse_rel_comp: (r O s)\<^bsup>-1\<^esup> = s\<^bsup>-1\<^esup> O r\<^bsup>-1\

  Imagen de un conjunto mediante una relación:
  ů Image_iff: (b  r``A) = (x:A. (x, b)  r)

  Dominio de una relación:
  ů Domain_iff: (a  Domain r) = (y. (a, y)  r)

  Rango de una relación:
  ů Range_iff: (a  Range r) = (y. (y,a)  r)
*}

subsection {* Clausura reflexiva y transitiva *}

text {*
  La teoría de la clausura reflexiva y transitiva de una relación es
  HOL/Transitive_Closure.thy.

  Potencias de relaciones:
  ů R ^^ 0 = Id
  ů R ^^ (Suc n) = (R ^^ n) O R

  La clausura reflexiva y transitiva de la relación r es la menor
  solución de la ecuación:
  ů rtrancl_unfold: r\<^sup>* = Id  (r\<^sup>* O r)

  Propiedades básicas de la clausura reflexiva y transitiva:
  ů rtrancl_refl:   (a,a)  r\<^sup>*
  ů r_into_rtrancl: p  r  p  r\<^sup>*
  ů rtrancl_trans:  (a,b)  r\<^sup>*; (b,c)  r\<^sup>*  (a,c)  r\<^sup>*
```

```
  Inducción sobre la clausura reflexiva y transitiva
  ů rtrancl_induct: (a,b)  r\<^sup>*;
                    P b;
                    y z. (y,z)  r; (z,b)  r\<^sup>*; P z  P y
                    P a"}

  Idempotencia de la clausura reflexiva y transitiva:
  ů rtrancl_idemp: (r\<^sup>* )\<^sup>* = r\<^sup>*

  Reglas de introducción de la clausura transitiva:
  ů r_into_trancl': p  r  p  r\<^sup>+
  ů trancl_trans:   (a,b)  r\<^sup>+; (b,c)  r\<^sup>+  (a,c)  r\<^sup>+

  Ejemplo de propiedad:
  ů trancl_converse: (rŕź)\<^sup>+ = (r\<^sup>+)ŕź
*}

subsection {* Una demostración elemental *}

text {*
  El teorema que se desea demostrar es que la clausura reflexiva y
  transitiva conmuta con la inversa (rtrancl_converse). Para
  demostrarlo introducimos dos lemas auxiliares: rtrancl_converseD y
  rtrancl_converseI.
*}

lemma rtrancl_converseD: "(x,y)  (rŕź)\<^sup>*  (y,x)  r\<^sup>*"
proof (induct rule:rtrancl_induct)
  show "(x,x)  r\<^sup>*" by (rule rtrancl_refl)
next
  fix y z
  assume "(x,y)  (rŕź)\<^sup>*" and "(y,z)  rŕź" and "(y,x)  r\<^sup>*"
  show "(z,x)  r\<^sup>*"
  proof (rule rtrancl_trans)
    show "(z,y)  r\<^sup>*" using `(y,z)  rŕź` by simp
  next
    show "(y,x)  r\<^sup>*" using `(y,x)  r\<^sup>*` by simp
  qed
qed
```

```
lemma rtrancl_converseI: "(y,x)  r\<^sup>*  (x,y)  (r⁻¹)\<^sup>*"
proof (induct rule:rtrancl_induct)
  show "(y,y)  (r⁻¹)\<^sup>*" by (rule rtrancl_refl)
next
  fix u z
  assume "(y,u)  r\<^sup>*" and "(u,z)  r" and "(u,y)  (r⁻¹)\<^sup>*"
  show "(z,y)  (r⁻¹)\<^sup>*"
  proof (rule rtrancl_trans)
    show "(z,u)  (r⁻¹)\<^sup>*" using `(u,z)  r` by auto
  next
    show "(u,y)  (r⁻¹)\<^sup>*" using `(u,y)  (r⁻¹)\<^sup>*` by simp
  qed
qed

theorem rtrancl_converse: "(r⁻¹)\<^sup>* = (r\<^sup>*)⁻¹"
proof
  show "(r⁻¹)\<^sup>*  (r\<^sup>*)⁻¹" by (auto simp add:rtrancl_converseD)
next
  show "(r\<^sup>*)⁻¹  (r⁻¹)\<^sup>*" by (auto simp add:rtrancl_converseI)
qed

text {*
  Puede demostrarse de manera más corta como sigue:
*}

theorem "(r⁻¹)\<^sup>* = (r\<^sup>*)⁻¹"
by (auto intro: rtrancl_converseI dest: rtrancl_converseD)

section {* Relaciones bien fundamentadas e inducción *}

text {*
  La teoría de las relaciones bien fundamentadas es
  HOL/Wellfounded_Relations.thy.

  La relación-objeto "less_than" es el orden de los naturales que es
  bien fundamentada:
  ů less_than_iff: ((x,y)  less_than) = (x < y)
  ů wf_less_than:  wf less_than

  Notas sobre medidas:
```

```
  ů Imagen inversa de una relación mediante una función:
    ů inv_image_def: inv_image r f  {(x,y). (f x,f y)  r
  ů Conservación de la buena fundamentación:
    ů wf_inv_image: wf r  wf (inv_image r f)
  ů Definición de la medida:
    ů measure_def: measure  inv_image less_than
  ů Buena fundamentación de la medida:
    ů wf_measure: wf (measure f)
*}

text {*
  Notas sobre el producto lexicográfico:
  ů Definición del producto lexicográfico (lex_prod_def):
    ra <*lex*> rb  {((a,b),(a',b')). (a,a')  ra
                                     (a = a'  (b,b')  rb)}
  ů Conservación de la buena fundamentación:
    ů wf_lex_prod: wf ra; wf rb  wf (ra <*lex*> rb)

  El orden de multiconjuntos está en la teoría HOL/Library/Multiset.thy.

  Inducción sobre relaciones bien fundamentadas:
  ů wf_induct: wf r; x. (y. (y,x)  r  P y)  P x  P a
*}

end
```

8.1. Gramáticas libres de contexto

```
chapter {* T8R1: Gramáticas libres de contexto *}

theory T8R1
imports Main
begin

text {*
  En esta relación se definen dos gramáticas libres de contexto y se
  demuestra que son equivalentes. Además, se define por recursión una
  función para reconocer las palabras de la gramática y se demuestra que
  es correcta y completa. *}
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Una gramática libre de contexto para las expresiones
  parentizadas es
     S   | '(' S ')' | SS
  definir inductivamente la gramática S usando A y B para '(' y ')',
  respectivamente.
  ---------------------------------------------------------------------- *}

datatype alfabeto = A | B

inductive_set S :: "alfabeto list set" where
  S1: "[]  S"
| S2: "w  S  [A] @ w @ [B]  S"
| S3: "v  S  w  S  v @ w  S"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Otra gramática libre de contexto para las expresiones
  parentizadas es
     T   | T '(' T ')'
  definir inductivamente la gramática T usando A y B para '(' y ')',
  respectivamente.
  ---------------------------------------------------------------------- *}

inductive_set T :: "alfabeto list set" where
  T1: "[]  T"
| T2: "v  T  w  T  v @ [A] @ w @ [B]  T"

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Demostrar que T está contenido en S.
  ---------------------------------------------------------------------- *}

lemma T_en_S:
  assumes "w  T"
  shows   "w  S"
using assms
proof (induct rule: T.induct)
  show "[]  S" by (rule S1)
```

```
next
  fix v w
  assume "v  T" and "v  S" and "w  T" and "w  S"
  have "[A] @ w @ [B]  S" using ‘w  S‘ by (rule S2)
  with ‘v  S‘ show "v @ [A] @ w @ [B]  S" by (rule S3)
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 4. Demostrar que S está contenido en T.
  ---------------------------------------------------------------------- *}

text {*
  Se usarán dos lemas auxiliares:
  ů S_en_T_aux1: w  T  [A] @ w @ [B]  T
  ů S_en_T_aux2: v  T  u  T  u @ v  T
*}

-- "La demostración estructurada del primer lema es"
lemma S_en_T_aux1:
  assumes "w  T"
  shows   "[A] @ w @ [B]  T"
proof -
  have "[]  T" by (rule T1)
  hence "[] @ [A] @ w @ [B]  T" using assms by (rule T2)
  thus "[A] @ w @ [B]  T" by simp
qed

-- "La demostración automática del primer lema es"
lemma S_en_T_aux1b:
  "w  T  [A] @ w @ [B]  T"
using T1 T2[where v = "[]"] by simp

-- "La demostración estructurada del segundo lema es"
lemma S_en_T_aux2:
  "v  T  u  T  u @ v  T"
proof (induct rule: T.induct)
  assume "u  T"
  thus "u @ []  T" by auto
next
```

```
  fix w1 w2
  assume "w1 ∈ T"
         "u ∈ T ⟹ u @ w1 ∈ T"
         "w2 ∈ T"
         "u ∈ T ⟹ u @ w2 ∈ T"
         "u ∈ T"
  hence "u @ w1 ∈ T" by simp
  hence "(u @ w1) @ [A] @ w2 @ [B] ∈ T" using ‹w2 ∈ T› by (rule T2)
  thus "u @ w1 @ [A] @ w2 @ [B] ∈ T" by auto
qed

lemma S_en_T:
  "w ∈ S ⟹ w ∈ T"
proof (induct rule: S.induct)
  show "[] ∈ T" by (rule T1)
next
  fix w
  assume "w ∈ S" "w ∈ T"
  show "[A] @ w @ [B] ∈ T" using ‹w ∈ T› by (rule S_en_T_aux1)
next
  fix v w
  assume "v ∈ S" "v ∈ T" "w ∈ S" "w ∈ T"
  show "v @ w ∈ T" using ‹w ∈ T› ‹v ∈ T› by (rule S_en_T_aux2)
qed

text {*
  ---------------------------------------------------------------------
  Ejercicio 4. Demostrar que S y T son iguales.
  ---------------------------------------------------------------------- *}

lemma S_igual_T:
  "S = T"
by (auto simp add: S_en_T T_en_S)

text {*
  ---------------------------------------------------------------------
  Ejercicio 5. En lugar de una gramática, se puede usar el siguiente
  procedimiento para determinar si la cadena es una sucesión de
  paréntesis bien balanceada: se recorre la cadena de izquierda a
  derecha contando cuántos paréntesis de necesitan para que esté bien
```

```
  balanceada. Si el contador al final de la cadena es 0, la cadena está
  bien balanceada.

  Definir la función
     balanceada :: alfabeto list  bool
  tal que (balanceada w) se verifica si w está bien balanceada. Por
  ejemplo,
     balanceada [A,A,B,B] = True
     balanceada [A,B,A,B] = True
     balanceada [A,B,B,A] = False
  Indicación: Definir balanceada  usando la función auxiliar
     balanceada_aux :: alfabeto list  nat  bool
  tal que (balanceada_aux w 0) se verifica si w está bien balanceada.
  ------------------------------------------------------------------ *}

fun balanceada_aux :: "alfabeto list  nat  bool" where
  "balanceada_aux []    0       = True"
| "balanceada_aux (A#w) n       = balanceada_aux w (Suc n)"
| "balanceada_aux (B#w) (Suc n) = balanceada_aux w n"
| "balanceada_aux w     n       = False"

fun balanceada :: "alfabeto list  bool" where
  "balanceada w = balanceada_aux w 0"

value "balanceada [A,A,B,B]" -- "= True"
value "balanceada [A,B,A,B]" -- "= True"
value "balanceada [A,B,B,A]" -- "= False"

text {*
  ------------------------------------------------------------------
  Ejercicio 6. Demostrar que balanceada es un reconocedor correcto de la
  gramática S; es decir,
     w  S  balanceada w
  ------------------------------------------------------------------ *}

text {*
  Nota: En la demostración se usarán los siguientes lemas auxiliares:
  ů balanceada_correcto_aux_1:
       balanceada_aux w n  balanceada_aux (w @ [B]) (Suc n)
  ů balanceada_correcto_aux_2:
```

```
       balanceada_aux ?v ?n;
        balanceada_aux ?w 0
        balanceada_aux (?v @ ?w) ?n
  ⟹ balanceada_correcto_aux_3:
       w ∈ S ⟷ balanceada_aux w 0 ⟷ *}

-- "La demostración automática del primer lema auxiliar es"
lemma balanceada_correcto_aux_1:
  "balanceada_aux w n ⟷ balanceada_aux (w @ [B]) (Suc n)"
by (induct w n rule: balanceada_aux.induct) simp_all

-- "La demostración estructurada del primer lema auxiliar es"
lemma balanceada_correcto_aux_1b:
  assumes "balanceada_aux w n"
  shows   "balanceada_aux (w @ [B]) (Suc n)"
using assms
proof (induct w n rule: balanceada_aux.induct)
  assume "balanceada_aux [] 0"
  thus "balanceada_aux ([] @ [B]) (Suc 0)" by simp
next
  fix w n
  assume "balanceada_aux w (Suc n) ⟹ balanceada_aux (w @ [B]) (Suc (Suc n))"
         "balanceada_aux (A # w) n"
  thus "balanceada_aux ((A # w) @ [B]) (Suc n)" by simp
next
  fix w n
  assume "balanceada_aux w n ⟹ balanceada_aux (w @ [B]) (Suc n)"
         "balanceada_aux (B # w) (Suc n)"
  thus   "balanceada_aux ((B # w) @ [B]) (Suc (Suc n))" by simp
next
  fix v
  assume "balanceada_aux (B # v) 0"
  thus "balanceada_aux ((B # v) @ [B]) (Suc 0)" by simp
next
  fix v
  assume "balanceada_aux [] (Suc v)"
  thus "balanceada_aux ([] @ [B]) (Suc (Suc v))" by simp
qed

-- "La demostración automática del segundo lema auxiliar es"
```

```
lemma balanceada_correcto_aux_2:
  assumes "balanceada_aux v n"
          "balanceada_aux w 0"
  shows   "balanceada_aux (v @ w) n"
using assms
by (induct v n rule: balanceada_aux.induct) simp_all

-- "La demostración automática del tercer lema auxiliar es"
lemma balanceada_correcto_aux_3:
  "w ∈ S ⟹ balanceada_aux w 0"
by (induct rule: S.induct)
   (auto simp add: balanceada_correcto_aux_1 balanceada_correcto_aux_2)

-- "La demostración estructurada del tercer lema auxiliar es"
lemma balanceada_correcto_aux_3b:
  "w ∈ S ⟹ balanceada_aux w 0"
proof (induct rule: S.induct)
  show "balanceada_aux [] 0" by simp
next
  fix w
  assume "w ∈ S"
         "balanceada_aux w 0"
  thus "balanceada_aux ([A] @ w @ [B]) 0"
    by (simp add: balanceada_correcto_aux_1)
next
  fix v w
  assume "v ∈ S"
         "balanceada_aux v 0"
         "w ∈ S"
         "balanceada_aux w 0"
  thus "balanceada_aux (v @ w) 0"
    by (simp add: balanceada_correcto_aux_2)
qed

-- "La demostración estructurada del tercer lema es"
lemma balanceada_correcto:
  "w ∈ S ⟹ balanceada w"
by (simp add: balanceada_correcto_aux_3)

text {*
```

```
  ------------------------------------------------------------------
  Ejercicio 7. Demostrar que balanceada es un reconocedor completo de
  la gramática S; es decir,
     balanceada w  w  S
  ------------------------------------------------------------------ *}

-- "La demostración automática del primer lema auxiliar es"
lemma balanceada_completo_aux_1:
  "[A,B]  S"
using S1 S2 [where w = "[]"] by simp

-- "La demostración estructurada del primer lema auxiliar es"
lemma balanceada_completo_aux_1b:
  "[A,B]  S"
proof -
  have "[]  S" using S1 by simp
  hence "[A] @ [] @ [B]  S" using S2 [where w = "[]"] by simp
  thus "[A,B]  S" by simp
qed

-- "La demostración estructurada del segundo lema auxiliar es"
lemma balanceada_completo_aux_2:
  assumes "u  S"
  shows "v w. u = v @ w  v @ A # B # w  S"
using assms
proof (induct)
  fix v w :: "alfabeto list"
  assume "[] = v @ w"
  thus "v @ A # B # w  S" by (simp add: balanceada_completo_aux_1)
next
  fix v u w :: "alfabeto list"
  assume uS: "u  S" and
         HI: "v w. u = v @ w  v @ A # B # w  S" and
         sup: "[A] @ u @ [B] = v @ w"
  show "v @ A # B # w  S"
  proof (cases v)
    case Nil
    hence "w = A # u @ [B]" using sup by simp
    hence "w  S" using uS S2 by simp
    hence "[A,B] @ w  S" using balanceada_completo_aux_1 S3 by blast
```

```
    thus ?thesis using Nil by simp
  next
    case (Cons x v')
    show ?thesis
    proof (cases w rule:rev_cases)
      case Nil
      have "A # u @ [B]  S" using S2 uS by simp
      hence "(A # u @ [B]) @ [A,B]  S"
        using balanceada_completo_aux_1 S3 by blast
      thus ?thesis using Nil Cons sup by auto
    next
      case (snoc w' y)
      hence u: "u = v' @ w'" and [simp]: "x = A  y = B"
        using Cons sup by auto
      from u have "v' @ A # B # w'  S" by (rule HI)
      hence "A # (v' @ A # B # w') @ [B]  S"
        using S2 [where w = "v' @ A # B # w'"] by simp
      thus ?thesis using Cons snoc by auto
    qed
  qed
next
  fix v' w' v w
  assume v'S: "v'  S"
     and HIv: "v w. v' = v @ w  v @ A # B # w  S"
     and w'S: "w'  S"
     and HIw: "v w. w' = v @ w  v @ A # B # w  S"
     and sup: "v' @ w' = v @ w"
  then obtain r where "v' = v @ r  r @ w' = w  v' @ r = v  w' = r @ w"
    (is "?A  ?B")
    by (auto simp: append_eq_append_conv2)
  thus "v @ A # B # w  S"
  proof
    assume A: ?A
    hence "v @ A # B # r  S" using HIv by blast
    hence "(v @ A # B # r) @ w'  S" using w'S by (rule S3)
    thus ?thesis using A by auto
  next
    assume B: ?B
    hence "r @ A # B # w  S" using HIw by blast
    with v'S have "v' @ (r @ A # B # w)  S" by (rule S3)
```

```
    thus ?thesis using B by auto
  qed
qed

-- "La demostración estructurada del tercer lema auxiliar es"
lemma balanceada_completo_aux_3:
  "balanceada_aux w n  replicate n A @ w  S"
proof (induct w n rule: balanceada_aux.induct)
  assume "balanceada_aux [] 0"
  thus "replicate 0 A @ []  S" using S1 by simp
next
  fix w n
  assume "balanceada_aux w (Suc n)  replicate (Suc n) A @ w  S"
     and "balanceada_aux (A # w) n"
  thus "replicate n A @ A # w  S"
    by (simp add: replicate_app_Cons_same)
next
  fix w n
  assume "balanceada_aux w n  replicate n A @ w  S"
     and "balanceada_aux (B # w) (Suc n)"
  thus "replicate (Suc n) A @ B # w  S"
    by (simp add: balanceada_completo_aux_2
        replicate_app_Cons_same[symmetric])
next
  fix v
  assume "balanceada_aux (B # v) 0"
  thus "replicate 0 A @ B # v  S" by simp
next
  fix v
  assume "balanceada_aux [] (Suc v)"
  thus "replicate (Suc v) A @ []  S" by simp
qed

-- "La demostración del lema es"
lemma balanceada_completo:
  assumes "balanceada w"
  shows   " w  S"
proof -
  have "balanceada_aux w 0" using assms by simp
  hence "replicate 0 A @ w  S" by (rule balanceada_completo_aux_3)
```

```
  thus "w  S" by simp
qed

end
```