Introducción a la demostracción 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, 16 de agosto de 2013

Índice general

Introducción

Este libro es una recopilación de los temas y relaciones de ejercicios del curso de *Razonamiento automático*[1]. El curso es una introducción a la demostración asistida por ordenador con Isabelle/HOL[2] y consta de tres niveles.

En el primer nivel se presenta la formalización de las demostraciones por deducción natural. La presentación se basa en la realizada en los cursos de *Lógica informática*[3] (de 2º del Grado en Informática) y *Lógica matemática y fundamentos*[4] (de 3º del Grado en Matemáticas), en concreto en los temas de deducción natural proposicional[5] y de de primer orden[6] En este nivel se incluye los 3 primeros temas y las 6 primeras relaciones de ejercicios.

En el segundo nivel se presenta la programación funcional en Isabelle/HOL y el razonamiento sobre programas. La presentación se basa en la introducción a la programación con Haskell realizada en los cursos de *Informática*[7] (de 1º del Grado en Matemáticas) y *Programación declarativa*[8] (de 3º del Grado en Informática), en concreto en el tema 8[9] (para el razonamiento sobre programas) y en los restantes 9 primeros temas. En este nivel se incluye los temas 4, 5 y 6 y las relaciones de ejercicios desde la 7 a la 21.

En el tercer nivel se presentan casos de estudios y extensiones de la lógica para trabajar con conjuntos, relaciones y funciones. En este nivel se incluye los temas y y 8 y las relaciones de ejercicios 22 y 23.

Tanto los temas como las relaciones de ejercicios se presentan como teorías de Isabelle/HOL (versión Isabelle2013). Conforme se necesitan se va comentando los elementos del Isabelle/HOL.

De Isabelle2013 se dispone se dispone de versiones libres para Linux, Windows y Mac OS X. Para instalarlo basta seguir la guía de instalación[10].

[1] http://www.cs.us.es/~jalonso/cursos/m-ra-12
[2] http://www.cl.cam.ac.uk/research/hvg/Isabelle
[3] http://www.cs.us.es/~jalonso/cursos/li-12
[4] http://www.cs.us.es/~jalonso/cursos/lmf-12
[5] http://www.cs.us.es/~jalonso/cursos/li-12/temas/tema-2.pdf
[6] http://www.cs.us.es/~jalonso/cursos/li-12/temas/tema-8.pdf
[7] http://www.cs.us.es/~jalonso/cursos/i1m-12
[8] http://www.cs.us.es/cursos/pd-2012
[9] http://www.cs.us.es/~jalonso/cursos/i1m-12/temas/tema-8t.pdf
[10] http://www.cl.cam.ac.uk/research/hvg/Isabelle/installation.html

Parte I

Temas

Tema 1

Deducción natural en lógica proposicional

```
header {* Tema 1: Deducción natural proposicional con Isabelle/HOL *}

theory T1_Deduccion_natural_en_logica_proposicional
imports Main
begin

text {*
  En este tema 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
thm ejemplo_1_1
text {*
  Notas sobre el lenguaje: En la demostración anterior se ha usado
  · "assumes" para indicar las hipótesis,
  · "and" para separar las hipótesis,
  · "shows" para indicar la conclusión,
  · "proof" para iniciar la prueba,
  · "qed" para terminar la pruebas,
  · "-" (después de "proof") para no usar el método por defecto,
  · "have" para establecer un paso,
  · "using" para usar hechos en un paso,
  · "by (rule ..)" para indicar la regla con la que se peueba un hecho,
  · "show" para establecer la conclusión.

  Notas sobre la lógica: Las reglas de la conjunción son
  · conjI:      ⟦P; Q⟧ ⟹ P ∧ Q
  · conjunct1:  P ∧ Q ⟹ P
  · 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
```

```
  · ".." 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
  · "assms(n)" para indicar la hipótesis n y
  · "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
  · "assms" para indicar las hipótesis y
  · "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
  · "⟦ ... ⟧" para representar las hipótesis,
  · ";" para separar las hipótesis y
  · "⟹" 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
  · "proof (rule r)" para indicar que se hará la demostración con la
    regla r,
  · "next" para indicar el comienzo de la prueba del siguiente
    subobjetivo,
  · "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
```

```
  · "." para indicar por el hecho actual. *}

subsection {* Reglas de la doble negación *}

text {*
  La regla de eliminación de la doble negación es
  · notnotD: ¬¬ P ⟹ P

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

lemma notnotI [intro!]: "P ⟹ ¬¬ P"
by auto

text {*
  Ejemplo 2. (p. 5)
       p, ¬¬(q ∧ r) ⊢ ¬¬p ∧ r
*}

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

-- "La demostración estructurada es"
lemma ejemplo_2_2:
  assumes "p"
          "¬¬(q ∧ r)"
  shows   "¬¬p ∧ 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
  · "hence" para indicar que se tiene por el hecho anterior,
  · ‘...‘ para referenciar un hecho y
  · "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
  · 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
  · mt: ⟦F ⟶ G; ¬G⟧ ⟹ ¬F
  aunque, de momento, sin detallar su demostración.
*}

lemma mt: "⟦F ⟶ G; ¬G⟧ ⟹ ¬F"
by auto

text {*
  Ejemplo 5 (p. 7). Demostrar
     p ⟶ (q ⟶ r), p, ¬r ⊢ ¬q
 *}

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

-- "La demostración estructurada es"
lemma ejemplo_5_2:
  assumes "p ⟶ (q ⟶ r)"
          "p"
          "¬r"
  shows   "¬q"
proof -
  have "q ⟶ r" using assms(1,2) ..
  thus "¬q" 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
  · impI: (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
  · "{ ... }" 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
  · "moreover" para separar los bloques y
  · "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
  · "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
  · FalseE: False ⟹ P
  La regla de eliminación de la negación es
  · notE: ⟦¬P; P⟧ ⟹ R
  La regla de introducción de la negación es
  · notI: (P ⟹ False) ⟹ ¬P
*}

text {*
  Ejemplo 15 (p. 15). Demostrar
     ¬p ∨ q ⊢ p ⟶ q
*}

-- "La demostración detallada es"
lemma ejemplo_15_1:
  assumes 1: "¬p ∨ 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 ⟶ ¬q"
  shows "¬p"
proof (rule notI)
  assume 3: "p"
  have 4: "q" using 1 3 by (rule mp)
  have 5: "¬q" 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 ⟶ ¬q"
  shows "¬p"
proof
  assume "p"
  have "q" using assms(1) ‘p‘ ..
  have "¬q" using assms(2) ‘p‘ ..
  thus False using ‘q‘ ..
qed

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

subsection {* Reglas del bicondicional *}

text {*
  La regla de introducción del bicondicional es
  · iffI: ⟦P ⟹ Q; Q ⟹ P⟧ ⟹ P ⟷ Q
  Las reglas de eliminación del bicondicional son
  · iffD1: ⟦Q ⟷ P; Q⟧ ⟹ P
  · 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 "¬F"
```

```
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 "¬¬F"
proof (rule notI)
  assume 2: "¬F"
  show False using 2 1 by (rule notE)
qed

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

-- "La demostración automática es"
lemma ejemplo_21_3:
  assumes "F"
  shows "¬¬F"
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
```

```
  · ccontr: (¬P ⟹ False) ⟹ P
*}

subsubsection {* Ley del tercio excluso *}

text {*
  La ley del tercio excluso es
  · excluded_middle: ¬P ∨ 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 ∨ ¬F"
proof (rule ccontr)
  assume 1: "¬(F ∨ ¬F)"
  thus False
  proof (rule notE)
    show "F ∨ ¬F"
    proof (rule disjI2)
      show "¬F"
      proof (rule notI)
        assume 2: "F"
        hence 3: "F ∨ ¬F" by (rule disjI1)
        show False using 1 3 by (rule notE)
      qed
    qed
  qed
qed

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

Tema 2

Deducción natural en lógica de primer orden

```
header {* Tema 2: Deducción natural en lógica de primer orden *}

theory T2_Deduccion_natural_en_logica_de_primer_orden
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 8 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
  · allE:    ⟦∀x. P x; P a ⟹ R⟧ ⟹ R
  · 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
  · exI:     P a ⟹ ∃x. P x
  · 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 ‘¬(∃x. ¬P(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 "¬(∀x. P(x))"
  shows   "∃x. ¬P(x)"
proof (rule ccontr)
  assume "¬(∃x. ¬P(x))"
  have "∀x. P(x)"
  proof
    fix a
    show "P(a)"
    proof (rule ccontr)
      assume "¬P(a)"
      hence "∃x. ¬P(x)" ..
      with `¬(∃x. ¬P(x))` show False ..
    qed
  qed
  with assms show False ..
qed

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

text {*
  Ejemplo 5.2 (p. 16). Demostrar
     ∃x. ¬(P x)  ⊢ ¬∀x. P x *}

-- "La demostración detallada es"
lemma ejemplo_5_2a:
  assumes "∃x. ¬P(x)"
  shows   "¬(∀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:
  "(¬(∀x. P(x))) ⟷ (∃x. ¬P(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 iffI)
  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 ejemplo_11b:
  assumes "s = t"
  shows   "t = s"
using assms
by auto

end
```

Tema 3

Resumen de Isabelle/Isar y de la lógica

```
header {* Tema 3: Resumen del lenguaje Isabelle/Isar y las reglas de la lógica *}

theory T3
imports Main
begin

section {* Sintaxis (simplificada) de Isabelle/Isar *}

text {*
  Representación de lemas (y teoremas)
  · Un lema (o teorema) comienza con una etiqueta seguida por algunas
    premisas y una conclusión.
  · Las premisas se introducen con la palabra "assumes" y se separan
    con "and".
  · Cada premisa puede etiquetarse para referenciarse en la demostración.
  · 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:
  · TrueI:          True
  · conjI:          ⟦P; Q⟧ ⟹ P ∧ Q
  · conjunct1:      P ∧ Q ⟹ P
  · conjunct2:      P ∧ Q ⟹ Q
  · conjE:          ⟦P ∧ Q; ⟦P; Q⟧ ⟹ R⟧ ⟹ R
  · disjI1:         P ⟹ P ∨ Q
  · disjI2:         Q ⟹ P ∨ Q
  · disjE:          ⟦P ∨ Q; P ⟹ R; Q ⟹ R⟧ ⟹ R
  · notI:           (P ⟹ False) ⟹ ¬P
  · notE:           ⟦¬P; P⟧ ⟹ R
  · FalseE:         False ⟹ P
  · notnotD:        ¬¬ P ⟹ P
```

```
  · impI:           (P ⟹ Q) ⟹ P ⟶ Q
  · impE:           ⟦P ⟶ Q; P; Q ⟹ R⟧ ⟹ R
  · mp:             ⟦P ⟶ Q; P⟧ ⟹ Q
  · iff:            (P ⟶ Q) ⟶ (Q ⟶ P) ⟶ P = Q
  · iffI:           ⟦P ⟹ Q; Q ⟹ P⟧ ⟹ P = Q
  · iffD1:          ⟦Q = P; Q⟧ ⟹ P
  · iffD2:          ⟦P = Q; Q⟧ ⟹ P
  · iffE:           ⟦P = Q; ⟦P ⟶ Q; Q ⟶ P⟧ ⟹ R⟧ ⟹ R
  · ccontr:         (¬P ⟹ False) ⟹ P
  · classical:      (¬P ⟹ P) ⟹ P
  · excluded_middle: ¬P ∨ P
  · disjCI:         (¬Q ⟹ P) ⟹ P ∨ Q
  · impCE:          ⟦P ⟶ Q; ¬P ⟹ R; Q ⟹ R⟧ ⟹ R
  · iffCE:          ⟦P = Q; ⟦P; Q⟧ ⟹ R; ⟦¬P; ¬Q⟧ ⟹ R⟧ ⟹ R
  · swap:           ⟦¬P; ¬R ⟹ P⟧ ⟹ R
*}

text {*
  Resumen de reglas de cuantificadores:
  · allE:    ⟦∀x. P x; P y ⟹ R⟧ ⟹ R
  · allI:    (⋀x. P x) ⟹ ∀x. P x
  · exI:     P a ⟹ ∃x. P x
  · exE:     ⟦∃x. P x; ⋀x. P x ⟹ Q⟧ ⟹ Q
*}


text {*
  Resumen de reglas de la igualdad:
  · refl:       t = t
  · subst:      ⟦s = t; P s⟧ ⟹ P t
  · trans:      ⟦r = s; s = t⟧ ⟹ r = t
  · sym:        s = t ⟹ t = s
  · not_sym:    t ≠ s ⟹ s ≠ t
  · ssubst:     ⟦t = s; P s⟧ ⟹ P t
  · box_equals: ⟦a = b; a = c; b = d⟧ ⟹ c = d
  · arg_cong:   x = y ⟹ f x = f y
  · fun_cong:   f = g ⟹ f x = g x
  · 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).
*}
```

Tema 4

Programación funcional con Isabelle/HOL

```
header {* Tema 4: Programación funcional en Isabelle *}

theory T4
imports Main
begin

section {* Introducción *}

text {*
  En este tema 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 (¬, ∧, ∨, ⟶ 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 "¬True" -- "= 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 descomposión 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 "(λx. 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
```

Tema 5

Razonamiento sobre programas

```
header {* Tema 5: Razonamiento sobre programas *}

theory T5
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://goo.gl/Imvyt *}

section {* Razonamiento ecuacional *}

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. (p.6) Demostrar que
     intercambia (intercambia (x,y)) = (x,y)
  ------------------------------------------------------------------ *}

-- "La demostración detallada es"
lemma "intercambia (intercambia (x,y)) = (x,y)"
proof -
  have "intercambia (intercambia (x,y)) = intercambia (y,x)"
    by (simp only: intercambia.simps)
  also have "... = (x,y)"
    by (simp only: intercambia.simps)
  finally show "intercambia (intercambia (x,y)) = (x,y)" by simp
qed

text {*
  El razonamiento ecuacional se realiza usando la combinación de "also"
  (además) y "finally" (finalmente). *}

-- "La demostración estructurada es"
```

```
lemma "intercambia (intercambia (x,y)) = (x,y)"
proof -
  have "intercambia (intercambia (x,y)) = intercambia (y,x)"  by simp
  also have "... = (x,y)" by simp
  finally show "intercambia (intercambia (x,y)) = (x,y)" by simp
qed

-- "La demostración automática es"
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. (p. 9) Demostrar que
     inversa [x] = [x]
  ---------------------------------------------------------------- *}

-- "La demostración detallada es"
lemma "inversa [x] = [x]"
proof -
  have "inversa [x] = inversa (x#[])" by simp
  also have "... = (inversa []) @ [x]" by (simp only: inversa.simps(2))
  also have "... = [] @ [x]" by (simp only: inversa.simps(1))
  also have "... = [x]" by (simp only: append_Nil)
  finally show "inversa [x] = [x]" by simp
qed
```

```
-- "La demostración estructurada es"
lemma "inversa [x] = [x]"
proof -
  have "inversa [x] = inversa (x#[])" by simp
  also have "... = (inversa []) @ [x]" by simp
  also have "... = [] @ [x]" by simp
  also have "... = [x]" by simp
  finally show "inversa [x] = [x]" by simp
qed

-- "La demostración automática es"
lemma "inversa [x] = [x]"
by simp

section {* Razonamiento por inducción sobre los naturales *}

text {*
  [Principio de inducción sobre los naturales] 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 sobre los naturales está
  formalizado en el teorema nat.induct y puede verse con
     thm nat.induct
*}

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. (p. 18) Demostrar que
     longitud (repite n x) = n
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma "longitud (repite n x) = n"
proof (induct n)
  show "longitud (repite 0 x) = 0" by simp
next
  fix n
  assume HI: "longitud (repite n x) = n"
  have "longitud (repite (Suc n) x) = longitud (x # (repite n x))"
    by simp
  also have "... = 1 + longitud (repite n x)" by simp
  also have "... = 1 + n" using HI by simp
  finally show "longitud (repite (Suc n) x) = Suc n" by simp
qed

-- "La demostración automática es"
lemma "longitud (repite n x) = n"
by (induct n) auto

section {* Razonamiento por inducción sobre listas *}

text {*
  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 otra lista que también tiene la
  propiedad.

  En Isabelle el principio de inducción sobre listas está formalizado
  mediante el teorema list.induct que puede verse con
     thm list.induct
*}

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. (p. 24) Demostrar que
     conc xs (conc ys zs) = (conc xs ys) zs
  ------------------------------------------------------------ *}

-- "La demostración estructurada es"
lemma "conc xs (conc ys zs) = conc (conc xs ys) zs"
proof (induct xs)
  show "conc [] (conc ys zs) = conc (conc [] ys) zs" by simp
next
  fix x xs
  assume HI: "conc xs (conc ys zs) = conc (conc xs ys) zs"
  have "conc (x # xs) (conc ys zs) = x # (conc xs (conc ys zs))" by simp
  also have "... = x # (conc (conc xs ys) zs)" using HI by simp
  also have "... = conc (conc (x # xs) ys) zs" by simp
  finally show "conc (x # xs) (conc ys zs) = conc (conc (x # xs) ys) zs" by simp
qed

-- "La demostración automática es"
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 = [a2]
  ys = [a1] *}

text {* ----------------------------------------------------------------
  Ejemplo 12. (p. 28) Demostrar que
     conc xs [] = xs
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma "conc xs [] = xs"
proof (induct xs)
  show "conc [] [] = []" by simp
next
  fix x xs
  assume HI: "conc xs [] = xs"
  have "conc (x # xs) [] = x # (conc xs [])" by simp
  also have "... = x # xs" using HI by simp
  finally show "conc (x # xs) [] = x # xs" by simp
qed

-- "La demostración automática es"
lemma "conc xs [] = xs"
by (induct xs) auto

text {* ----------------------------------------------------------------
  Ejemplo 13. (p. 30) Demostrar que
     longitud (conc xs ys) = longitud xs + longitud ys
  ---------------------------------------------------------------- *}

-- "La demostración automática es"
lemma "longitud (conc xs ys) = longitud xs + longitud ys"
proof (induct xs)
  show "longitud (conc [] ys) = longitud [] + longitud ys" by simp
next
  fix x xs
  assume HI: "longitud (conc xs ys) = longitud xs + longitud ys"
```

```
  have "longitud (conc (x # xs) ys) = longitud (x # (conc xs ys))"
    by simp
  also have "... = 1 + longitud (conc xs ys)" by simp
  also have "... = 1 + longitud xs + longitud ys" using HI by simp
  also have "... = longitud (x # xs) + longitud ys" by simp
  finally show "longitud (conc (x # xs) ys) =
                longitud (x # xs) + longitud ys" by simp
qed

-- "La demostración automática es"
lemma "longitud (conc xs ys) = longitud xs + longitud ys"
by (induct xs) auto

section {* Inducción correspondiente a la definición recursiva *}

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 {*
  La definición coge genera el esquema de inducción coge.induct:
     ⟦⋀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 16. (p. 35) Demostrar que
     conc (coge n xs) (elimina n xs) = xs
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma "conc (coge n xs) (elimina n xs) = xs"
proof (induct rule: coge.induct)
  fix n
  show "conc (coge n []) (elimina n []) = []" by simp
next
  fix x xs
  show "conc (coge 0 (x#xs)) (elimina 0 (x#xs)) = x#xs" by simp
next
  fix n x xs
  assume HI: "conc (coge n xs) (elimina n xs) = xs"
  have "conc (coge (Suc n) (x#xs)) (elimina (Suc n) (x#xs)) =
        conc (x#(coge n xs)) (elimina n xs)" by simp
  also have "... = x#(conc (coge n xs) (elimina n xs))" by simp
  also have "... = x#xs" using HI by simp
  finally show "conc (coge (Suc n) (x#xs)) (elimina (Suc n) (x#xs)) = x#xs"
    by simp
qed

-- "La demostración automática es"
lemma "conc (coge n xs) (elimina n xs) = xs"
```

```
by (induct rule: coge.induct) auto

section {* Razonamiento por casos *}

text {*
  Distinción de casos sobre listas:
  · El método de distinción de casos se activa con (cases xs) donde xs
    es del tipo lista.
  · "case Nil" es una abreviatura de
       "assume Nil: xs =[]".
  · "case Cons" es una abreviatura de
       "fix ? ?? assume Cons: xs = ? # ??"
    donde ? y ?? son variables anónimas. *}

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 (p. 39) . Demostrar que
     esVacia xs = esVacia (conc xs xs)
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma "esVacia xs = esVacia (conc xs xs)"
proof (cases xs)
  assume "xs = []"
  thus "esVacia xs = esVacia (conc xs xs)" by simp
next
```

```
  fix y ys
  assume "xs = y#ys"
  thus "esVacia xs = esVacia (conc xs xs)" by simp
qed

-- "La demostración estructurada simplificada es"
lemma "esVacia xs = esVacia (conc xs xs)"
proof (cases xs)
  case Nil
  thus "esVacia xs = esVacia (conc xs xs)" by simp
next
  case Cons
  thus "esVacia xs = esVacia (conc xs xs)" by simp
qed

-- "La demostración automática es"
lemma "esVacia xs = esVacia (conc xs xs)"
by (cases xs) auto

section {* Heurística de generalización *}

text {*
  Heurística de generalización: Cuando se use demostración estructural,
  cuantificar universalmente las variables libres (o, equivalentemente,
  considerar las variables libres como variables arbitrarias). *}

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. (p. 44) Demostrar que
     inversaAcAux xs ys = (inversa xs) @ ys
  ------------------------------------------------------------ *}

-- "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

-- "La demostración automática es"
lemma "inversaAcAux xs ys = (inversa xs)@ys"
by (induct xs arbitrary: ys) auto

text {* ------------------------------------------------------------
  Ejemplo 21. (p. 43) Demostrar que
     inversaAc xs = inversa xs
  ------------------------------------------------------------ *}

-- "La demostración automática es"
corollary "inversaAc xs = inversa xs"
by (simp add: inversaAcAux_es_inversa)

section {* Demostración por inducción para funciones de orden superior *}
```

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

text {* ------------------------------------------------------------
  Ejemplo 24. (p. 45) Demostrar que
     sum (map (λx. 2*x) xs) = 2 * (sum xs)
  ------------------------------------------------------------ *}

-- "La demostración estructurada es"
lemma "sum (map (λx. 2*x) xs) = 2 * (sum xs)"
proof (induct xs)
  show "sum (map (λx. 2*x) []) = 2 * (sum [])" by simp
next
  fix a xs
  assume HI: "sum (map (λx. 2*x) xs) = 2 * (sum xs)"
```

```
  have "sum (map (λx. 2*x) (a#xs)) = sum ((2*a)#(map (λx. 2*x) xs))"
    by simp
  also have "... = 2*a + sum (map (λx. 2*x) xs)" by simp
  also have "... = 2*a + 2*(sum xs)" using HI by simp
  also have "... = 2*(a + sum xs)" by simp
  also have "... = 2*(sum (a#xs))" by simp
  finally show "sum (map (λx. 2*x) (a#xs)) = 2*(sum (a#xs))" by simp
qed

-- "La demostración automática es"
lemma "sum (map (λx. 2*x) xs) = 2 * (sum xs)"
by (induct xs) auto

text {* ---------------------------------------------------------------
  Ejemplo 25. (p. 48) Demostrar que
     longitud (map f xs) = longitud xs
  ---------------------------------------------------------------- *}

-- "La demostración estructurada es"
lemma "longitud (map f xs) = longitud xs"
proof (induct xs)
  show "longitud (map f []) = longitud []" by simp
next
  fix a xs
  assume HI: "longitud (map f xs) = longitud xs"
  have "longitud (map f (a#xs)) = longitud (f a # (map f xs))" by simp
  also have "... = 1 + longitud (map f xs)" by simp
  also have "... = 1 + longitud xs" using HI by simp
  also have "... = longitud (a#xs)" by simp
  finally show "longitud (map f (a#xs)) = longitud (a#xs)" by simp
qed

-- "La demostración automática es"
lemma "longitud (map f xs) = longitud xs"
by (induct xs) auto

section {* Referencias *}

text {*
  · J.A. Alonso. "Razonamiento sobre programas" http://goo.gl/R0603
```

```
  · G. Hutton. "Programming in Haskell". Cap. 13 "Reasoning about
    programms".
  · S. Thompson. "Haskell: the Craft of Functional Programming, 3rd
    Edition. Cap. 8 "Reasoning about programms".
  · L. Paulson. "ML for the Working Programmer, 2nd Edition". Cap. 6.
    "Reasoning about functional programs".
*}

end
```

Tema 6

Razonamiento por casos y por inducción

```
header {* Tema 6: Razonamiento por casos y por inducción *}

theory T6
imports Main Parity
begin

text {*
  En este tema se amplían los métodos de demostración por casos y por
  inducción iniciados en el tema anterior.
*}

section {* Razonamiento por distinción de casos *}

subsection {* Distinción de casos booleanos *}

text {*
  Ejemplo de demostración por distinción de casos booleanos:
     ¬A ∨ A
*}

-- "La demostración estructurada es"
lemma "¬A ∨ A"
proof cases
  assume "A"
  thus ?thesis ..
next
  assume "¬A"
```

```
  thus ?thesis ..
qed

-- "La demostración detallada es"
lemma "¬A ∨ 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 {*
  Ejemplo de 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
  · La expresión "case True" es una abreviatura de F.
  · La expresión "case False" es una abreviatura de ¬F.
  · Ventajas de "cases" con nombre:
    · reduce la escritura de la fórmula y
    · es independiente del orden de los casos.
*}

subsection {* Distinción de casos sobre otros tipos de datos *}

text {*
  Ejemplo de 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:
  · El método de distinción de casos se activa con (cases xs) donde xs
    es del tipo lista.
  · "case Nil" es una abreviatura de
       "assume Nil: xs =[]".
  · "case Cons" es una abreviatura de
       "fix ? ?? assume Cons: xs = ? # ??"
```

```
    donde ? y ?? son variables anónimas.
*}

text {*
  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
*}

text {*
  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. Por ejemplo,
     suma_impares 3  =  9
*}

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

value "suma_impares 3"

text {*
  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

-- "La demostración usando patrones es"
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 {*
  Ejemplo de definición con existenciales.
  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 {*
  [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:
  · En Isabelle puede hacerse inducción estructural sobre cualquier tipo
    recursivo.
  · La inducción matemática es la inducción estructural sobre el tipo de
    los naturales.
  · El esquema de inducción estructural sobre listas es
    · list.induct: ⟦P []; ⋀x ys. P ys ⟹ P (x # ys)⟧ ⟹ P zs
  · 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.
  · En Isabelle el principio de inducción sobre listas está formalizado
    mediante el teorema list.induct que puede verse con
       thm list.induct
*}

text {*
  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)"
*}

text {*
  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 de definición de tipos recursivos:
  Definir un tipo de dato para los árboles binarios.
*}

datatype 'a arbolB = Hoja "'a"
                   | Nodo "'a" "'a arbolB" "'a arbolB"

text {*
  Ejemplo de definición sobre árboles binarios:
  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 de demostración sobre árboles binarios:
  Demostrar que la función "espejo" es 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
  · append_assoc:       (xs @ ys) @ zs = xs @ (ys @ zs)
  · append.append_Cons: (x#xs)@ys = x#(xs@ys)
  · 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)"

-- "Ejemplo de evaluación"
value "ack 2 3" (* devuelve 9 *)

text {*
  Esquema de inducción correspondiente a una función:
  · 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
*}

text {*
  Ejemplo de demostración por la inducción correspondiente a una función:
  Para todos m y n, A(m,n) > n.
*}

-- "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]
  · Un árbol de tipo a es una hoja o un nodo de tipo a junto con un
    bosque de tipo a.
  · 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 {*
  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
*}

text {*
  Ejemplos de definición por recursión cruzada:
  · aplana_arbol a) es la lista obtenida aplanando el árbol a.
  · (aplana_bosque b) es la lista obtenida aplanando el bosque b.
  · (map_arbol a h) es el árbol obtenido aplicando la función h a
    todos los nodos del árbol a.
  · (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 {*
  Ejemplo de dempstración por inducción cruzada:
  · aplana_arbol (map_arbol a h) = map h (aplana_arbol a)
  · 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
```

Tema 7

Caso de estudio: Compilación de expresiones

```
header {* 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) (λx. x+1) = 3 ∧
   valor (App (op +) (Const 3) (Var 2)) (λx. x+1) = 6 ∧
   valor (App (op +) (Const 3) (Var 2)) (λx. x+4) = 9"
by simp

section {* La máquina de pila *}

text {*
  Nota. La máquina de pila tiene tres clases de intrucciones:
  · cargar en la pila una constante,
  · cargar en la pila el contenido de una dirección y
  · 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] (λx. x+4) [7] = [3,6,7] ∧
   ejec [ILoad 2, IConst 3, IApp (op +)] (λx. 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
```

Tema 8

Conjuntos, funciones y relaciones

```
header {* Tema 8: Conjuntos, funciones y relaciones *}

theory T8a
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 τ) y el conjunto tiene tipo (en el ejemplo, "τ set").

  Reglas de la intersección:
  · IntI:  ⟦c ∈ A; c ∈ B⟧ ⟹ c ∈ A ∩ B
  · IntD1: c ∈ A ∩ B ⟹ c ∈ A
  · IntD2: c ∈ A ∩ B ⟹ c ∈ B

  Nota. Propiedades del complementario:
  · Compl_iff: (c ∈ - A) = (c ∉ A)
  · 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:
  · Diff_disjoint:   A ∩ (B - A) = {}
  · Compl_partition: A ∪ - A = UNIV

  Nota. Reglas de la relación de subconjunto:
  · subsetI: (⋀x. x ∈ A ⟹ x ∈ B) ⟹ A ⊆ B
  · subsetD: ⟦A ⊆ B; c ∈ A⟧ ⟹ c ∈ B
*}

text {*
  Ejemplo: A ∪ B ⊆ C syss A ⊆ C ∧ B ⊆ C.
*}

lemma "(A ∪ B ⊆ C) = (A ⊆ C ∧ B ⊆ C)"
by blast

text {*
  Ejemplo: A ⊆ -B syss B ⊆ -A.
*}

lemma "(A ⊆ -B) = (B ⊆ -A)"
by blast

text {*
  Principio de extensionalidad de conjuntos:
  · set_ext: (⋀x. (x ∈ A) = (x ∈ B)) ⟹ A = B

  Reglas de la igualdad de conjuntos:
  · equalityI: ⟦A ⊆ B; B ⊆ A⟧ ⟹ A = B
  · equalityE: ⟦A = B; ⟦A ⊆ B; B ⊆ A⟧ ⟹ P⟧ ⟹ P
*}

text {*
  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:
  · El conjunto vacío es un conjunto finito.
    · emptyI: "finite {}"
  · Si se le añade un elemento a un conjunto finito se obtiene otro
    conjunto finito.
    · 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.
*}

text {*
  Ejemplo: {a,b} ∪ {c,d} = {a,b,c,d}
*}

lemma "{a,b} ∪ {c,d} = {a,b,c,d}"
by blast

text {*
  Ejemplo de conjetura falsa y su refutación.
*}

lemma "{a,b} ∩ {b,c} = {b}"
nitpick
oops

text {*
  Ejemplo con la 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.
  · sumaCuadradosConj A es la suma de los cuadrados de los elementos A.
*}

definition sumaConj :: "nat set ⇒ nat" where
  "sumaConj S ≡ ∑S"

value "sumaConj {2,5,3}" -- "= 10"

definition productoConj :: "nat set ⇒ nat" where
  "productoConj S ≡ ∏S"

definition sumaCuadradosConj :: "nat set ⇒ nat" where
  "sumaCuadradosConj S ≡ setsum (λx. x*x) S"

value "sumaCuadradosConj {2,5,3}" -- "= 38"

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
  · El conjunto vacío tiene la propiedad P.
  · 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
  · finite_induct: ⟦finite F;
                    P {};
                    ⋀x F. ⟦finite F; x ∉ F; P F⟧ ⟹ P ({x} ∪ F)⟧
                   ⟹ P F
*}

text {*
  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.
*}

-- "La demostración automática es"
lemma "finite S ⟹ ∀x∈S. x ≤ sumaConj S"
by (induct rule: finite_induct) auto

-- "La demostración estructurada es"
lemma sumaConj_acota:
  "finite S ⟹ ∀x∈S. 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: "∀ x∈F. 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):
  · mem_Collect_eq: (a ∈ {x. P x}) = P a
  · Collect_mem_eq: {x. x ∈ A} = A
*}

text {*
  Ejemplo de comprensión: {x. P x ∨ x ∈ A} = {x. P x} ∪ A
*}

lemma "{x. P x ∨ x ∈ A} = {x. P x} ∪ A"
by blast

text {*
  Ejemplo de comprensión: {x. P x ⟶ Q x} = -{x. P x} ∪ {x. Q x}
*}

lemma "{x. P x ⟶ Q x} = -{x. P x} ∪ {x. Q x}"
```

```
by blast

text {*
  Ejemplo con la sintaxis general de comprensión.
     {p*q | p q. p ∈ prime ∧ q ∈ prime} =
     {z. ∃p q. z = p*q ∧ p ∈ prime ∧ q ∈ prime}
*}

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:
   · x ∈ A  es equivalente a A(x).
   · {x. P} es equivalente a λx. P.
*}

text {*
  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 {*
  Ejemplo con las reglas de intersección y comprensión: El conjunto de
  los pares es disjunto con el de los impares.
*}

-- "La demostración detallada es"
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

-- "La demostración estructurada es"
lemma "x ∉ (Pares ∩ Impares)"
proof
  fix x assume S: "x ∈ (Pares ∩ Impares)"
  hence "x ∈ Pares" ..
  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" ..
  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

-- "La demostración automática es"
lemma "x ∉ (Pares ∩ Impares)"
by (auto simp add: Pares_def Impares_def mem_Collect_eq, arith)

subsection {* Cuantificadores acotados *}
```

```
text {*
  Reglas de cuantificador universal acotado ("bounded"):
  · ballI: (⋀x. x ∈ A ⟹ P x) ⟹ ∀x∈A. P x
  · bspec: ⟦∀x∈A. P x; x ∈ A⟧ ⟹ P x

  Reglas de cuantificador existencial acotado ("bounded"):
  · bexI: ⟦P x; x ∈ A⟧ ⟹ ∃x∈A. P x
  · bexE: ⟦∃x∈A. P x; ⋀x. ⟦x ∈ A; P x⟧ ⟹ Q⟧ ⟹ Q

  Reglas de la unión indexada:
  · UN_iff: (b ∈ (⋃x∈A. B x)) = (∃x∈A. b ∈ B x)
  · UN_I:   ⟦a ∈ A; b ∈ B a⟧ ⟹ b ∈ (⋃x∈A. B x)
  · UN_E:   ⟦b ∈ (⋃x∈A. B x); ⋀x. ⟦x ∈ A; b ∈ B x⟧ ⟹ R⟧ ⟹ R

  Reglas de la unión de una familia:
  · Union_def: ⋃S = (⋃x∈S. x)
  · Union_iff: (A ∈ ⋃C) = (∃X∈C. A ∈ X)

  Reglas de la intersección indexada:
  · INT_iff: (b ∈ (⋂x∈A. B x)) = (∀x∈A. b ∈ B x)
  · INT_I:   (⋀x. x ∈ A ⟹ b ∈ B x) ⟹ b ∈ (⋂x∈A. B x)
  · INT_E:   ⟦b ∈ (⋂x∈A. B x); b ∈ B a ⟹ R; a ∉ A ⟹ R⟧ ⟹ R

  Reglas de la intersección de una familia:
  · Inter_def: ⋂S = (⋂x∈S. x)
  · Inter_iff: (A ∈ ⋂C) = (∀X∈C. A ∈ X)

  Abreviaturas:
  · "Collect P" es lo mismo que "{x. P}".
  · "All P"     es lo mismo que "∀x. P x".
  · "Ex P"      es lo mismo que "∃x. P x".
  · "Ball A P"  es lo mismo que "∀x∈A. P x".
  · "Bex A P"   es lo mismo que "∃x∈A. 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".
*}

text {*
  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:
  · Cardinal de la unión de conjuntos finitos:
    card_Un_Int: ⟦finite A; finite B⟧
                 ⟹ card A + card B = card (A ∪ B) + card (A ∩ B)"
  · 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:
  · ext: (⋀x. f x = g x) ⟹ f = g

  Actualización de funciones
  · fun_upd_apply: (f(x := y)) z = (if z = x then y else f z)
  · fun_upd_upd:   f(x := y, x := z) = f(x := z)

  Función identidad
```

```
  · id_def: id ≡ λx. x

  Composición de funciones:
  · o_def: f ∘ g = (λx. f (g x))

  Asociatividad de la composición:
  · o_assoc: f ∘ (g ∘ h) = (f ∘ g) ∘ h
*}

subsection {* Funciones inyectivas, suprayectivas y biyectivas *}

text {*
  Función inyectiva sobre A:
  · inj_on_def: inj_on f A ≡ ∀x∈A. ∀y∈A. f x = f y ⟶ x = y

  Nota. "inj f" es una abreviatura de "inj_on f UNIV".

  Función suprayectiva:
  · surj_def: surj f ≡ ∀y. ∃x. y = f x

  Función biyectiva:
  · bij_def: bij f ≡ inj f ∧ surj f

  Propiedades de las funciones inversas:
  · inv_f_f:      inj f ⟹ inv f (f x) = x
  · surj_f_inv_f: surj f ⟹ f (inv f y) = y
  · inv_inv_eq:   bij f ⟹ inv (inv f) = f

  Igualdad de funciones (por extensionalidad):
  · fun_eq_iff: (f = g) = (∀x. f x = g x)
*}

text {*
  Ejemplo de lema de demostración de propiedades de funciones: Una
  función inyectiva puede cancelarse en el lado izquierdo de la
  composición de funciones.
*}

-- "La demostración detallada es"
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

-- "La demostración estructurada es"
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: inj_on_def fun_eq_iff)
next
  assume "g = h"
  thus "f ∘ g = f ∘ h" by auto
qed

-- "La demostración automática es"
lemma
  assumes "inj f"
  shows "(f ∘ g = f ∘ h) = (g = h)"
using assms
```

```
by (auto simp add: inj_on_def fun_eq_iff)

subsubsection {* Función imagen *}

text {*
  Imagen de un conjunto mediante una función:
  · image_def: f ‘ A = {y. (∃x∈A. y = f x)

  Propiedades de la imagen:
  · image_compose: (f ∘ g)‘r = f‘g‘r
  · image_Un:      f‘(A ∪ B) = f‘A ∪ f‘B
  · image_Int:     inj f ⟹ f‘(A ∩ B) = f‘A ∩ f‘B"
*}

text {*
  Ejemplo de demostración de propiedades de la imagen:
     f‘A ∪ g‘A = (⋃x∈A. {f x, g x})
*}

lemma "f‘A ∪ g‘A = (⋃x∈A. {f x, g x})"
by auto

text {*
  Ejemplo de demostración de propiedades de la imagen:
     f‘{(x,y). P x y} = {f(x,y) | x y. P x y}
*}

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:
  · vimage_def: f -‘ B ≡ {x. f x : B}

  Propiedad de la imagen inversa de un conjunto:
  · 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:
  · Id_def: Id ≡ {p. ∃x. p = (x,x)}

  Composición de relaciones:
  · rel_comp_def: r O s ≡ {(x,z). ∃y. (x, y) ∈ r ∧ (y, z) ∈ s}

  Propiedades:
  · R_O_Id:        R O Id = R
  · rel_comp_mono: ⟦r' ⊆ r; s' ⊆ s⟧ ⟹ (r' O s') ⊆ (r O s)

  Imagen inversa de una relación:
  · converse_iff: ((a,b) ∈ r\<^bsup>\<^sup>-1\<^esup>) = ((b,a) ∈ r)

  Propiedad de la imagen inversa de una relación:
  · converse_rel_comp: (r O s)\<^bsup>-1\<^esup> = s\<^bsup>-1\<^esup> O r\<^bsup>-1\<

  Imagen de un conjunto mediante una relación:
  · Image_iff: (b ∈ r``A) = (∃x:A. (x, b) ∈ r)

  Dominio de una relación:
  · Domain_iff: (a ∈ Domain r) = (∃y. (a, y) ∈ r)

  Rango de una relación:
  · 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:
  · R ^^ 0 = Id
  · R ^^ (Suc n) = (R ^^ n) O R

  La clausura reflexiva y transitiva de la relación r es la menor
  solución de la ecuación:
  · rtrancl_unfold: r\<^sup>* = Id ∪ (r\<^sup>* O r)

  Propiedades básicas de la clausura reflexiva y transitiva:
  · rtrancl_refl:   (a,a) ∈ r\<^sup>*
  · r_into_rtrancl: p ∈ r ⟹ p ∈ r\<^sup>*
  · rtrancl_trans:  ⟦(a,b) ∈ r\<^sup>*; (b,c) ∈ r\<^sup>*⟧ ⟹ (a,c) ∈ r\<^sup>*

  Inducción sobre la clausura reflexiva y transitiva
  · 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:
  · rtrancl_idemp: (r\<^sup>* )\<^sup>* = r\<^sup>*

  Reglas de introducción de la clausura transitiva:
  · r_into_trancl': p ∈ r ⟹ p ∈ r\<^sup>+
  · trancl_trans:   ⟦(a,b) ∈ r\<^sup>+; (b,c) ∈ r\<^sup>+⟧ ⟹ (a,c) ∈ r\<^sup>+

  Ejemplo de propiedad:
  · trancl_converse: (r^-1)\<^sup>+ = (r\<^sup>+)^-1
*}

subsection {* Una demostración elemental *}

text {*
  Se desea demostrar que la clausura reflexiva y transitiva conmuta con
  la inversa (cl_rtrans_inversa). Para demostrarlo introducimos dos lemas
  auxiliares: cl_rtrans_inversaD y cl_rtrans_inversaI.
*}

-- "La demostración detallada del primer lema es"
```

```
lemma cl_rtrans_inversaD:
  "(x,y) ∈ (r^-1)\<^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) ∈ (r^-1)\<^sup>*" and "(y,z) ∈ r^-1" and "(y,x) ∈ r\<^sup>*"
  show "(z,x) ∈ r\<^sup>*"
  proof (rule rtrancl_trans)
    show "(z,y) ∈ r\<^sup>*" using `(y,z) ∈ r^-1` by simp
  next
    show "(y,x) ∈ r\<^sup>*" using `(y,x) ∈ r\<^sup>*` by simp
  qed
qed

-- "La demostración automática del primer lema es"
lemma  cl_rtrans_inversaD2:
  "(x,y) ∈ (r^-1)\<^sup>* ⟹ (y,x) ∈ r\<^sup>*"
by (induct rule: rtrancl_induct)
   (auto simp add: rtrancl_refl rtrancl_trans)

-- "La demostración detallada del segundo lema es"
lemma cl_rtrans_inversaI:
  "(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

-- "La demostración detalla del teorema es"
theorem cl_rtrans_inversa:
  "(r^-1)\<^sup>* = (r\<^sup>*)^-1"
```

```
proof
  show "(r^-1)\<^sup>* ⊆ (r\<^sup>*)^-1" by (auto simp add:cl_rtrans_inversaD)
next
  show "(r\<^sup>*)^-1 ⊆ (r^-1)\<^sup>*" by (auto simp add:cl_rtrans_inversaI)
qed

-- "La demostración automática del teorema es"
theorem "(r^-1)\<^sup>* = (r\<^sup>*)^-1"
by (auto intro: cl_rtrans_inversaI dest: cl_rtrans_inversaD)

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 definido por
  · less_than = pred_nat\<^bsup>+\<^esup>
  donde pred_nat está definida por
  · pred_nat = {(m, n). n = Suc m}

  La caracterización de less_than es
  · less_than_iff: ((x,y) ∈ less_than) = (x < y)

  La relación less_than está bien fundamentada
  · wf_less_than:  wf less_than

  Notas sobre medidas:
  · Imagen inversa de una relación mediante una función:
    · inv_image_def: inv_image r f ≡ {(x,y). (f x,f y) ∈ r}
  · Conservación de la buena fundamentación:
    · wf_inv_image: wf r ⟹ wf (inv_image r f)
  · Definición de la medida:
    · measure_def: measure ≡ inv_image less_than
  · Buena fundamentación de la medida:
    · wf_measure: wf (measure f)
*}

text {*
  Notas sobre el producto lexicográfico:
```

```
  · Definición del producto lexicográfico (lex_prod_def):
    ra <*lex*> rb ≡ {((a,b),(a',b')). (a,a') ∈ ra ∨
                                      (a = a' ∧ (b,b') ∈ rb)}
  · Conservación de la buena fundamentación:
    · 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:
  · wf_induct: ⟦wf r; ⋀x. (⋀y. (y,x) ∈ r ⟹ P y) ⟹ P x⟧ ⟹ P a
*}

section {* Puntos fijos *}

text {*
  La teoría de los puntos fijos se aplican a las funciones monótonas.

  Las funciones monótonas está definida (en Orderings.thy) por
  · mono_def: mono f ≡ ∀A B. A ≤ B ⟶ f A ≤ f B

  Las reglas de introducción y eliminación de la monotonicidad son:
  . monoI: (⋀A B. A ≤ B ⟹ f A ≤ f B) ⟹ mono f
  · monoD: ⟦mono f ⟹ A ≤ B⟧ ⟹ f A ≤ f B

  El menor punto fijo de un operador está definido en la teoría
  Inductive.thy, para los retículos completos, por
  · lfp_def: lfp f = Inf {u. f u ≤ u}

  El menor punto fijo de una función monótona es un punto fijo:
  · lfp_unfold: mono f ⟹ lfp f = f (lfp f)

  La regla de inducción del menor punto fijo es
  · lfp_induct_set: ⟦ a ∈ lfp(f);
                      mono(f);
                      ⋀x. ⟦ x ∈ f(lfp(f) ⋂ {x. P(x)}) ⟧ ⟹ P(x) ⟧
                    ⟹ P(a)
*}

text {*
  El mayor punto fijo de un operador está definido en la teoría
```

```
  Inductive.thy, para los retículos completos, por
  · gfp_def: gfp f = Sup {u. u ≤ f u}

  El menor punto fijo de una función monótona es un punto fijo:
  · gfp_unfold: mono f ⟹ gfp f = f (gfp f)

  La regla de inducción del menor punto fijo es
  · coinduct_set: ⟦ mono(f);
                    a ∈ X;
                    X ⊆ f(X ⋃ gfp(f)) ⟧
                  ⟹ a ∈ gfp(f)
*}

end
```

Parte II

Ejercicios

Relación 1

Deducción natural en lógica proposicional

```
header {* R1: Deducción natural proposicional *}

theory R1
imports Main
begin

text {*
  ----------------------------------------------------------------------
  El objetivo de esta relación es demostrar cada uno de los ejercicios
  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:
  · conjI:      ⟦P; Q⟧ ⟹ P ∧ Q
  · conjunct1:  P ∧ Q ⟹ P
  · conjunct2:  P ∧ Q ⟹ Q
  · notnotD:    ¬¬ P ⟹ P
  · notnotI:    P ⟹ ¬¬ P
  · mp:         ⟦P ⟶ Q; P⟧ ⟹ Q
  · mt:         ⟦F ⟶ G; ¬G⟧ ⟹ ¬F
  · impI:       (P ⟹ Q) ⟹ P ⟶ Q
  · disjI1:     P ⟹ P ∨ Q
  · disjI2:     Q ⟹ P ∨ Q
  · disjE:      ⟦P ∨ Q; P ⟹ R; Q ⟹ R⟧ ⟹ R
  · FalseE:     False ⟹ P
```

```
  · notE:       ⟦¬P; P⟧ ⟹ R
  · notI:       (P ⟹ False) ⟹ ¬P
  · iffI:       ⟦P ⟹ Q; Q ⟹ P⟧ ⟹ P = Q
  · iffD1:      ⟦Q = P; Q⟧ ⟹ P
  · iffD2:      ⟦P = Q; Q⟧ ⟹ P
  · ccontr:     (¬P ⟹ False) ⟹ P
  ------------------------------------------------------------------------
*}

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

lemma notnotI: "P ⟹ ¬¬ P"
by auto

lemma mt: "⟦F ⟶ G; ¬G⟧ ⟹ ¬F"
by auto

section {* Implicaciones *}

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

lemma ejercicio_1:
  assumes "p ⟶ q"
          "p"
  shows "q"
oops

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

lemma ejercicio_2:
  assumes "p ⟶ q"
          "q ⟶ r"
          "p"
```

```
  shows "r"
oops

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

lemma ejercicio_3:
  assumes "p ⟶ (q ⟶ r)"
          "p ⟶ q"
          "p"
  shows "r"
oops

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

lemma ejercicio_4:
  assumes "p ⟶ q"
          "q ⟶ r"
  shows "p ⟶ r"
oops

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

lemma ejercicio_5:
  assumes "p ⟶ (q ⟶ r)"
  shows   "q ⟶ (p ⟶ r)"
oops

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

```
lemma ejercicio_6:
  assumes "p ⟶ (q ⟶ r)"
  shows   "(p ⟶ q) ⟶ (p ⟶ r)"
oops

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

lemma ejercicio_7:
  assumes "p"
  shows   "q ⟶ p"
oops

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

lemma ejercicio_8:
  "p ⟶ (q ⟶ p)"
oops

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

lemma ejercicio_9:
  assumes "p ⟶ q"
  shows   "(q ⟶ r) ⟶ (p ⟶ r)"
oops

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

```
lemma ejercicio_10:
  assumes "p ⟶ (q ⟶ (r ⟶ s))"
  shows   "r ⟶ (q ⟶ (p ⟶ s))"
oops

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

lemma ejercicio_11:
  "(p ⟶ (q ⟶ r)) ⟶ ((p ⟶ q) ⟶ (p ⟶ r))"
oops

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

lemma ejercicio_12:
  assumes "(p ⟶ q) ⟶ r"
  shows   "p ⟶ (q ⟶ r)"
oops

section {* Conjunciones *}

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

lemma ejercicio_13:
  assumes "p"
          "q"
  shows "p ∧ q"
oops

text {* ---------------------------------------------------------------
  Ejercicio 14. Demostrar
     p ∧ q ⊢ p
```

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

lemma ejercicio_14:
  assumes "p ∧ q"
  shows   "p"
oops

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

lemma ejercicio_15:
  assumes "p ∧ q"
  shows   "q"
oops

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

lemma ejercicio_16:
  assumes "p ∧ (q ∧ r)"
  shows   "(p ∧ q) ∧ r"
oops

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

lemma ejercicio_17:
  assumes "(p ∧ q) ∧ r"
  shows   "p ∧ (q ∧ r)"
oops

text {* ----------------------------------------------------------------
  Ejercicio 18. Demostrar
     p ∧ q ⊢ p ⟶ q
```

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

lemma ejercicio_18:
  assumes "p ∧ q"
  shows   "p ⟶ q"
oops

text {* ------------------------------------------------------------
  Ejercicio 19. Demostrar
     (p ⟶ q) ∧ (p ⟶ r) ⊢ p ⟶ q ∧ r
  ------------------------------------------------------------ *}

lemma ejercicio_19:
  assumes "(p ⟶ q) ∧ (p ⟶ r)"
  shows   "p ⟶ q ∧ r"
oops

text {* ------------------------------------------------------------
  Ejercicio 20. Demostrar
     p ⟶ q ∧ r ⊢ (p ⟶ q) ∧ (p ⟶ r)
  ------------------------------------------------------------ *}

lemma ejercicio_20:
  assumes "p ⟶ q ∧ r"
  shows   "(p ⟶ q) ∧ (p ⟶ r)"
oops

text {* ------------------------------------------------------------
  Ejercicio 21. Demostrar
     p ⟶ (q ⟶ r) ⊢ p ∧ q ⟶ r
  ------------------------------------------------------------ *}

lemma ejercicio_21:
  assumes "p ⟶ (q ⟶ r)"
  shows   "p ∧ q ⟶ r"
oops

text {* ------------------------------------------------------------
  Ejercicio 22. Demostrar
     p ∧ q ⟶ r ⊢ p ⟶ (q ⟶ r)
```

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

lemma ejercicio_22:
  assumes "p ∧ q ⟶ r"
  shows   "p ⟶ (q ⟶ r)"
oops

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

lemma ejercicio_23:
  assumes "(p ⟶ q) ⟶ r"
  shows   "p ∧ q ⟶ r"
oops

text {* ----------------------------------------------------------------
  Ejercicio 24. Demostrar
     p ∧ (q ⟶ r) ⊢ (p ⟶ q) ⟶ r
  ---------------------------------------------------------------- *}

lemma ejercicio_24:
  assumes "p ∧ (q ⟶ r)"
  shows   "(p ⟶ q) ⟶ r"
oops

section {* Disyunciones *}

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

lemma ejercicio_25:
  assumes "p"
  shows   "p ∨ q"
oops

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

```
  Ejercicio 26. Demostrar
     q ⊢ p ∨ q
  ------------------------------------------------------------ *}

lemma ejercicio_26:
  assumes "q"
  shows   "p ∨ q"
oops

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

lemma ejercicio_27:
  assumes "p ∨ q"
  shows   "q ∨ p"
oops

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

lemma ejercicio_28:
  assumes "q ⟶ r"
  shows   "p ∨ q ⟶ p ∨ r"
oops

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

lemma ejercicio_29:
  assumes "p ∨ p"
  shows   "p"
oops

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

```
  Ejercicio 30. Demostrar
     p ⊢ p ∨ p
  ------------------------------------------------------------ *}

lemma ejercicio_30:
  assumes "p"
  shows   "p ∨ p"
oops

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

lemma ejercicio_31:
  assumes "p ∨ (q ∨ r)"
  shows   "(p ∨ q) ∨ r"
oops

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

lemma ejercicio_32:
  assumes "(p ∨ q) ∨ r"
  shows   "p ∨ (q ∨ r)"
oops

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

lemma ejercicio_33:
  assumes "p ∧ (q ∨ r)"
  shows   "(p ∧ q) ∨ (p ∧ r)"
oops

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

```
  Ejercicio 34. Demostrar
     (p ∧ q) ∨ (p ∧ r) ⊢ p ∧ (q ∨ r)
  ------------------------------------------------------------------ *}

lemma ejercicio_34:
  assumes "(p ∧ q) ∨ (p ∧ r)"
  shows   "p ∧ (q ∨ r)"
oops

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

lemma ejercicio_35:
  assumes "p ∨ (q ∧ r)"
  shows   "(p ∨ q) ∧ (p ∨ r)"
oops

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

lemma ejercicio_36:
  assumes "(p ∨ q) ∧ (p ∨ r)"
  shows   "p ∨ (q ∧ r)"
oops

text {* ------------------------------------------------------------------
  Ejercicio 37. Demostrar
     (p ⟶ r) ∧ (q ⟶ r) ⊢ p ∨ q ⟶ r
  ------------------------------------------------------------------ *}

lemma ejercicio_37:
  assumes "(p ⟶ r) ∧ (q ⟶ r)"
  shows   "p ∨ q ⟶ r"
oops

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

```
  Ejercicio 38. Demostrar
     p ∨ q ⟶ r ⊢ (p ⟶ r) ∧ (q ⟶ r)
  ---------------------------------------------------------------- *}

lemma ejercicio_38:
  assumes "p ∨ q ⟶ r"
  shows   "(p ⟶ r) ∧ (q ⟶ r)"
oops

section {* Negaciones *}

text {* ----------------------------------------------------------------
  Ejercicio 39. Demostrar
     p ⊢ ¬¬p
  ---------------------------------------------------------------- *}

lemma ejercicio_39:
  assumes "p"
  shows   "¬¬p"
oops

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

lemma ejercicio_40:
  assumes "¬p"
  shows   "p ⟶ q"
oops

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

lemma ejercicio_41:
  assumes "p ⟶ q"
  shows   "¬q ⟶ ¬p"
oops
```

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

lemma ejercicio_42:
  assumes "p∨q"
          "¬q"
  shows   "p"
oops

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

lemma ejercicio_43:
  assumes "p ∨ q"
          "¬p"
  shows   "q"
oops

text {* ---------------------------------------------------------------
  Ejercicio 40. Demostrar
     p ∨ q ⊢ ¬(¬p ∧ ¬q)
  ---------------------------------------------------------------- *}

lemma ejercicio_44:
  assumes "p ∨ q"
  shows   "¬(¬p ∧ ¬q)"
oops

text {* ---------------------------------------------------------------
  Ejercicio 45. Demostrar
     p ∧ q ⊢ ¬(¬p ∨ ¬q)
  ---------------------------------------------------------------- *}

lemma ejercicio_45:
  assumes "p ∧ q"
```

```
  shows   "¬(¬p ∨ ¬q)"
oops

text {* ---------------------------------------------------------------
  Ejercicio 46. Demostrar
     ¬(p ∨ q) ⊢ ¬p ∧ ¬q
  ---------------------------------------------------------------- *}

lemma ejercicio_46:
  assumes "¬(p ∨ q)"
  shows   "¬p ∧ ¬q"
oops

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

lemma ejercicio_47:
  assumes "¬p ∧ ¬q"
  shows   "¬(p ∨ q)"
oops

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

lemma ejercicio_48:
  assumes "¬p ∨ ¬q"
  shows   "¬(p ∧ q)"
oops

text {* ---------------------------------------------------------------
  Ejercicio 49. Demostrar
     ⊢ ¬(p ∧ ¬p)
  ---------------------------------------------------------------- *}

lemma ejercicio_49:
  "¬(p ∧ ¬p)"
```

```
oops

text {* ---------------------------------------------------------------
  Ejercicio 50. Demostrar
     p ∧ ¬p ⊢ q
  ---------------------------------------------------------------- *}

lemma ejercicio_50:
  assumes "p ∧ ¬p"
  shows   "q"
oops

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

lemma ejercicio_51:
  assumes "¬¬p"
  shows   "p"
oops

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

lemma ejercicio_52:
  "p ∨ ¬p"
oops

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

lemma ejercicio_53:
  "((p ⟶ q) ⟶ p) ⟶ p"
oops
```

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

lemma ejercicio_54:
  assumes "¬q ⟶ ¬p"
  shows   "p ⟶ q"
oops

text {* ---------------------------------------------------------------
  Ejercicio 55. Demostrar
     ¬(¬p ∧ ¬q) ⊢ p ∨ q
  ---------------------------------------------------------------- *}

lemma ejercicio_55:
  assumes "¬(¬p ∧ ¬q)"
  shows   "p ∨ q"
oops

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

lemma ejercicio_56:
  assumes "¬(¬p ∨ ¬q)"
  shows   "p ∧ q"
oops

text {* ---------------------------------------------------------------
  Ejercicio 57. Demostrar
     ¬(p ∧ q) ⊢ ¬p ∨ ¬q
  ---------------------------------------------------------------- *}

lemma ejercicio_57:
  assumes "¬(p ∧ q)"
  shows   "¬p ∨ ¬q"
oops
```

```
text {* ------------------------------------------------------------
  Ejercicio 58. Demostrar
     ⊢ (p ⟶ q) ∨ (q ⟶ p)
  ------------------------------------------------------------ *}

lemma ejercicio_58:
  "(p ⟶ q) ∨ (q ⟶ p)"
oops

end
```

Relación 2

Argumentación lógica proposicional

```
header {* R2: Argumentación proposicional *}

theory R2
imports Main
begin

text {*
  ----------------------------------------------------------------------
  El objetivo de esta 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:
  · conjI:      ⟦P; Q⟧ ⟹ P ∧ Q
  · conjunct1:  P ∧ Q ⟹ P
  · conjunct2:  P ∧ Q ⟹ Q
  · notnotD:    ¬¬ P ⟹ P
  · notnotI:    P ⟹ ¬¬ P
  · mp:         ⟦P ⟶ Q; P⟧ ⟹ Q
  · mt:         ⟦F ⟶ G; ¬G⟧ ⟹ ¬F
  · impI:       (P ⟹ Q) ⟹ P ⟶ Q
  · disjI1:     P ⟹ P ∨ Q
  · disjI2:     Q ⟹ P ∨ Q
  · disjE:      ⟦P ∨ Q; P ⟹ R; Q ⟹ R⟧ ⟹ R
  · FalseE:     False ⟹ P
  · notE:       ⟦¬P; P⟧ ⟹ R
  · notI:       (P ⟹ False) ⟹ ¬P
```

```
  · iffI:       ⟦P ⟹ Q; Q ⟹ P⟧ ⟹ P = Q
  · iffD1:      ⟦Q = P; Q⟧ ⟹ P
  · iffD2:      ⟦P = Q; Q⟧ ⟹ P
  · ccontr:     (¬P ⟹ False) ⟹ P
  ------------------------------------------------------------------
*}

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

lemma notnotI: "P ⟹ ¬¬ P"
by auto

lemma mt: "⟦F ⟶ G; ¬G⟧ ⟹ ¬F"
by auto

text {* ------------------------------------------------------------
  Ejercicio 1. 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".
  ------------------------------------------------------------ *}

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".
  ------------------------------------------------------------ *}

text {* ------------------------------------------------------------
  Ejercicio 3. 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.
  ------------------------------------------------------------- *}

text {* -------------------------------------------------------------
  Ejercicio 4. 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.
  ------------------------------------------------------------- *}

text {* -------------------------------------------------------------
  Ejercicio 5. 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.
  ------------------------------------------------------------- *}

text {* -------------------------------------------------------------
  Ejercicio 6. 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.
  ------------------------------------------------------------- *}
```

```
text {* ------------------------------------------------------------
  Ejercicio 7. 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.
  ------------------------------------------------------------ *}

text {* ------------------------------------------------------------
  Ejercicio 8. 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.
       O: El general obedece las órdenes.
       I: El general es inteligente.
       C: El general comprende las órdenes.
  ------------------------------------------------------------ *}

text {* ------------------------------------------------------------
  Ejercicio 9. 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.
       O: Dios es omnipotente.
       M: Dios es malévolo.
       P: Dios evita el mal.
       E: Dios existe.
```

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

text {* ------------------------------------------------------------
  Ejercicio 10. Formalizar, y demostrar la corrección, del siguiente
  argumento
     Nadie más que Pedro, Quintín y Raúl están bajo sospecha y al menos
     uno es traidor. Pedro nunca trabaja sin llevar al menos un cómplice
     (que puede ser Quintín o Raúl). Raúl es leal. Por lo tanto,
     Pedro es traidor.
  Usar p: Pedro es traidor.
       q : Quintín es traidor.
       r : Raúl es traidor.
  ------------------------------------------------------------ *}

text {* ------------------------------------------------------------
  Ejercicio 11. 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.
  ------------------------------------------------------------ *}

text {* ------------------------------------------------------------
  Ejercicio 12. 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.
  ------------------------------------------------------------------ *}

end
```

Relación 3

Eliminación de conectivas

```
header {* R3: Eliminación de conectivas *}

theory R3
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.
  ------------------------------------------------------------ *}

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

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

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

end
```

Relación 4

Deducción natural en lógica de primer orden

```
header {* R4: Deducción natural de primer orden *}

theory R4
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:
  · conjI:      ⟦P; Q⟧ ⟹ P ∧ Q
  · conjunct1:  P ∧ Q ⟹ P
  · conjunct2:  P ∧ Q ⟹ Q
  · notnotD:    ¬¬ P ⟹ P
  · mp:         ⟦P ⟶ Q; P⟧ ⟹ Q
  · impI:       (P ⟹ Q) ⟹ P ⟶ Q
  · disjI1:     P ⟹ P ∨ Q
  · disjI2:     Q ⟹ P ∨ Q
  · disjE:      ⟦P ∨ Q; P ⟹ R; Q ⟹ R⟧ ⟹ R
  · FalseE:     False ⟹ P
  · notE:       ⟦¬P; P⟧ ⟹ R
  · notI:       (P ⟹ False) ⟹ ¬P
  · iffI:       ⟦P ⟹ Q; Q ⟹ P⟧ ⟹ P = Q
  · iffD1:      ⟦Q = P; Q⟧ ⟹ P
  · iffD2:      ⟦P = Q; Q⟧ ⟹ P
```

```
  · ccontr:      (¬P ⟹ False) ⟹ P

  · allI:        ⟦∀x. P x; P x ⟹ R⟧ ⟹ R
  · allE:        (⋀x. P x) ⟹ ∀x. P x
  · exI:         P x ⟹ ∃x. P x
  · exE:         ⟦∃x. P x; ⋀x. P x ⟹ Q⟧ ⟹ Q

  · refl:        t = t
  · subst:       ⟦s = t; P s⟧ ⟹ P t
  · trans:       ⟦r = s; s = t⟧ ⟹ r = t
  · sym:         s = t ⟹ t = s
  · not_sym:     t ≠ s ⟹ s ≠ t
  · ssubst:      ⟦t = s; P s⟧ ⟹ P t
  · box_equals: ⟦a = b; a = c; b = d⟧ ⟹ a: = d
  · arg_cong:    x = y ⟹ f x = f y
  · fun_cong:    f = g ⟹ f x = g x
  · cong:        ⟦f = g; x = y⟧ ⟹ f x = g y
*}

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

lemma notnotI: "P ⟹ ¬¬ P"
by auto

lemma mt: "⟦F ⟶ G; ¬G⟧ ⟹ ¬F"
by auto

text {* ------------------------------------------------------------
  Ejercicio 1. Demostrar
       ∀x. P x ⟶ Q x ⊢ (∀x. P x) ⟶ (∀x. Q x)
  ------------------------------------------------------------ *}

lemma ejercicio_1:
  assumes "∀x. P x ⟶ Q x"
  shows   "(∀x. P x) ⟶ (∀x. Q x)"
oops

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

```
  Ejercicio 2. Demostrar
       ∃x. ¬(P x) ⊢ ¬(∀x. P x)
  ------------------------------------------------------------------ *}

lemma ejercicio_2:
  assumes "∃x. ¬(P x)"
  shows   "¬(∀x. P x)"
oops

text {* ------------------------------------------------------------------
  Ejercicio 3. Demostrar
       ∀x. P x ⊢ ∀y. P y
  ------------------------------------------------------------------ *}

lemma ejercicio_3:
  assumes "∀x. P x"
  shows   "∀y. P y"
oops

text {* ------------------------------------------------------------------
  Ejercicio 4. Demostrar
       ∀x. P x ⟶ Q x ⊢ (∀x. ¬(Q x)) ⟶ (∀x. ¬ (P x))
  ------------------------------------------------------------------ *}

lemma ejercicio_4:
  assumes "∀x. P x ⟶ Q x"
  shows   "(∀x. ¬(Q x)) ⟶ (∀x. ¬ (P x))"
oops

text {* ------------------------------------------------------------------
  Ejercicio 5. Demostrar
       ∀x. P x  ⟶ ¬(Q x) ⊢ ¬(∃x. P x ∧ Q x)
  ------------------------------------------------------------------ *}

lemma ejercicio_5:
  assumes "∀x. P x  ⟶ ¬(Q x)"
  shows   "¬(∃x. P x ∧ Q x)"
oops

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

```
  Ejercicio 6. Demostrar
       ∀x y. P x y ⊢ ∀u v. P u v
  ---------------------------------------------------------------- *}

lemma ejercicio_6:
  assumes "∀x y. P x y"
  shows   "∀u v. P u v"
oops

text {* ----------------------------------------------------------------
  Ejercicio 7. Demostrar
       ∃x y. P x y ⟹ ∃u v. P u v
  ---------------------------------------------------------------- *}

lemma ejercicio_7:
  assumes "∃x y. P x y"
  shows   "∃u v. P u v"
oops

text {* ----------------------------------------------------------------
  Ejercicio 8. Demostrar
       ∃x. ∀y. P x y ⊢ ∀y. ∃x. P x y
  ---------------------------------------------------------------- *}

lemma ejercicio_8:
  assumes "∃x. ∀y. P x y"
  shows   "∀y. ∃x. P x y"
oops

text {* ----------------------------------------------------------------
  Ejercicio 9. Demostrar
       ∃x. P a ⟶ Q x ⊢ P a ⟶ (∃x. Q x)
  ---------------------------------------------------------------- *}

lemma ejercicio_9:
  assumes "∃x. P a ⟶ Q x"
  shows   "P a ⟶ (∃x. Q x)"
oops

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

```
  Ejercicio 10. Demostrar
       P a ⟶ (∃x. Q x) ⊢ ∃x. P a ⟶ Q x
  ------------------------------------------------------------------ *}

lemma ejercicio_10:
  fixes P Q :: "'b ⇒ bool"
  assumes "P a ⟶ (∃x. Q x)"
  shows   "∃x. P a ⟶ Q x"
oops

text {* ------------------------------------------------------------------
  Ejercicio 11. Demostrar
       (∃x. P x) ⟶ Q a ⊢ ∀x. P x ⟶ Q a
  ------------------------------------------------------------------ *}

lemma ejercicio_11:
  assumes "(∃x. P x) ⟶ Q a"
  shows   "∀x. P x ⟶ Q a"
oops

text {* ------------------------------------------------------------------
  Ejercicio 12. Demostrar
       ∀x. P x ⟶ Q a ⊢ ∃ x. P x ⟶ Q a
  ------------------------------------------------------------------ *}

lemma ejercicio_12:
  assumes "∀x. P x ⟶ Q a"
  shows   "∃x. P x ⟶ Q a"
oops

text {* ------------------------------------------------------------------
  Ejercicio 13. Demostrar
       (∀x. P x) ∨ (∀x. Q x) ⊢ ∀x. P x ∨ Q x
  ------------------------------------------------------------------ *}

lemma ejercicio_13:
  assumes "(∀x. P x) ∨ (∀x. Q x)"
  shows   "∀x. P x ∨ Q x"
oops
```

```
text {* ----------------------------------------------------------
  Ejercicio 14. Demostrar
       ∃x. P x ∧ Q x ⊢ (∃x. P x) ∧ (∃x. Q x)
  ---------------------------------------------------------- *}

lemma ejercicio_14:
  assumes "∃x. P x ∧ Q x"
  shows   "(∃x. P x) ∧ (∃x. Q x)"
oops

text {* ----------------------------------------------------------
  Ejercicio 15. Demostrar
       ∀x y. P y ⟶ Q x ⊢ (∃y. P y) ⟶ (∀x. Q x)
  ---------------------------------------------------------- *}

lemma ejercicio_15:
  assumes "∀x y. P y ⟶ Q x"
  shows   "(∃y. P y) ⟶ (∀x. Q x)"
oops

text {* ----------------------------------------------------------
  Ejercicio 16. Demostrar
       ¬(∀x. ¬(P x)) ⊢ ∃x. P x
  ---------------------------------------------------------- *}

lemma ejercicio_16:
  assumes "¬(∀x. ¬(P x))"
  shows   "∃x. P x"
oops

text {* ----------------------------------------------------------
  Ejercicio 17. Demostrar
       ∀x. ¬(P x) ⊢ ¬(∃x. P x)
  ---------------------------------------------------------- *}

lemma ejercicio_17:
  assumes "∀x. ¬(P x)"
  shows   "¬(∃x. P x)"
oops
```

```
text {* ---------------------------------------------------------------
  Ejercicio 18. Demostrar
       ∃x. P x ⊢ ¬(∀x. ¬(P x))
  ---------------------------------------------------------------- *}

lemma ejercicio_18:
  assumes "∃x. P x"
  shows   "¬(∀x. ¬(P x))"
oops

text {* ---------------------------------------------------------------
  Ejercicio 19. Demostrar
       P a ⟶ (∀x. Q x) ⊢ ∀x. P a ⟶ Q x
  ---------------------------------------------------------------- *}

lemma ejercicio_19:
  assumes "P a ⟶ (∀x. Q x)"
  shows   "∀x. P a ⟶ Q x"
oops

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)
  ---------------------------------------------------------------- *}

lemma ejercicio_20:
  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)"
oops

text {* ---------------------------------------------------------------
  Ejercicio 21. Demostrar
     {∀x. P x ∨ Q x, ∃x. ¬(Q x), ∀x. R x ⟶ ¬(P x)} ⊢ ∃x. ¬(R x)
  ---------------------------------------------------------------- *}

lemma ejercicio_21:
  assumes "∀x. P x ∨ Q x"
```

```
          "∃x. ¬(Q x)"
          "∀x. R x ⟶ ¬(P x)"
  shows   "∃x. ¬(R x)"
oops

text {* ---------------------------------------------------------------
  Ejercicio 22. Demostrar
     {∀x. P x ⟶ Q x ∨ R x, ¬(∃x. P x ∧ R x)} ⊢ ∀x. P x ⟶ Q x
  ---------------------------------------------------------------- *}

lemma ejercicio_22:
  assumes "∀x. P x ⟶ Q x ∨ R x"
          "¬(∃x. P x ∧ R x)"
  shows   "∀x. P x ⟶ Q x"
oops

text {* ---------------------------------------------------------------
  Ejercicio 23. Demostrar
     ∃x y. R x y ∨ R y x ⊢ ∃x y. R x y
  ---------------------------------------------------------------- *}

lemma ejercicio_23:
  assumes "∃x y. R x y ∨ R y x"
  shows   "∃x y. R x y"
oops

text {* ---------------------------------------------------------------
  Ejercicio 24. Demostrar
       (∃x. ∀y. P x y) ⟶ (∀y. ∃x. P x y)
  ---------------------------------------------------------------- *}

lemma ejercicio_24:
  "(∃x. ∀y. P x y) ⟶ (∀y. ∃x. P x y)"
oops

text {* ---------------------------------------------------------------
  Ejercicio 25. Demostrar
       (∀x. P x ⟶ Q) ⟷ ((∃x. P x) ⟶ Q)
  ---------------------------------------------------------------- *}
```

```
lemma ejercicio_25:
  "(∀x. P x ⟶ Q) ⟷ ((∃x. P x) ⟶ Q)"
oops

text {* ---------------------------------------------------------------
  Ejercicio 26. Demostrar
       ((∀x. P x) ∧ (∀x. Q x)) ⟷ (∀x. P x ∧ Q x)
  ---------------------------------------------------------------- *}

lemma ejercicio_26:
  "((∀x. P x) ∧ (∀x. Q x)) ⟷ (∀x. P x ∧ Q x)"
oops

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

text {* ---------------------------------------------------------------
  Ejercicio 28. Demostrar o refutar
       ((∃x. P x) ∨ (∃x. Q x)) ⟷ (∃x. P x ∨ Q x)
  ---------------------------------------------------------------- *}

lemma ejercicio_28:
  "((∃x. P x) ∨ (∃x. Q x)) ⟷ (∃x. P x ∨ Q x)"
oops

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)"
oops
```

```
text {* ---------------------------------------------------------------
  Ejercicio 30. Demostrar o refutar
       (¬(∀x. P x)) ⟷ (∃x. ¬P x)
  ---------------------------------------------------------------- *}

lemma ejercicio_30:
  "(¬(∀x. P x)) ⟷ (∃x. ¬P x)"
oops

section {* Ejercicios sobre igualdad *}

text {* ---------------------------------------------------------------
  Ejercicio 31. Demostrar o refutar
       P a ⟹ ∀x. x = a ⟶ P x
  ---------------------------------------------------------------- *}

lemma ejercicio_31b:
  assumes "P a"
  shows   "∀x. x = a ⟶ P x"
oops

text {* ---------------------------------------------------------------
  Ejercicio 32. Demostrar o refutar
       ∃x y. R x y ∨ R y x; ¬(∃x. R x x)⟧ ⟹ ∃x y. x ≠ y
  ---------------------------------------------------------------- *}

lemma ejercicio_32:
  fixes R :: "'c ⇒ 'c ⇒ bool"
  assumes "∃x y. R x y ∨ R y x"
          "¬(∃x. R x x)"
  shows   "∃(x::'c) y. x ≠ y"
oops

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

```
lemma ejercicio_33:
  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)"
oops

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))
  ---------------------------------------------------------------- *}

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))"
oops

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))
  ---------------------------------------------------------------- *}

lemma ejercicio_35:
  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))"
oops

text {* ---------------------------------------------------------------
  Ejercicio 36. Demostrar o refutar
     {x = f x, odd (f x)} ⊢ odd x
  ---------------------------------------------------------------- *}

lemma ejercicio_36b:
  "⟦x = f x; odd (f x)⟧ ⟹ odd x"
oops
```

```
text {* ---------------------------------------------------------------
  Ejercicio 37. Demostrar o refutar
     {x = f x, triple (f x) (f x) x} ⊢ triple x x x
  ---------------------------------------------------------------- *}

lemma ejercicio_37b:
  "⟦x = f x; triple (f x) (f x) x⟧ ⟹ triple x x x"
oops

end
```

Relación 5

Argumentación lógica de primer orden

```
header {* R5: Argumentación en lógica de primer orden *}

theory R5
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:
  · conjI:      ⟦P; Q⟧ ⟹ P ∧ Q
  · conjunct1:  P ∧ Q ⟹ P
  · conjunct2:  P ∧ Q ⟹ Q
  · notnotD:    ¬¬ P ⟹ P
  · notnotI:    P ⟹ ¬¬ P
  · mp:         ⟦P ⟶ Q; P⟧ ⟹ Q
  · mt:         ⟦F ⟶ G; ¬G⟧ ⟹ ¬F
  · impI:       (P ⟹ Q) ⟹ P ⟶ Q
  · disjI1:     P ⟹ P ∨ Q
  · disjI2:     Q ⟹ P ∨ Q
  · disjE:      ⟦P ∨ Q; P ⟹ R; Q ⟹ R⟧ ⟹ R
  · FalseE:     False ⟹ P
```

```
  · notE:        ⟦¬P; P⟧ ⟹ R
  · notI:        (P ⟹ False) ⟹ ¬P
  · iffI:        ⟦P ⟹ Q; Q ⟹ P⟧ ⟹ P = Q
  · iffD1:       ⟦Q = P; Q⟧ ⟹ P
  · iffD2:       ⟦P = Q; Q⟧ ⟹ P
  · ccontr:      (¬P ⟹ False) ⟹ P
  · excluded_middle: ¬P ∨ P

  · allI:        ⟦∀x. P x; P x ⟹ R⟧ ⟹ R
  · allE:        (⋀x. P x) ⟹ ∀x. P x
  · exI:         P x ⟹ ∃x. P x
  · exE:         ⟦∃x. P x; ⋀x. P x ⟹ Q⟧ ⟹ Q

  ------------------------------------------------------------------------
*}

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

lemma notnotI: "P ⟹ ¬¬ P"
by auto

lemma mt: "⟦F ⟶ G; ¬G⟧ ⟹ ¬F"
by auto

lemma no_ex: "¬(∃x. P(x)) ⟹ ∀x. ¬P(x)"
by auto

lemma no_para_todo: "¬(∀x. P(x)) ⟹ ∃x. ¬P(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
  ---------------------------------------------------------------- *}

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
  ---------------------------------------------------------------- *}

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
  ---------------------------------------------------------------- *}

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
  ---------------------------------------------------------------- *}

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
  ---------------------------------------------------------------- *}

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
  ---------------------------------------------------------------- *}

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
  ---------------------------------------------------------------- *}

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
  ------------------------------------------------------------------ *}

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
  ------------------------------------------------------------------ *}

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
  ------------------------------------------------------------------ *}

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
  ------------------------------------------------------------------ *}

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
  ------------------------------------------------------------------ *}

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
  ------------------------------------------------------------------ *}

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
  ------------------------------------------------------------- *}

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
   ------------------------------------------------------------- *}

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
   ------------------------------------------------------------------ *}

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
   ------------------------------------------------------------------ *}

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
   ------------------------------------------------------------------ *}

text {* ------------------------------------------------------------------
  Ejercicio 19. [Problema del apisonador de Schubert (en inglés,
  "Schubert's 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
   ------------------------------------------------------------------ *}

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
   ------------------------------------------------------------------ *}

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
   ------------------------------------------------------------------- *}

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
   ------------------------------------------------------------------- *}

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
   ------------------------------------------------------------------- *}

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
  ------------------------------------------------------------------ *}

end
```

Relación 6

Argumentación lógica de primer orden con igualdad

```
header {* R6: Argumentación en lógica de primer orden con igualdad *}

theory R6
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:
  · conjI:      ⟦P; Q⟧ ⟹ P ∧ Q
  · conjunct1:  P ∧ Q ⟹ P
  · conjunct2:  P ∧ Q ⟹ Q
  · notnotD:    ¬¬ P ⟹ P
  · notnotI:    P ⟹ ¬¬ P
  · mp:         ⟦P ⟶ Q; P⟧ ⟹ Q
  · mt:         ⟦F ⟶ G; ¬G⟧ ⟹ ¬F
  · impI:       (P ⟹ Q) ⟹ P ⟶ Q
  · disjI1:     P ⟹ P ∨ Q
  · disjI2:     Q ⟹ P ∨ Q
```

```
  · disjE:       ⟦P ∨ Q; P ⟹ R; Q ⟹ R⟧ ⟹ R
  · FalseE:      False ⟹ P
  · notE:        ⟦¬P; P⟧ ⟹ R
  · notI:        (P ⟹ False) ⟹ ¬P
  · iffI:        ⟦P ⟹ Q; Q ⟹ P⟧ ⟹ P = Q
  · iffD1:       ⟦Q = P; Q⟧ ⟹ P
  · iffD2:       ⟦P = Q; Q⟧ ⟹ P
  · ccontr:      (¬P ⟹ False) ⟹ P
  · excluded_middle: ¬P ∨ P

  · allI:        ⟦∀x. P x; P x ⟹ R⟧ ⟹ R
  · allE:        (⋀x. P x) ⟹ ∀x. P x
  · exI:         P x ⟹ ∃x. P x
  · exE:         ⟦∃x. P x; ⋀x. P x ⟹ Q⟧ ⟹ Q

  · refl:        t = t
  · subst:       ⟦s = t; P s⟧ ⟹ P t

  · trans:       ⟦r = s; s = t⟧ ⟹ r = t
  · sym:         s = t ⟹ t = s
  · not_sym:     t ≠ s ⟹ s ≠ t
  · ssubst:      ⟦t = s; P s⟧ ⟹ P t
  · box_equals:  ⟦a = b; a = c; b = d⟧ ⟹ a: = d
  · arg_cong:    x = y ⟹ f x = f y
  · fun_cong:    f = g ⟹ f x = g x
  · 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
  ------------------------------------------------------------ *}

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
  ------------------------------------------------------------ *}

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
  ------------------------------------------------------------ *}

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
  ------------------------------------------------------------------ *}

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
  ------------------------------------------------------------------ *}

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
  ------------------------------------------------------------------ *}

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
   ---------------------------------------------------------------- *}

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
   ---------------------------------------------------------------- *}

text {* ----------------------------------------------------------------
  Ejercicio 9. 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
   ---------------------------------------------------------------- *}

text {* ----------------------------------------------------------------
  Ejercicio 10. 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
   ------------------------------------------------------------------ *}

text {* ------------------------------------------------------------------
  Ejercicio 11. 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
   ------------------------------------------------------------------ *}

text {* ------------------------------------------------------------------
  Ejercicio 12. 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
   ------------------------------------------------------------------ *}

end
```

Relación 7

Programación funcional con Isabelle/HOL

```
header {* R7: Programación funcional en Isabelle *}

theory R7
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 xs = undefined"

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) = undefined"

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 xs = undefined"

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 n x = undefined"

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 xs ys = undefined"

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 xs = undefined"

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 xs = undefined"

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 xs = undefined"

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 xs ys = undefined"

fun inversaAc :: "'a list ⇒ 'a list" where
  "inversaAc xs = undefined"

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 xs = undefined"

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 (λx. 2*x) [3,2,5] = [6,4,10]
  --------------------------------------------------------------- *}
```

```
fun map :: "('a ⇒ 'b) ⇒ 'a list ⇒ 'b list" where
  "map f xs = undefined"

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

end
```

Relación 8

Razonamiento sobre programas

```
header {* R8: Razonamiento sobre programas en Isabelle/HOL *}

theory R8
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 n = undefined"

value "sumaImpares 5" -- "= 25"

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

lemma "sumaImpares n = n*n"
oops
```

```
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 n = undefined"

value "sumaPotenciasDeDosMasUno 3" -- "= 16"

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

lemma "sumaPotenciasDeDosMasUno n = 2^(n+1)"
oops

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 n x = undefined"

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

fun todos :: "('a ⇒ bool) ⇒ 'a list ⇒ bool" where
  "todos p xs = undefined"

value "todos (λx. x>(1::nat)) [2,6,4]" -- "= True"
value "todos (λx. 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 (λy. y=x) (copia n x)"
oops

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 n = undefined"

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"
oops

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

corollary "factI n = factR n"
oops

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 xs y = undefined"

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

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

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

lemma "amplia xs y = xs @ [y]"
oops

end
```

Relación 9

Cons inverso

```
header {* R9: Cons inverso *}

theory R9
imports Main
begin

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 xs a = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar el siguiente teorema
     snoc xs a = xs @ [a]
  ----------------------------------------------------------------------
*}
```

```
lemma snoc_append:
  "snoc xs a = xs @ [a]"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Demostrar el siguiente teorema
     rev (x # xs) = snoc (rev xs) x"
  ----------------------------------------------------------------------
*}

theorem rev_cons:
  "rev (x # xs) = snoc (rev xs) x"
oops

end
```

Relación 10

Cuantificadores sobre listas

```
header {* R10: Cuantificadores sobre listas *}

theory R10
imports Main
begin

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. 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 (λx. 1<length x) [[2,1,4],[1,3]]
     ¬todos (λx. 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 xs = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. 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 (λx. 1<length x) [[2,1,4],[3]]
     ¬algunos (λx. 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 xs = undefined"

text {*
  ------------------------------------------------------------------------
  Ejercicio 3. Demostrar o refutar:
     todos (λx. P x ∧ Q x) xs = (todos P xs ∧ todos Q xs)
  ------------------------------------------------------------------------
*}

lemma "todos (λx. P x ∧ Q x) xs = (todos P xs ∧ todos Q xs)"
oops

text {*
  ------------------------------------------------------------------------
  Ejercicio 4. Demostrar o refutar:
     todos P (x @ y) = (todos P x ∧ todos P y)
  ------------------------------------------------------------------------
*}

lemma todos_append:
  "todos P (x @ y) = (todos P x ∧ todos P y)"
oops

text {*
  ------------------------------------------------------------------------
  Ejercicio 5. Demostrar o refutar:
     todos P (rev xs) = todos P xs
  ------------------------------------------------------------------------
*}

lemma "todos P (rev xs) = todos P xs"
```

```
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 6. Demostrar o refutar:
    algunos (λx. P x ∧ Q x) xs = (algunos P xs ∧ algunos Q xs)
  ----------------------------------------------------------------------
*}

lemma "algunos (λx. P x ∧ Q x) xs = (algunos P xs ∧ algunos Q xs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Demostrar o refutar:
     algunos P (map f xs) = algunos (P ∘ f) xs
  ----------------------------------------------------------------------
*}

lemma "algunos P (map f xs) = algunos (P o f) xs"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 8. Demostrar o refutar:
     algunos P (xs @ ys) = (algunos P xs ∨ algunos P ys)
  ----------------------------------------------------------------------
*}

lemma algunos_append:
  "algunos P (xs @ ys) = (algunos P xs ∨ algunos P ys)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Demostrar o refutar:
     algunos P (rev xs) = algunos P xs
  ----------------------------------------------------------------------
*}
```

```
lemma "algunos P (rev xs) = algunos P xs"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Encontrar un término no trivial Z tal que sea cierta la
  siguiente ecuación:
     algunos (λx. P x ∨ Q x) xs = Z
  ----------------------------------------------------------------------
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 11. Demostrar o refutar:
     algunos P xs = (¬ todos (λx. (¬ P x)) xs)
  ----------------------------------------------------------------------
*}

lemma "algunos P xs = (¬ todos (λx. (¬ P x)) xs)"
by (induct xs) simp_all

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. 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 xs = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 13. Expresar la relación existente entre estaEn y algunos.
  ----------------------------------------------------------------------
*}
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 14. 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 xs = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 15. 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 xs = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 16. Demostrar o refutar:
     length (borraDuplicados xs) ≤ length xs
  ----------------------------------------------------------------------
*}

lemma length_borraDuplicados:
  "length (borraDuplicados xs) ≤ length xs"
```

```
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 17. Demostrar o refutar:
     estaEn a (borraDuplicados xs) = estaEn a xs
  ------------------------------------------------------------------
*}

lemma estaEn_borraDuplicados:
  "estaEn a (borraDuplicados xs) = estaEn a xs"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 18. Demostrar o refutar:
     sinDuplicados (borraDuplicados xs)
  ------------------------------------------------------------------
*}

lemma sinDuplicados_borraDuplicados:
  "sinDuplicados (borraDuplicados xs)"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 19. Demostrar o refutar:
    borraDuplicados (rev xs) = rev (borraDuplicados xs)
  ------------------------------------------------------------------
*}

lemma "borraDuplicados (rev xs) = rev (borraDuplicados xs)"
oops

end
```

Relación 11

Sustitución, inversión y eliminación

```
header {* R11: Sustitución, inversión y eliminación *}

theory R11
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 zs = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar o refutar:
     sust x y (xs@ys) = (sust x y xs)@(sust x y ys)"
  ----------------------------------------------------------------------
*}
```

```
lemma sust_append:
  "sust x y (xs@ys) = (sust x y xs)@(sust x y ys)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Demostrar o refutar:
     rev (sust x y zs) = sust x y (rev zs)
  ----------------------------------------------------------------------
*}

lemma rev_sust:
  "rev(sust x y zs) = sust x y (rev zs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Demostrar o refutar:
     sust x y (sust u v zs) = sust u v (sust x y zs)
  ----------------------------------------------------------------------
*}

lemma
  "sust x y (sust u v zs) = sust u v (sust x y zs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Demostrar o refutar:
     sust y z (sust x y zs) = sust x z zs
  ----------------------------------------------------------------------
*}

lemma
  "sust y z (sust x y zs) = sust x z zs"
oops

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 ys = undefined"

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 ys = undefined"

text {*
  ------------------------------------------------------------------
  Ejercicio 8. Demostrar o refutar:
     borra x (borraTodas x xs) = borraTodas x xs
  ------------------------------------------------------------------
*}

lemma
  "borra x (borraTodas x xs) = borraTodas x xs"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 9. Demostrar o refutar:
```

```
     borraTodas x (borraTodas x xs) = borraTodas x xs
  ------------------------------------------------------------------
*}

lemma
  "borraTodas x (borraTodas x xs) = borraTodas x xs"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 10. Demostrar o refutar:
     borraTodas x (borra x xs) = borraTodas x xs
  ------------------------------------------------------------------
*}

lemma
  "borraTodas x (borra x xs) = borraTodas x xs"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 11. Demostrar o refutar:
     borra x (borra y xs) = borra y (borra x xs)
  ------------------------------------------------------------------
*}

lemma
  "borra x (borra y xs) = borra y (borra x xs)"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 12. Demostrar o refutar el teorema:
     borraTodas x (borra y xs) = borra y (borraTodas x xs)
  ------------------------------------------------------------------
*}

lemma
  "borraTodas x (borra y xs) = borra y (borraTodas x xs)"
oops
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 13. Demostrar o refutar:
     borra y (sust x y xs) = borra x xs
  ----------------------------------------------------------------------
*}

lemma
  "borra y (sust x y xs) = borra x xs"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 14. Demostrar o refutar:
     borraTodas y (sust x y xs) = borraTodas x xs"
  ----------------------------------------------------------------------
*}

lemma
  "borraTodas y (sust x y xs) = borraTodas x xs"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 15. Demostrar o refutar:
     sust x y (borraTodas x zs) = borraTodas x zs
  ----------------------------------------------------------------------
*}

lemma
  "sust x y (borraTodas x zs) = borraTodas x zs"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 16. Demostrar o refutar:
     sust x y (borraTodas z zs) = borraTodas z (sust x y zs)
  ----------------------------------------------------------------------
*}
```

```
lemma
  "sust x y (borraTodas z zs) = borraTodas z (sust x y zs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 17. Demostrar o refutar:
     rev (borra x xs) = borra x (rev xs)
  ----------------------------------------------------------------------
*}

lemma
  "rev (borra x xs) = borra x (rev xs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 18. Demostrar o refutar el teorema:
     borraTodas x (xs@ys) = (borraTodas x xs)@(borraTodas x ys)
  ----------------------------------------------------------------------
*}

lemma
  "borraTodas x (xs@ys) = (borraTodas x xs)@(borraTodas x ys)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 19. Demostrar o refutar el teorema:
     rev (borraTodas x xs) = borraTodas x (rev xs)
  ----------------------------------------------------------------------
*}

lemma
  "rev (borraTodas x xs) = borraTodas x (rev xs)"
oops

end
```

Relación 12

Menor posición válida

```
header {* R12: Menor posición válida *}

theory R12
imports Main
begin

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 (λx. 4<x) [1::nat, 3, 5, 3, 1]     = 2
     menorValida (λx. 6<x) [1::nat, 3, 5, 3, 1]     = 5
     menorValida (λx. 1<length x) [[], [1, 2], [3]] = 1
  ----------------------------------------------------------------------
*}

fun menorValida :: "('a ⇒ bool) ⇒ 'a list ⇒ nat" where
  "menorValida P xs = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar que menorValida devuelve la longitud de la
  lista syss ningún elemento satisface el predicado dado.
  ----------------------------------------------------------------------
```

```
*}

lemma "(menorValida P xs = length xs) = list_all (λ x. (¬ P x)) xs"
oops

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.
  ---------------------------------------------------------------------
*}

lemma "list_all (λx. ¬ P x) (take (menorValida P xs) xs)"
oops

text {*
  ---------------------------------------------------------------------
  Ejercicio 4. ¿Cómo se puede relacionar
    "menorValida (λx. 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 {*
  ---------------------------------------------------------------------
  Ejercicio 5.  Si P implica Q, ¿qué relación puede deducirse entre
  "menorValida P xs" y "menorValida Q xs"? Prueba tu conjetura.
  ---------------------------------------------------------------------
*}


end
```

Relación 13

Número de elementos válidos

```
header {* R13: Número de elementos válidos *}

theory R13
imports Main
begin

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 (λx. 2<x) [1,3,4,0,5] = 3
  ----------------------------------------------------------------------
*}

fun cuentaP :: "('a ⇒ bool) ⇒ 'a list ⇒ nat" where
  "cuentaP Q xs = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar o refutar:
     cuentaP P (xs @ ys) = cuentaP P xs + cuentaP P ys*
  ----------------------------------------------------------------------
*}

lemma cuentaP_append_auto:
```

```
  "cuentaP P (xs @ ys) = cuentaP P xs + cuentaP P ys"
oops

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.
  ------------------------------------------------------------------
*}

lemma cuentaP_rev:
  "cuentaP P xs = cuentaP P (rev xs)"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 4. Encontrar y demostrar una relación entre las funciones
  filter y  cuentaP.
  ------------------------------------------------------------------
*}

end
```

Relación 14

Contador de occurrencias

```
header {* R14: Contador de occurrencias *}

theory R14
imports Main
begin

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 ys = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar o refutar:
     veces a (xs @ ys) = veces a xs + veces a ys
  ----------------------------------------------------------------------
*}

lemma veces_append:
```

```
  "veces a (xs @ ys) = veces a xs + veces a ys"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 3. Demostrar o refutar:
     veces a xs = veces a (rev xs)
  ----------------------------------------------------------------------
*}

lemma veces_rev:
  "veces a xs = veces a (rev xs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Demostrar o refutar:
    "veces a xs ≤ length xs".
  ----------------------------------------------------------------------
*}

lemma veces_le_length_auto:
  "veces a xs ≤ length xs"
oops

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
  ----------------------------------------------------------------------
*}

lemma veces_map:
  "veces a (map f xs) = veces (f a) xs"
oops

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
  y demostrar la conjetura.
  ------------------------------------------------------------------
*}

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 xs = undefined"

text {*
  ------------------------------------------------------------------
  Ejercicio 8. Encontrar una expresión e que no contenga la función
  borraDups tal que se verifique
     veces x (borraDups xs) = e
  y demostrar la conjetura.
  ------------------------------------------------------------------
*}

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]
     ¬ distintos [1,4,1]

  Nota: La función "distintos" es equivalente a la predefinida "distinct".
  ------------------------------------------------------------------------
*}

fun distintos :: "'a list ⇒ bool" where
  "distintos xs = undefined"

text {*
  ------------------------------------------------------------------------
  Ejercicio 10. Demostrar que el valor de "borraDups" verifica
  "distintos".
  ------------------------------------------------------------------------
*}

lemma distintos_borraDups:
  "distintos (borraDups xs)"
oops

end
```

Relación 15

Suma y aplanamiento de listas

```
header {* R15: Suma y aplanamiento de listas *}

theory R15
imports Main R10
begin

section {* Suma y aplanamiento de listas *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     suma :: "nat list \<Rightarrow> 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 \<Rightarrow> nat" where
  "suma xs = undefined"

value "suma [3::nat,2,4]" -- "= 9"

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Definir la función
     aplana :: "'a list list \<Rightarrow> '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 \<Rightarrow> 'a list" where
  "aplana xs = undefined"

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)
  ----------------------------------------------------------------------
*}

lemma length_aplana:
  "length (aplana xs) = suma (map length xs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Demostrar o refutar
     suma (xs @ ys) = suma xs + suma ys
  ----------------------------------------------------------------------
*}

lemma suma_append:
  "suma (xs @ ys) = suma xs + suma ys"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Demostrar o refutar
     aplana (xs @ ys) = (aplana xs) @ (aplana ys)
  ----------------------------------------------------------------------
*}
```

```
lemma aplana_append:
  "aplana (xs @ ys) = (aplana xs) @ (aplana ys)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 6. Demostrar o refutar
     aplana (map rev (rev xs)) = rev (aplana xs)
  ----------------------------------------------------------------------
*}

lemma aplana_map_rev_rev:
  "aplana (map rev (rev xs)) = rev (aplana xs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Demostrar o refutar
     aplana (rev (map rev xs)) = rev (aplana xs)
  ----------------------------------------------------------------------
*}

lemma aplana_rev_map_rev:
  "aplana (rev (map rev xs)) = rev (aplana xs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 8. Demostrar o refutar
     list_all (list_all P) xs = list_all P (aplana xs)
  ----------------------------------------------------------------------
*}

lemma list_all_list_all:
  "list_all (list_all P) xs = list_all P (aplana xs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Demostrar o refutar
```

```
     aplana (rev xs) = aplana xs
  ------------------------------------------------------------------
*}

lemma aplana_rev:
  "aplana (rev xs) = aplana xs"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 10. Demostrar o refutar
     suma (rev xs) = suma xs
  ------------------------------------------------------------------
*}

lemma suma_rev:
  "suma (rev xs) = suma xs"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 11. Buscar un predicado P para que se verifique la siguiente
  propiedad
     list_all P xs \<longrightarrow> length xs \<le> suma xs
  ------------------------------------------------------------------
*}

text {*
  ------------------------------------------------------------------
  Ejercicio 12. Demostrar o refutar
     algunos (algunos P) xs = algunos P (aplana xs)
  ------------------------------------------------------------------
*}

lemma algunos_algunos:
  "algunos (algunos P) xs = algunos P (aplana xs)"
oops

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 \<Rightarrow> bool) \<Rightarrow> ('a list \<Rightarrow> bool)"
  "algunos2 P xs = undefined"

lemma algunos2_algunos:
  "algunos2 P xs = algunos P xs"
oops

end
```

Relación 16

Conjuntos mediante listas

```
header {* R16: Conjuntos mediante listas *}

theory R16
imports Main
begin

text {*
  Los conjuntos finitos se pueden representar mediante listas. En esta
  relación se define la unión y se demuestran algunas de sus
  propiedades. Análogamente se puede hacer con la intersección y
  diferencia.
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 1. Definir la función
     union_l :: "'a list \<Rightarrow> 'a list \<Rightarrow> '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 \<Rightarrow> 'a list \<Rightarrow> 'a list" where
  "union_l xs ys = undefined"

value "union_l [1::nat,2] [2,3]" -- "= [1,2,3]"
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar o refutar
     set (union_l xs ys) = set xs \<union> set ys
  ----------------------------------------------------------------------
*}

lemma set_union_l: "set (union_l xs ys) = set xs \<union> set ys"
oops

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:
  assumes "distinct xs"
          "distinct ys"
  shows   "distinct (union_l xs ys)"
oops

section {* Cuantificación sobre conjuntos *}

text {*
  ----------------------------------------------------------------------
  Ejercicio 4. Definir un conjunto S para que se verifique que
     (\<forall>x\<in>A. P x) \<and> (\<forall>x\<in> B. P x) \<longrightarrow> (\<for
  ----------------------------------------------------------------------
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 5. Definir una propiedad P para que se verifique que
     \<forall>x\<in> A. P (f x) \<Longrightarrow>  \<forall>y\<in> f ‘ A. Q y
  ----------------------------------------------------------------------
```

```
*}

end
```

Relación 17

Ordenación de listas por inserción

```
header {* R17: Ordenación de listas por inserción *}

theory R17
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 xs = undefined"

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,
     ordena [3,2,5,3] = [2,3,3,5]
  ------------------------------------------------------------------ *}

fun ordena :: "nat list ⇒ nat list" where
  "ordena xs = undefined"

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 xs = undefined"

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 xs = undefined"
```

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

lemma menor_menor:
  assumes "x ≤ y"
  shows   "menor y xs ⟶ menor x xs"
oops

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.
  ---------------------------------------------------------------- *}

lemma menor_inserta:
  "menor x (inserta y zs) = (x ≤ y ∧ menor x zs)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 6. Demostrar que al insertar un elemento la lista obtenida
  está ordenada syss lo estaba la original.
  ---------------------------------------------------------------- *}

lemma ordenada_inserta:
  "ordenada (inserta a xs) = ordenada xs"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 7. Demostrar que, para toda lista xs, (ordena xs) está
  ordenada.
```

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

theorem ordenada_ordena:
  "ordenada (ordena xs)"
oops

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 xs y = undefined"

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;
  ------------------------------------------------------------------ *}

lemma cuenta_inserta:
  "cuenta (inserta x xs) y =
   (if x=y then Suc (cuenta xs y) else cuenta xs y)"
```

```
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Demostrar que el número de veces que aparece y en
  (ordena xs) es el número de veces que aparece en xs.
  ---------------------------------------------------------------------- *}

theorem cuenta_ordena:
  "cuenta (ordena xs) y = cuenta xs y"
oops

end
```

Relación 18

Ordenación de listas por mezcla

```
header {* R18: Ordenación de listas por mezcla *}

theory R18
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 xs = undefined"

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 xs = undefined"

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 xs y = undefined"

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 xs ys = undefined"

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 xs = undefined"

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"
oops

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"
oops

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)"
oops

text {*
  ---------------------------------------------------------------------
  Ejercicio 9. Demostrar que la mezcla de dos listas ordenadas es una
  lista ordenada.
  Indicación: Usar los siguientes lemas
  · linorder_not_le: (¬ x ≤ y) = (y < x)
  · order_less_le:   (x < y) = (x ≤ y ∧ x ≠ y)
  ---------------------------------------------------------------- *}

lemma ordenada_mezcla:
  assumes "ordenada xs"
          "ordenada ys"
  shows   "ordenada (mezcla xs ys)"
oops

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
  · min_def:         min a b = (if a ≤ b then a else b)
  · linorder_not_le: (¬ x ≤ y) = (y < x)
  ---------------------------------------------------------------- *}
```

```
lemma min_mitad:
  "1 < x ⟹ min x (x div 2::nat) < x"
oops

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"
oops

text {*
  ---------------------------------------------------------------------
  Ejercicio 11. Demostrar que (ordenaM xs) está ordenada.
  ---------------------------------------------------------------------- *}

theorem ordenada_ordenaM:
  "ordenada (ordenaM xs)"
oops

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"
oops

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"
oops

end
```

Relación 19

Recorridos de árboles

```
header {* R19: Recorridos de árboles *}

theory R19
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 t = undefined"

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 t = undefined"

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 t = undefined"

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 t = undefined"

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)"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 7. Demostrar que
     postOrden (espejo a) = rev (preOrden a)
  ------------------------------------------------------------------
```

```
*}

lemma "postOrden (espejo a) = rev (preOrden a)"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 8. Demostrar que
     inOrden (espejo a) = rev (inOrden a)
  ----------------------------------------------------------------------
*}

theorem "inOrden (espejo a) = rev (inOrden a)"
oops

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 t = undefined"

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 t = undefined"

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 t = undefined"

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
  ----------------------------------------------------------------------
*}

theorem "last (inOrden a) = extremo_derecha a"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 13. Demostrar o refutar
     hd (inOrden a) = extremo_izquierda a
  ----------------------------------------------------------------------
*}

theorem "hd (inOrden a) = extremo_izquierda a"
oops

text {*
```

```
  ------------------------------------------------------------------
  Ejercicio 14. Demostrar o refutar
     hd (preOrden a) = last (postOrden a)
  ------------------------------------------------------------------
*}

theorem "hd (preOrden a) = last (postOrden a)"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 15. Demostrar o refutar
     hd (preOrden a) = raiz a
  ------------------------------------------------------------------
*}

theorem "hd (preOrden a) = raiz a"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 16. Demostrar o refutar
     hd (inOrden a) = raiz a
  ------------------------------------------------------------------
*}

theorem "hd (inOrden a) = raiz a"
oops

text {*
  ------------------------------------------------------------------
  Ejercicio 17. Demostrar o refutar
     last (postOrden a) = raiz a
  ------------------------------------------------------------------
*}

theorem "last (postOrden a) = raiz a"
oops

end
```

Relación 20

Plegados de listas y de árboles

```
header {* R20: Plegados de listas y de árboles *}

theory R20
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 xs = undefined"

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"
oops

text {*
  ----------------------------------------------------------------------
  Ejercicio 2. Demostrar que
     length xs = foldr (λ x res. 1 + res) xs 0
  ----------------------------------------------------------------------
*}

lemma length_foldr: "length xs = foldr (λ x res. 1 + res) xs 0"
oops

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 (λx. x + 3) xs) = foldr h xs b
  Determinar los valores de h y b para que se verifique la igualdad
  anterior y demostrarla.
```

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


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 {*
  ------------------------------------------------------------------
  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:
  "inversa_ac_aux xs a = foldl undefined a xs"
oops
```

```
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"
oops

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 {*
  ----------------------------------------------------------------------
  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 ≡ undefined"

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"
oops

section {* Functiones sobre árboles *}

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
       / \   / \
      .   . .   .
  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 t = undefined"
```

```
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 t = undefined"

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 t = undefined"

definition postOrdenA :: "'a arbol ⇒ 'a list" where
  "postOrdenA a ≡ undefined"

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 "postOrdenAaux a xs = (postOrden a) @ xs"
oops

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 t = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 16. Demostrar que
     postOrdenAaux t a = foldl_arbol (λ xs x. Cons x xs) a t
  ----------------------------------------------------------------------
*}

lemma "postOrdenAaux t a = foldl_arbol (λ xs x. Cons x xs) a t"
oops

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 t = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 18. Demostrar que
     suma_arbol a = suma (preOrden a)"
  ----------------------------------------------------------------------
*}

lemma "suma_arbol a = suma (preOrden a)"
oops

end
```

Relación 21

Árboles binarios completos

```
header {* R21: Árboles binarios completos *}

theory R21
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
          .
         / \
        /   \
       .     .
      / \   / \
     .   . .   .
  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 t = undefined"

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 t = undefined"

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 = undefined"

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 t = undefined"

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 {*
  ----------------------------------------------------------------------
  Ejercicio 8. Demostrar que un árbol binario a es completo respecto del
  número de hojas syss es completo respecto del número de nodos
  ----------------------------------------------------------------------
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 9. Demostrar que un árbol binario a es completo respecto de
  la profundidad syss es completo respecto del número de nodos
  ----------------------------------------------------------------------
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 10. Demostrar que (abc n) es un árbol binario completo.
  ----------------------------------------------------------------------
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 11. Demostrar que si a es un árbolo binario completo
  respecto de la profundidad, entonces a es (abc (profundidad a)).
  ----------------------------------------------------------------------
*}

text {*
  ----------------------------------------------------------------------
  Ejercicio 12. Encontrar una medida f tal que (es_abc f) es distinto de
  (es_abc size).
  ----------------------------------------------------------------------
*}

end
```

Relación 22

Diagramas de decisión binarios

```
header {* R22: Diagramas de decisión binarios *}

theory R22
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 a = undefined"

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 (λx. ¬x) 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 a = undefined"

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)"
oops

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 d1 d2 = undefined"

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)"
oops

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 ≡ undefined"

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)"
oops

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 ≡ undefined"

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)"
oops

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 ≡ undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 14. Demostrar que la definición de ddb_not es correcta;
  es decir,
     valor i n (ddb_not d) = (¬ valor i n d)
  ----------------------------------------------------------------------
*}

theorem ddb_not_correcta:
  "valor i n (ddb_not d) = (¬ valor i n d)"
oops

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 ≡ undefined"

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 ≡ undefined"

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)"
oops

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 n = undefined"

text {*
  ----------------------------------------------------------------------
  Ejercicio 19. Demostrar que la definición de ddb_var es correcta;
  es decir,
     "valor i 0 (ddb_var n) = i n
  ----------------------------------------------------------------------
*}

lemma ddb_var_correcta:
```

```
  "valor i 0 (ddb_var n) = i n"
oops

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 (λn. True) (Xor T T)        = False
     valor_fla (λn. False) (Xor T (Var 1)) = True
  ----------------------------------------------------------------------
*}

fun valor_fla :: "(nat ⇒ bool) ⇒ form ⇒ bool" where
  "valor_fla i f = undefined"

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 f = undefined"

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"
oops

text {*
  Referencias:
  · 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
  · Wikipedia "Binary decision diagram". En
       http://en.wikipedia.org/wiki/Binary_decision_diagram
*}

end
```

Relación 23

Representación de fórmulas proposicionales mediante polinomios

```
header {* R23: Representación de fórmulas proposicionales mediante
  polinomios *}

theory R23
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\<^bsub>1\<^esub> ∧ ... ∧ p\<^bsub>n\<^esub>) ⊕ ... ⊕ (q\<^bsub>1\<^esub> ∧
  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:
  · ⊤ es una fórmula proposicional
  · Las variables proposicionales p_1, p_2, ... son fórmulas
    proposicionales,
  · 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
  · T en lugar de ⊤,
  · (Var i) en lugar de p_i,
  · (And F G) en lugar de (F ∧ G) y
  · (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)
  es usará en lo que sigue.
  ------------------------------------------------------------------ *}

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 ≡ undefined"

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 f = undefined"
```

```
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 ns = undefined"

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 ns = undefined"

lemma correccion_valorM:
  "valorM i m = valorF i (formM m)"
oops

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 ms = undefined"

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 ms = undefined"

lemma correccion_valorP:
  "valorP i p = valorF i (formP p)"
oops

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 ns = undefined"

text {*
  ----------------------------------------------------------------
  Ejercicio 11. Demostrar que, en cualquier interpretación i, el valor
  de la concatenación de dos monomios es la conjunción de sus valores.
  ---------------------------------------------------------------- *}

lemma valorM_conc:
  "valorM i (xs @ ys) = (valorM i xs ∧ valorM i ys)"
oops

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.
  ---------------------------------------------------------------- *}

lemma correccion_productoM:
  "valorP i (productoM m p) = (valorM i m ∧ valorP i p)"
oops

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 p q = undefined"

text {*
  ---------------------------------------------------------------------
  Ejercicio 14. Demostrar que, en cualquier interpretación i, el valor
  de la concatenación de dos polinomios es la disyunción exclusiva de
  sus valores.
  --------------------------------------------------------------------- *}

lemma valorP_conc:
  "valorP i (xs @ ys) = (xor (valorP i xs) (valorP i ys))"
oops

text {*
  ---------------------------------------------------------------------
  Ejercicio 15. Demostrar que, en cualquier interpretación i, el valor
  del producto de dos polinomios es la conjunción de sus valores.
  --------------------------------------------------------------------- *}

lemma correccion_producto:
  "valorP i (producto p q) = (valorP i p ∧ valorP i q)"
oops

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 f = undefined"
```

```
text {*
  ----------------------------------------------------------------------
  Ejercicio 17. Demostrar que, en cualquier interpretación i, el valor
  de f es igual que el de su polinomio.
  ---------------------------------------------------------------------- *}

theorem correccion_polinomio:
  "valorF i f = valorP i (polinomio f)"
oops

end
```

Bibliografía

[1] J. A. Alonso. Temas de "Programación funcional". Technical report, Universidad de Sevilla, 2012.

[2] J. A. Alonso. Temas de "Lógica informática" (2012-13). Technical report, Universidad de Sevilla, 2012.

[3] K. Doets and J. van Eijck. The Haskell Road to Logic, Maths and Programming. Technical report, ILLC (Institute for Logic, Language and Computation), Amsterdam, 2004.

[4] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[5] T. Nipkow, M. Wenzel, and L. C. Paulson. *A Proof Assistant for Higher-Order Logic*. Springer–Verlag, 2013.

[6] J. Siek. Practical Theorem Proving with Isabelle/Isar Lecture Notes. Technical report, University of Colorado at Boulder, 2007.