Introducción a la demostración asistida por ordenador (con Isabelle/Isar)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 5 de abril de 2008 (versión de 29 de junio de 2008)

Índice

Capítulo 1

Isabelle como un lenguaje funcional

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. El ejemplo del lema más trivial es el siguiente

---

lemma *elMasTrivial*: *True*
by *simp*

En este capítulos se presenta el lenguaje funcional que está incluido en Isabelle. El lenguaje funcional es muy parecido al ML estándard.

1.1 Números naturales, enteros y booleanos

Nota 1.1.1 (Números naturales)**.**

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

Lema 1.1.2 (Ejemplo de simplificación de números naturales)**.** *El siguiente del* 0 *es el* 1*.*

---

lemma *Suc 0 = 1*
by *simp*

Nota 1.1.3 (Suma y producto de números naturales)**.** En Isabelle están definida la suma '+' y el producto '*' de números naturales.

Lema 1.1.4 (Ejemplo de suma)**.** *La suma de los números naturales 1 y 2 es el número natural 3.*

---

lemma $1 + 2 = (3::nat)$
by *simp*

Nota 1.1.5 (Epecificación de tipo)**.** La notación del par de dos puntos se usa para asignar un tipo a un término (por ejemplo, `3::nat` significa que se considera que `3` es un número natural).

Lema 1.1.6 (Ejemplo de producto)**.** *El producto de los números naturales 2 y 3 es el número natural 6.*

---

lemma $2 * 3 = (6::nat)$
by *simp*

Nota 1.1.7 (División de números naturales)**.** En Isabelle está definida la división de números naturales: *n div m* es el mayor número natural que multiplicado por *m* es menor o igual que *n*.

Lema 1.1.8 (Ejemplo de división)**.** *La división natural de 7 entre 3 es 2.*

---

lemma $7\ div\ 3 = (2::nat)$
by *simp*

Nota 1.1.9 (Resto de división de números naturales)**.** En Isabelle está definida el resto de números naturales: *n mod m* es el resto de dividir *n* entre *m*.

Lema 1.1.10 (Ejemplo de resto)**.** *El resto de dividir 7 entre 3 es 1.*

---

lemma $7\ mod\ 3 = (1::nat)$
by *simp*

Nota 1.1.11 (Números enteros)**.** En Isabelle también están definidos los números enteros. El tipo de los enteros se representa por *int*.

Lema 1.1.12 (Ejemplo de operación con enteros)**.** *La suma de 1 y -2 es el número entero -1.*

---

lemma $1 + -2 = (-1::int)$
by *simp*

Nota 1.1.13 (Sobrecarga)**.** 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.

Nota 1.1.14 (Booleanos, conectivas y cuantificadores)**.** En Isabelle están definidos los valores booleanos (*True* y *False*), las conectivas ($\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\leftrightarrow$) y los cuantificadores ($\forall$, $\exists$). El tipo de los booleanos es *bool*.

Lema 1.1.15 (Ejemplos de evaluaciones booleanas)**.**

1. *La conjunción de dos fórmulas verdaderas es verdadera.*
2. *La conjunción de un fórmula verdadera y una falsa es falsa.*
3. *La disyunción de una fórmula verdadera y una falsa es verdadera.*
4. *La disyunción de dos fórmulas falsas es falsa.*
5. *La negación de una fórmula verdadera es falsa.*
6. *Una fórmula falsa implica una fórmula verdadera.*
7. *Todo elemento es igual a sí mismo.*
8. *Existe un elemento igual a 1.*

---

lemma $True \wedge True = True$
by *simp*

lemma $True \wedge False = False$
by *simp*

lemma $True \vee False = True$
by *simp*

lemma $False \vee False = False$
by *simp*

lemma $\neg\ True = (False{::}bool)$
by *simp*

lemma $False \longrightarrow True$
by *simp*

lemma $\forall\ x.\ x = x$

by *simp*

lemma $\exists\ x.\ x = 1$
by *simp*

1.2 Definiciones no recursivas"

Definición 1.2.1 (Ejemplo de definición no recursiva)**.** *La disyunción exclusiva de A y B se verifica si una es verdadera y la otra no lo es.*

---

constdefs *xor* :: *bool* ⇒ *bool* ⇒ *bool*
xor A B ≡ $(A \wedge \neg B) \vee (\neg A \wedge B)$

Lema 1.2.2 (Ejemplo de demostración con definiciones no recursivas)**.** *La disyunción exclusiva de dos fórmulas verdaderas es falsa.*

Demostración: Por simplificación, usando la definición de la disyunción exclusiva.
□

---

lemma *xor True True* = *False*
by (*simp add*: *xor-def*)

Nota 1.2.3 (Ejemplo de ampliación de las reglas de simplificación)**.** Se añade la definición de la disyunción exlusiva al conjunto de reglas de simplificación automáticas.

---

declare *xor-def*[*simp*]

1.3 Definiciones locales

Nota 1.3.1 (Variables locales)**.** Se puede asignar valores a variables locales mediante 'let' y usarlo en las expresiones dentro de 'in'.

Lema 1.3.2 (Ejemplo de entorno local)**.** *Sea x el número natural 3. Entonces* $x \times x = 9$*.*

---

lemma (*let x* = *3*::*nat in x* ∗ *x* = *9*)
by *simp*

1.4 Pares

Nota 1.4.1 (Pares)**.**

- 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.

Lema 1.4.2 (Ejemplo de uso de pares)**.** *Sea p el par de números naturales* $(2,3)$*. La suma del primer elemento de p y* 1 *es igual al segundo elemento de p.*

---

lemma *let p* $=$ *(2,3)*$::nat \times nat$ *in fst p* $+$ *1* $=$ *snd p*
by *simp*

1.5 Listas

Nota 1.5.1 (Construcción de listas)**.**

- Una lista se representa escribiendo los elementos entre corchetes y separados por coma.
- 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 `a#l` representa la lista obtenida añadiendo el elemento `a` al principio de la lista `l`.

Lema 1.5.2 (Ejemplo de construcción de listas)**.** *La lista obtenida añadiendo sucesivamente a la lista vacía los elementos 3, 2 y 1 es [1,2,3].*

---

lemma *1*#(*2*#(*3*#[])) $=$ [*1*,*2*,*3*]
by *simp*

Nota 1.5.3 (Primero y resto)**.**

- `hd l` es el primer elemento de la lista `l`.
- `tl l` es el resto de la lista `l`.

Lema 1.5.4 (Ejemplo de cálculo con listas)**.** *Sea* `l` *la lista de números naturales* `[1,2,3]`*. Entonces, el primero de* `l` *es 1 y el resto de* `l` *es* `[2,3]`*.*

lemma *let l = [1,2,3]::(nat list) in hd l = 1 ∧ tl l = [2,3]*
by *simp*

Nota 1.5.5 (Longitud)**.** `length l` es la longitud de la lista `l`.

Lema 1.5.6 (Ejemplo de cálculo de longitud)**.** *La lonngitud de la lista [1,2,3] es 3.*

lemma *length [1,2,3] = 3*
by *simp*

Nota 1.5.7 (Referencias sobre listas)**.** En la sesión 38 de "HOL: The basis of Higher-Order Logic" se encuentran más definiciones y propiedades de las listas.

1.6 Registros

Nota 1.6.1 (Registro)**.** Un registro es una colección de campos y valores.

Definición 1.6.2 (Ejemplo de definición de registro)**.** *Los puntos del plano pueden representarse mediante registros con dos campos, las coordenadas, con valores enteros.*

record *punto* =
coordenada-x :: *int*
coordenada-y :: *int*

Definición 1.6.3 (Ejemplo de definición de un registro)**.** *El punto* `pt` *tiene de coordenadas 3 y 7.*

constdefs *pt* :: *punto*
pt ≡ (|*coordenada-x* = 3, *coordenada-y* = 7|)

Lema 1.6.4 (Ejemplo de propiedad de registro)**.** *La coordenada x del punto* `pt` *es 3.*

lemma *coordenada-x pt = 3*
by (*simp add*: *pt-def*)

Lema 1.6.5 (Ejemplo de actualización de un registro)**.** *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.*

lemma *let pt2=pt(|coordenada-x:=4|) in coordenada-x (pt2) = 4*
by (*simp add*: *pt-def*)

1.7 Funciones anónimas

Nota 1.7.1 (Funciones anónimas)**.** En Isabelle pueden definirse funciones anónimas.

Lema 1.7.2 (Ejemplo de uso de funciones anónimas)**.** *El valor de la función que a un número le asigna su doble aplicada a 1 es 2.*

---

lemma $(\lambda\, x.\ x + x)\ 1 = (2{::}nat)$
by *simp*

1.8 Condicionales

Definición 1.8.1 (Ejemplo con el condicional *if*)**.** *El valor absoluto del entero x es x, si* $x \geq 0$ *y es* $-x$ *en caso contrario.*

---

constdefs *absoluto* :: $int \Rightarrow int$
$absoluto\ x \equiv (if\ x \geq 0\ then\ x\ else\ -x)$

Lema 1.8.2 (Ejemplo de simplificación con el condicional *if*)**.** *El valor absoluto de -3 es 3.*

---

lemma $absoluto(-3) = 3$
by (*simp add*:*absoluto-def*)

Definición 1.8.3 (Ejemplo con el condicional *case*)**.** *Un número natural n es un sucesor si es de la forma Suc m.*

---

constdefs *es-sucesor* :: $nat \Rightarrow bool$
$es\text{-}sucesor\ n \equiv$
$(case\ n\ of$
$\quad 0 \quad \Rightarrow False$
$\mid Suc\ m \Rightarrow True)$

Lema 1.8.4 (Ejemplo de simplificación con el condicional *case*)**.** *El número 3 es sucesor.*

---

lemma *es-sucesor 3*
by (*simp add*:*es-sucesor-def*)

1.9 Tipos de datos y recursión primitiva

Definición 1.9.1 (Ejemplo de definición de tipo de dato recursivo)**.** *Una lista de elementos de tipo* `a` *es la lista* `Vacia` *o se obtiene añadiendo, con* `ConsLista`*, un elemento de tipo* `a` *a una lista de elementos de tipo* `a`*.*

---

datatype *'a Lista = Vacia | ConsLista 'a 'a Lista*

Definición 1.9.2 (Ejemplo de definición primitiva recursiva)**.** `conc xs ys` *e la concatenación de las lista* `xs` *e* `ys`*.*

---

consts *conc* :: *'a Lista ⇒ 'a Lista ⇒ 'a Lista*
primrec
conc Vacia ys = ys
conc (ConsLista x xs) ys = ConsLista x (conc xs ys)

Lema 1.9.3 (Ejemplo de simplificación con tipo de dato recursivo)**.** *La concatenación de la lista formada por 1 y 2 con la lista formada por el 3 es la lista cuyos elementos son 1,2 y 3.*

---

lemma *conc (ConsLista 1 (ConsLista 2 Vacia)) (ConsLista 3 Vacia) =*
(ConsLista 1 (ConsLista 2 (ConsLista 3 Vacia)))
by *simp*

Ejercicio 1.9.4 (Ejemplo de definición primitiva recursiva sobre los naturales)**.** Definir una función que sume los primeros n números naturales y usarla para comprobar que la suma de los 3 primeros números naturales es 6.

---

consts *suma* :: *nat ⇒ nat*
primrec
suma 0 = 0
suma (Suc m) = (Suc m) + suma m

lemma *suma 3 = 6*
by *(simp add:suma-nat-def)*

Capítulo 2

El lenguaje de demostración Isar

Este capítulo describe los elementos básicos del lenguaje de demostración Isar (*Intelligible semi–automated reasoning*).

2.1 Panorama de la sintaxis (simplificada) de Isar

Nota 2.1.1 (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**.

Nota 2.1.2 (Gramática (simplificada) de las demostraciones en 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.

2.2 Razonamiento proposicional

Nota 2.2.1 (Regla de introducción de la conjunción)**.**

$$(conjI)\ \frac{P \qquad Q}{P \wedge Q}$$

Lema 2.2.2 (Ejemplo de introducción de conjunción con razonamiento progresivo)**.** $P, Q \vdash P \wedge (Q \wedge P)$.

Demostración: Estamos suponiendo

$$P \tag{2.1}$$

y

$$Q \tag{2.2}$$

De 2.2 y 2.1, por introducción de la conjunción, se tiene

$$Q \wedge P \tag{2.3}$$

De 2.1 y 2.3, por introducción de la conjunción, se tiene

$$P \wedge (Q \wedge P)$$

□

lemma *conj2*:
assumes *p*: P **and** *q*: Q
shows $P \wedge (Q \wedge P)$
proof −
from *q p* **have** *qp*: $Q \wedge P$ **by** (*rule conjI*)
from *p qp* **show** $P \wedge (Q \wedge P)$ **by** (*rule conjI*)
qed

Nota 2.2.3 (Razonamiento progresivo y regresivo)**.**

- Isabelle soporta *razonamiento progresivo*. La anterior demostración es una muestra.
- Isabelle soporta *razonamiento regresivo*. La siguiente demostración es una muestra.

Lema 2.2.4 (Ejemplo de introducción de la conjunción con razonamiento regresivo)**.** $P, Q \vdash P \wedge (Q \wedge P)$.

Demostración: Estamos suponiendo

$$P \tag{2.4}$$

y

$$Q \tag{2.5}$$

Para demostrar el lema, por introducción de la conjunción, basta probar

$$P \tag{2.6}$$

y

$$Q \wedge P \tag{2.7}$$

La condición 2.6 se tiene por la hipótesis 2.4. Para demostrar la condición 2.7, por introducción de la conjunción, basta probar

$$Q \tag{2.8}$$

y

$$P \tag{2.9}$$

La condición 2.8 se tiene por la hipótesis 2.5 y la condición 2.9 se tiene por la hipótesis 2.4.

□

lemma
assumes *p*: P **and** *q*: Q

```
 shows P ∧ (Q ∧ P)
proof (rule conjI)
 from p show P by this
next
 show Q ∧ P
 proof (rule conjI)
  from q show Q by this
 next
  from p show P by this
 qed
qed
```

Nota 2.2.5 (El método *this*)**.** El método *this* demuestra el objetivo usando el hecho actual (es decir, el de la cláusula **from**).

Nota 2.2.6 (Reglas de eliminación de la conjunción)**.**

$$(conjunct1)\ \frac{P \wedge Q}{P} \qquad (conjunct2)\ \frac{P \wedge Q}{Q}$$

Nota 2.2.7 (Regla de introducción de la implicación)**.**

$$(impI)\ \frac{\begin{array}{c} P \\ \hline Q \end{array}}{P \longrightarrow Q}$$

Lema 2.2.8 (Ejemplo de razonamiento híbrido)**.** *Sean a y b dos números naturales. Si* $0 < a$ *y* $a < b$*, entonces* $a \times a < b \times b$*.*

```
lemma
 fixes a b :: nat
 shows 0 < a ∧ a < b ⟶ a * a < b * b
proof (rule impI)
 assume x: 0 < a ∧ a < b
 from x have za: 0 < a by (rule conjunct1)
 from x have ab: a < b by (rule conjunct2)
 from za ab have aa: a*a < a*b by simp
 from ab have bb: a*b < b*b by simp
 from aa bb show a*a < b*b by arith
qed
```

Nota 2.2.9 (Modus ponens)**.**

$$(mp)\ \frac{P \longrightarrow Q \qquad P}{Q}$$

Nota 2.2.10 (Reglas de introducción de la disyunción)**.**

$$(disjI1)\ \frac{P}{P \vee Q} \qquad (disjI2)\ \frac{Q}{P \vee Q}$$

Nota 2.2.11 (Regla de eliminación de la disyunción)**.**

$$(disjE)\ \frac{P \vee Q \qquad \begin{array}{c} P \\ \hline R \end{array} \qquad \begin{array}{c} Q \\ \hline R \end{array}}{R}$$

Lema 2.2.12 (Razonamiento por casos)**.** $A \vee B, A \to C, B \to C \vdash C$.

---

lemma
 assumes *ab*: $A \vee B$ **and** *ac*: $A \longrightarrow C$ **and** *bc*: $B \longrightarrow C$
 shows C
proof −
 note *ab*
 moreover {
  assume *a*: A
  from *ac a* **have** C **by** (*rule mp*) }
 moreover {
  assume *b*: B
  from *bc b* **have** C **by** (*rule mp*) }
 ultimately show C **by** (*rule disjE*)
qed

Nota 2.2.13 (Resumen de reglas proposicionales)**.**

TrueI	$True$
FalseE	$False \Longrightarrow P$
conjI	$[\![P; Q]\!] \Longrightarrow P \wedge Q$
conjunct1	$P \wedge Q \Longrightarrow Q$
conjE	$[\![P \wedge Q; [\![P; Q]\!] \Longrightarrow R]\!] \Longrightarrow R$
disjI1	$P \Longrightarrow P \vee Q$
disjI2	$Q \Longrightarrow P \vee Q$
disjE	$[\![P \vee Q; P \Longrightarrow R; Q \Longrightarrow R]\!] \Longrightarrow R$
notI	$(P \Longrightarrow False) \Longrightarrow \neg P$
notE	$[\![\neg P; P]\!] \Longrightarrow R$
impI	$(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$
impE	$[\![P \longrightarrow Q; P; Q \Longrightarrow R]\!] \Longrightarrow R$
mp	$[\![P \longrightarrow Q; P]\!] \Longrightarrow Q$
iff	$(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow P = Q$
iffI	$[\![P \Longrightarrow Q; Q \Longrightarrow P]\!] \Longrightarrow P = Q$
iffD1	$[\![Q = P; Q]\!] \Longrightarrow P$
iffD2	$[\![P = Q; Q]\!] \Longrightarrow P$
iffE	$[\![P = Q; [\![P \longrightarrow Q; Q \longrightarrow P]\!] \Longrightarrow R]\!] \Longrightarrow R$
ccontr	$(\neg P \Longrightarrow False) \Longrightarrow P$
classical	$(\neg P \Longrightarrow P) \Longrightarrow P$
excluded_middle	$\neg P \vee P$
disjCI	$(\neg Q \Longrightarrow P) \Longrightarrow P \vee Q$
impCE	$[\![P \longrightarrow Q; \neg P \Longrightarrow R; Q \Longrightarrow R]\!] \Longrightarrow R$
iffCE	$[\![P = Q; [\![P; Q]\!] \Longrightarrow R; [\![\neg P; \neg Q]\!] \Longrightarrow R]\!] \Longrightarrow R$
notnotD	$\neg \neg P \Longrightarrow P$
swap	$[\![\neg Pa; \neg P \Longrightarrow Pa]\!] \Longrightarrow P$

Nota 2.2.14 (Referencia de reglas de inferencia)**.** Más información sobre las reglas de inferencia se encuentra en la sección 2.2 de Isabelle's Logics: HOL.

2.3 Atajos de Isar

Nota 2.3.1 (Atajos de Isar)**.** 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 (donde Isabelle adivina la regla)

Nota 2.3.2 (Razonamiento acumulativo)**.** Una sucesión de hechos que se van a usar como premisa en una declaración puede agruparse usando **moreover** (además) y usarse en la declaración usando **ultimately** (finalmente).

Lema 2.3.3 (Ejemplo de uso de atajos y razonamiento acumulativo)**.** $A \wedge B \vdash B \wedge A$.

---

lemma $A \wedge B \longrightarrow B \wedge A$
proof (*rule impI*)
 assume *ab*: $A \wedge B$
 hence B **by** (*rule conjunct2*)
 moreover from *ab* **have** A **..**
 ultimately show $B \wedge A$ **by** (*rule conjI*)
qed

2.4 Cuantificadores universal y existencial

Nota 2.4.1 (Reglas del cuantificador universal)**.**

$$(allI)\ \frac{\bigwedge x.\ P\ x}{\forall\, x.\ P\ x} \qquad (allE)\ \frac{\forall\, x.\ P\ x \qquad \dfrac{P\ x}{R}}{R}$$

En la regla *allI* la nueva variable se introduce mediante la palabra **fix**.

Lema 2.4.2 (Ejemplo con cuantificadores universales)**.** $\forall x.P \longrightarrow Qx \vdash P \longrightarrow (\forall x.Qx)$

---

lemma
 assumes *a*: $\forall\ x.\ P \longrightarrow Q\ x$
 shows $P \longrightarrow (\forall\ x.\ Q\ x)$
proof (*rule impI*)
 assume *p*: P

```
 show ∀ x. Q x
 proof (rule allI)
  fix x
  from a have pq: P ⟶ Q x by (rule allE)
  from pq p show Q x by (rule mp)
 qed
qed
```

Nota 2.4.3 (Reglas del cuantificador existencial)**.**

$$(exI)\ \frac{P\ x}{\exists\, x.\ P\ x} \qquad (exE)\ \frac{\exists\, x.\ P\ x \qquad \bigwedge x.\ \dfrac{P\ x}{Q}}{Q}$$

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

Lema 2.4.4 (Ejemplo con cuantificador existencial y demostración progresiva)**.** $\exists x.P \wedge Q(x) \vdash P \wedge (\exists x.Q(x))$

```
lemma
 assumes e: ∃ x. P ∧ Q(x)
 shows P ∧ (∃ x. Q(x))
proof −
 from e obtain x where f: P ∧ Q(x) by (rule exE)
 from f have p: P by (rule conjunct1)
 from f have q: Q(x) by (rule conjunct2)
 from q have eq: ∃ x. Q(x) by (rule exI)
 from p eq show P ∧ (∃ x. Q(x)) by (rule conjI)
qed
```

Lema 2.4.5 (Ejemplo con cuantificador existencial y demostración progresiva automática)**.** $\exists x.P \wedge Q(x) \vdash P \wedge (\exists x.Q(x))$

```
lemma
 assumes e: ∃ x. P ∧ Q(x)
 shows P ∧ (∃ x. Q(x))
proof −
 from e obtain x where f: P ∧ Q(x) ..
 from f have p: P ..
 from f have q: Q(x) ..
 from q have eq: ∃ x. Q(x) ..
```

from *p eq* **show** $P \wedge (\exists\ x.\ Q(x))$ **..**
qed

Lema 2.4.6 (Ejemplo con cuantificador existencial y demostración regresiva)**.** $\exists x.P \wedge Q(x) \vdash P \wedge (\exists x.Q(x))$

---

lemma
assumes *e*: $\exists\ x.\ P \wedge Q(x)$
shows $P \wedge (\exists\ x.\ Q(x))$
proof (*rule conjI*)
show *P*
proof −
from *e* **obtain** *x* **where** *p*: $P \wedge Q(x)$ **by** (*rule exE*)
from *p* **show** *P* **by** (*rule conjunct1*)
qed
show $\exists\ y.\ Q(y)$
proof −
from *e* **obtain** *x* **where** *p*: $P \wedge Q(x)$ **by** (*rule exE*)
from *p* **have** *q*: $Q(x)$ **by** (*rule conjunct2*)
from *q* **show** $\exists\ y.\ Q(y)$ **by** (*rule exI*)
qed
qed

Definición 2.4.7 (Ejemplo de definición existencial)**.** *El número natural x divide al número natural y si existe un natural k tal que* $k \times x = y$*. Se representa por* $x \mid y$*.*

---

constdefs *divide* :: *nat* ⇒ *nat* ⇒ *bool* (- | - [*80,80*] *80*)
$x \mid y \equiv \exists\ k.\ k{*}x = y$

Nota 2.4.8 (Ejemplo de activación automática de regla de simplificación)**.** La definición de `divide` se añade a las reglas de simplificación.

---

declare *divide-def* [*simp*]

Lema 2.4.9 (Transitividad de la divisibilidad)**.** *Sean a, b y c números naturales. Si a es divisible por b y b es divisible por c, entonces a es divisible por c.*

---

lemma *divide-trans*:
fixes *a b c* :: *nat*
assumes *ab*: $a \mid b$ **and** *bc*: $b \mid c$
shows $a \mid c$
proof *simp*
from *ab* **obtain** *m* **where** *m*: $m{*}a = b$ **by** *auto*

```
  from bc obtain n where n: n*b = c by auto
  from m n have m*n*a = c by auto
  thus ∃ k. k*a = c by (rule exI)
qed
```

Nota 2.4.10 (Método *auto*)**.** En el lema anterior es la primera vez que se usa el método automático (**by** *auto*).

Lema 2.4.11 (CNS de divisibilidad)**.** *Sean a y b dos números naturales. Entonces a es divisible por b syss el resto de dividir a entre b es cero.*

```
lemma CNS-divisibilidad:
  (a | b) = (b mod a = 0)
by auto
```

2.5 Razonamiento ecuacional

Nota 2.5.1 (Elementos para el razonamiento ecuacional)**.** El razonamiento ecuacional se realiza de manera más concisa usando la combinación de **also** (además) y **finally** (finalmente).

Lema 2.5.2 (Ejemplo de razonamiento ecuacional)**.** *Si a = b, b = c y c = d, entonces a = d.*

```
lemma
  assumes 1: a = b and 2: b = c and 3: c = d
  shows a = d
proof −
  have a = b by (rule 1)
  also have ... = c by (rule 2)
  also have ... = d by (rule 3)
  finally show a = d .
qed
```

Nota 2.5.3 (Demostración automática con la maza)**.** El lema anterior puede demostrarse automáticamente con la maza (“sledgehammer”).

```
lemma
  assumes 1: a = b and 2: b = c and 3: c = d
  shows a = d
proof −
  show a=d by (metis 1 2 3)
```

qed

Capítulo 3

Distinción de casos e inducción

3.1 Razonamiento por distinción de casos

3.1.1 Distinción de casos booleanos

Ejemplo 3.1.1 (Demostración por distinción de casos booleanos)**.** $\neg A \vee A$

---

lemma $\neg A \vee A$
proof *cases*
 assume A **thus** *?thesis* **..**
next
 assume $\neg A$ **thus** *?thesis* **..**
qed

Nota 3.1.2 (El método *cases*)**.** El método *cases* es una abreviatura de *rule case-split-thm* donde *case-split-thm* es

$$[\![P \Longrightarrow Q; \neg\, P \Longrightarrow Q]\!] \Longrightarrow Q.$$

Ejemplo 3.1.3 (Redemostración por distinción de casos booleanos nominados)**.** $\neg A \vee A$

---

lemma $\neg A \vee A$
proof $(cases\ A)$
 case *True* **thus** *?thesis* **..**
next
 case *False* **thus** *?thesis* **..**
qed

Nota 3.1.4 (El método *cases* sobre una fórmula)**.**

1. El método *(cases F)* es una abreviatura de la aplicación de la regla
$$[\![F \Longrightarrow Q; \neg\, F \Longrightarrow Q]\!] \Longrightarrow Q.$$

2. **assume** *True* es una abreviatura de F.
3. **assume** *False* es una abreviatura de $\neg F$.
4. Ventajas de *cases* con nombre: reduce la escritura de la fórmula y es independiente del orden de los casos.

3.1.2 Distinción de casos sobre otros tipos de datos

Lema 3.1.5 (Distinción de casos sobre listas)**.** *La longitud del resto de una lista es la longitud de la lista menos 1.*

---

lemma *length*$(tl\ xs) = length\ xs - 1$
proof $(cases\ xs)$
case *Nil* **thus** *?thesis* **by** *simp*
next
case *Cons* **thus** *?thesis* **by** *simp*
qed

Nota 3.1.6 (Distinción de casos sobre listas)**.**

1. El método de distinción de casos se activa con (*cases xs*) donde *xs* es del tipo lista.
2. **case** *Nil* es una abreviatura de **assume** *Nil: xs =[].*
3. **case** *Cons* es una abreviatura de **fix** `? ??` **assume** `Cons: xs = ? # ??`, donde `?` y `??` son variables anónimas.

Lema 3.1.7 (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.*

---

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

Nota 3.1.8 (La función *drop*)**.** La función *drop* está definida en la teoría `List` de forma que *drop n xs* es la lista obtenida eliminando en *xs* los *n* primeros elementos. Su definición es la siguiente

$$\begin{array}{lcl} drop\ n\ [] & = & [] \\ drop\ n\ (x{\cdot}xs) & = & \mathsf{case}\ n\ \mathsf{of}\ 0 \Rightarrow x{\cdot}xs \mid Suc\ m \Rightarrow drop\ m\ xs \end{array}$$

3.2 Inducción matemática

Nota 3.2.1 (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.

$$\frac{P\ 0 \qquad \bigwedge n.\ \dfrac{P\ n}{P\ (Suc\ n)}}{P\ n}$$

Nota 3.2.2 (Ejemplo de demostración por inducción)**.** Usaremos el principio de inducción matemática para demostrar que

$$1 + 3 + \cdots + (2n - 1) = n^2$$

Definición 3.2.3 (Suma de los primeros impares)**.** *suma-impares n es la suma de los n primeros números impares.*

---

consts *suma-impares* :: *nat* ⇒ *nat*
primrec
suma-impares 0 = *0*
suma-impares (*Suc n*) = (*2* ∗ (*Suc n*) − *1*) + *suma-impares n*

Lema 3.2.4 (Ejemplo de suma de impares)**.** *La suma de los 3 primeros números impares es 9.*

---

lemma *suma-impares 3* = *9*
by (*simp add*:*suma-impares-nat-def*)

Lema 3.2.5 (Ejemplo de demostración por inducción matemática)**.** *La suma de los n primeros números impares es* n^2*.*

Nota 3.2.6. Demostración automática del lema 3.2.5.

---

lemma *suma-impares n* = *n* ∗ *n*
by (*induct n*) *simp-all*

Nota 3.2.7 (Los métodos *induct* y *simp_all*)**.** En la demostración **by** *(induct n) simp_all* se aplica inducción en *n* y los dos casos se prueban por simplificación.

Nota 3.2.8. Demostración por inducción y casos del lema 3.2.5.

---

lemma *suma-impares n* = *n* ∗ *n*
proof (*induct n*)
case *0* **show** *?case* **by** *simp*

next
case *Suc* **show** *?case* **by** *simp*
qed

Nota 3.2.9. Mediante *?case* se refiere a la fórmula a demostrar en el correspondiente caso de prueba por inducción.

Nota 3.2.10. Demostración con patrones del lema 3.2.5.

---

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

Nota 3.2.11 (Patrones)**.** Cualquier fórmula seguida de (**is** *patrón*) equipara el patrón con la fórmula.

Nota 3.2.12. Demostración con patrones y razonamiento ecuacional del lema 3.2.5.

---

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 $\ldots$ = (2 $*$ (*Suc* n) $-$ 1) $+$ $n * n$ **using** *HI* **by** *simp*
also have $\ldots$ = $n * n + 2 * n + 1$ **by** *simp*
finally show *?P*(*Suc* n) **by** *simp*
qed

Nota 3.2.13. Demostración por inducción y razonamiento ecuacional del lema 3.2.5.

---

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 $\ldots$ = (2 $*$ (*Suc* n) $-$ 1) $+$ $n * n$ **using** *HI* **by** *simp*
also have $\ldots$ = $n * n + 2 * n + 1$ **by** *simp*

finally show *suma-impares* $(Suc\ n) = (Suc\ n) * (Suc\ n)$ **by** *simp*
qed

Definición 3.2.14 (Números pares)**.** *Un número natural n es par si existe un natural m tal que* $n = m + m$.

---

constdefs *par* :: *nat* $\Rightarrow$ *bool*
par n $\equiv \exists\ m.\ n{=}m{+}m$

Lema 3.2.15 (Ejemplo de inducción y existenciales)**.** *Para todo número natural n, se verifica que* $n \times (n + 1)$ *es par.*

---

lemma
fixes *n* :: *nat*
shows *par* $(n{*}(n{+}1))$
proof $(induct\ n)$
show *par* $(0 * (0 + 1))$ **by** $(simp\ add{:}par\text{-}def)$
next
fix *n* **assume** *par* $(n{*}(n{+}1))$
then have $\exists\ m.\ n{*}(n{+}1) = m{+}m$ **by** $(simp\ add{:}par\text{-}def)$
then obtain *m* **where** *m*: $n{*}(n{+}1) = m{+}m$ **by** $(rule\ exE)$
then have $(Suc\ n){*}((Suc\ n){+}1) = (m{+}n{+}1){+}(m{+}n{+}1)$ **by** *auto*
then have $\exists\ m.\ (Suc\ n){*}((Suc\ n){+}1) = m{+}m$ **by** $(rule\ exI)$
then show *par* $((Suc\ n){*}((Suc\ n){+}1))$ **by** $(simp\ add{:}par\text{-}def)$
qed

3.3 Inducción estructural

Nota 3.3.1 (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

$$\frac{P\ [] \qquad \bigwedge a\ list.\ \dfrac{P\ list}{P\ (a{\cdot}list)}}{P\ list}$$

- 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.

Nota 3.3.2 (Concatenación de listas)**.** En la teoría `List.thy` está definida la concatenación de listas (que se representa por `@`) como sigue

```
primrec
  append_Nil:  "[]@ys = ys"
  append_Cons: "(x#xs)@ys = x#(xs@ys)"
```

Lema 3.3.3 (Ejemplo de inducción sobre listas)**.** *La concatenación de listas es asociativa.*

Nota 3.3.4. Demostración automática de 3.3.3.

---

lemma *conc-asociativa*: $xs \mathbin{@} (ys \mathbin{@} zs) = (xs \mathbin{@} ys) \mathbin{@} zs$
by (*induct xs*) *simp-all*

Nota 3.3.5. Demostración estructurada de 3.3.3.

---

lemma *conc-asociativa*: $xs \mathbin{@} (ys \mathbin{@} zs) = (xs \mathbin{@} ys) \mathbin{@} zs$
proof (*induct xs*)
 show $[] \mathbin{@} (ys \mathbin{@} zs) = ([] \mathbin{@} ys) \mathbin{@} zs$
 proof −
  have $[] \mathbin{@} (ys \mathbin{@} zs) = ys \mathbin{@} zs$ **by** *simp*
  also have $\ldots = ([] \mathbin{@} ys) \mathbin{@} zs$ **by** *simp*
  finally show *?thesis* **.**
 qed
next
 fix *x xs*
 assume *HI*: $xs \mathbin{@} (ys \mathbin{@} zs) = (xs \mathbin{@} ys) \mathbin{@} zs$
 show $(x\#xs) \mathbin{@} (ys \mathbin{@} zs) = ((x\#xs) \mathbin{@} ys) \mathbin{@} zs$
 proof −
  have $(x\#xs) \mathbin{@} (ys \mathbin{@} zs) = x\#(xs \mathbin{@} (ys \mathbin{@} zs))$ **by** *simp*
  also have $\ldots = x\#((xs \mathbin{@} ys) \mathbin{@} zs)$ **using** *HI* **by** *simp*
  also have $\ldots = (x\#(xs \mathbin{@} ys)) \mathbin{@} zs$ **by** *simp*
  also have $\ldots = ((x\#xs) \mathbin{@} ys) \mathbin{@} zs$ **by** *simp*
  finally show *?thesis* **.**
 qed
qed

Ejercicio 3.3.6 (Árboles binarios)**.** Definir un tipo de dato para los árboles binarios.

---

datatype *'a arbol* = *Hoja 'a* | *Nodo 'a 'a arbol 'a arbol*

Ejercicio 3.3.7 (Imagen especular)**.** Definir la función *espejo* que aplicada a un árbol devuelve su imagen especular.

---

consts *espejo* :: *'a arbol* ⇒ *'a arbol*
primrec
espejo (*Hoja a*) = (*Hoja a*)
espejo (*Nodo f x y*) = (*Nodo f* (*espejo y*) (*espejo x*))

Ejercicio 3.3.8 (La imagen especular es involutiva)**.** Demostrar que la función *espejo* es involutiva; es decir, para cualquier árbol *t*,

espejo (*espejo t*) = *t*.

Nota 3.3.9. Demostración automática de 3.3.8.

---

lemma *espejo-involutiva*: *espejo*(*espejo*(*t*)) = *t*
by (*induct t*) *auto*

Nota 3.3.10. Demostración estructurada de 3.3.8.

---

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 arbol* **assume** *h1*: *?P t1*
fix *t2* :: *'a arbol* **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

Ejercicio 3.3.11 (Aplanamiento de árboles)**.** Definir la función *aplana* que aplane los árboles recorriéndolos en orden infijo.

---

consts *aplana* :: *'a arbol* ⇒ *'a list*
primrec
aplana (*Hoja a*) = [*a*]
aplana (*Nodo x t1 t2*) = (*aplana t1*)@[*x*]@(*aplana t2*)

Ejercicio 3.3.12 (Aplanamiento de la imagen especular)**.** *aplana* (*espejo t*) = *rev* (*aplana t*).

Nota 3.3.13. Demostración automática de 3.3.12.

---

lemma *aplana*(*espejo t*) = *rev*(*aplana t*)
by (*induct t*) *auto*

Nota 3.3.14. Demostración estructurada de 3.3.12.

---

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 arbol* **assume** *h1*: *?P t1*
fix *t2* :: *'a arbol* **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

Capítulo 4

Patrones de demostración

4.1 Demostraciones por casos

Nota 4.1.1 (Regla de eliminación de la disyunción)**.**

$$(disjE)\ \frac{P \vee Q \qquad \begin{array}{c}[P]\\ \vdots\\ R\end{array} \qquad \begin{array}{c}[Q]\\ \vdots\\ R\end{array}}{R}$$

Lema 4.1.2 (Ejemplo de demostración por casos)**.** $P \vee Q \Longrightarrow Q \vee P$

---

lemma *disj-conmutativa*: $P \vee Q \Longrightarrow Q \vee P$
proof $-$
 assume $P \vee Q$
 then show $Q \vee P$
 proof $(rule\ disjE)$
  assume P
  then show *?thesis* **by** $(rule\ disjI2)$
 next
  assume Q
  then show *?thesis* **by** $(rule\ disjI1)$
 qed
qed

Nota 4.1.3. El lema anterior puede demostrarse automáticamente como se muestra a continuación.

---

lemma *disj-conmutativa*: $P \vee Q \Longrightarrow Q \vee P$
by *auto*

4.2 Negación

Nota 4.2.1 (Reglas de la negación)**.**

$$(notI)\ \frac{\dfrac{P}{False}}{\neg P} \qquad (notE)\ \frac{\neg P \qquad P}{R}$$

Lema 4.2.2 (Ejemplo de demostración con negaciones)**.** *Si* $x^2 + y = 13$ *e* $y \neq 4$*, entonces* $x \neq 3$*.*

---

lemma
 fixes x :: *nat*
 assumes *1*: $x * x + y = 13$
  and *2*: $y \neq 4$
 shows $x \neq 3$
proof (*rule notI*)
 assume $x = 3$
 with *1* **have** $y = 4$ **by** *simp*
 with *2* **show** *False* **by** (*rule notE*)
qed

Nota 4.2.3. El lema anterior puede demostrarse más automáticamente como se muestra a continuación.

---

lemma
 fixes x :: *nat*
 assumes *1*: $x * x + y = 13$
  and *2*: $y \neq 4$
 shows $x \neq 3$
proof (*rule notI*)
 assume $x = 3$
 with *1 2* **show** *False* **by** *auto*
qed

4.3 Contradicciones

Nota 4.3.1 (Regla de contradicción)**.**

$$(FalseE)\ \frac{False}{P}$$

Lema 4.3.2 (Ejemplo de uso de la regla de contradicción)**.** *Si* $1 = 2$*, entonces* $3 = 7$*.*

---

lemma $1 = (2::nat) \longrightarrow 3 = (7::nat)$
proof $(rule\ impI)$
 assume $1 = (2::nat)$
 hence *False* **by** *simp*
 thus $3 = (7::nat)$ **by** $(rule\ FalseE)$
qed

Lema 4.3.3 (Ejemplo de demostración por casos y contradicción)**.** $\neg P, (P \vee Q) \vdash Q$

---

lemma *disjCE*:
 assumes $\neg\ P$ **and** $(P \vee Q)$
 shows Q
using $\langle P \vee Q \rangle$
proof $(rule\ disjE)$
next
 assume P
 show Q **by** *contradiction*
next
 assume Q
 show Q **by** *assumption*
qed

4.4 Equivalencias

Nota 4.4.1 (Reglas de equivalencia)**.**

$$(iffI)\ \frac{\dfrac{P}{Q} \qquad \dfrac{Q}{P}}{P = Q} \qquad (iffD1)\ \frac{Q = P \qquad Q}{P} \qquad (iffD2)\ \frac{P = Q \qquad Q}{P}$$

Lema 4.4.2 (Ejemplo de introducción de equivalencia)**.** *La fórmula* $((R \longrightarrow C) \wedge (S \longrightarrow C))$ *es equivalente a* $((R \vee S) \longrightarrow C)$*.*

---

lemma $((R \longrightarrow C) \wedge (S \longrightarrow C)) = ((R \vee S) \longrightarrow C)$
proof $(rule\ iffI)$
 assume $((R \longrightarrow C) \wedge (S \longrightarrow C))$
 then show $((R \vee S) \longrightarrow C)$ **by** *blast*
next

assume $R \vee S \longrightarrow C$
then show $(R \longrightarrow C) \wedge (S \longrightarrow C)$ **by** *blast*
qed

Nota 4.4.3 (El método *blast*)**.** En la demostración anterior es la primera vez que se usa el método de razonamiento automático *blast*.

Nota 4.4.4. El lema anterior puede demostrarse automáticamente como se muestra a continuación.

---

lemma $((R \longrightarrow C) \wedge (S \longrightarrow C)) = ((R \vee S) \longrightarrow C)$
by *auto*

Lema 4.4.5 (Ejemplo de eliminación de equivalencia)**.**

1. $A \longleftrightarrow B, A \vdash B$
2. $A \longleftrightarrow B, B \vdash A$

---

lemma assumes $A = B$ **and** A **shows** B
by $(rule\ iffD1)$

lemma assumes $A = B$ **and** B **shows** A
by $(rule\ iffD2)$

Capítulo 5

Heurísticas para la inducción y recursion general

5.1 Heurísticas para la inducción

Definición 5.1.1 (Definición recursiva de inversa)**.** *inversa xs es la inversa de la lista xs.*

---

consts *inversa* :: *'a list* ⇒ *'a list*
primrec
inversa [] = []
inversa (*x*#*xs*) = (*inversa xs*) @ [*x*]

Definición 5.1.2 (Definición de inversa con acumuladores)**.** *inversaAc xs es la inversa de la lista xs calculada con acumuladores.*

---

consts *inversaAc* :: *'a list* ⇒ *'a list* ⇒ *'a list*
primrec
inversaAc [] *ys* = *ys*
inversaAc (*x*#*xs*) *ys* = *inversaAc xs* (*x*#*ys*)

Lema 5.1.3 (Ejemplo de equivalencia entre las definiciones)**.** *La inversa de [1,2,3] es lo mismo calculada con la primera definición que con la segunda iniciando el acumulador con la lista vacía.*

---

lemma *inversa* [*1*,*2*,*3*] = *inversaAc* [*1*,*2*,*3*] []
by *simp*

Nota 5.1.4 (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*
next
fix *a xs* **assume** *HI*: *inversaAc xs* [] = *inversa xs*
have *inversaAc* (*a*#*xs*) [] = *inversaAc xs* [*a*] **by** *simp*
— Problema: la hipótesis de inducción no es aplicable.
show *inversaAc* (*a*#*xs*) [] = *inversa* (*a*#*xs*)
oops

Nota 5.1.5 (Heurística de generalización)**.** Cuando se use demostración estructural, cuantificar universalmente las variables libres (o, equivalentemente, considerar las variables libres como variables arbitrarias).

Lema 5.1.6 (Lema con generalización)**.** *Para toda lista ys se tiene*
inversaAc xs ys = inversa xs @ ys.

---

lemma *inversaAc xs ys* = (*inversa xs*) @ *ys*
proof (*induct xs arbitrary*: *ys*)
show ⋀*ys*. *inversaAc* [] *ys* = (*inversa* []) @ *ys* **by** *simp*
next
fix *a xs* **assume** *HI*: ⋀*ys*. *inversaAc xs ys* = *inversa xs* @ *ys*
show ⋀*ys*. *inversaAc* (*a*#*xs*) *ys* = *inversa* (*a*#*xs*) @ *ys*
proof −
fix *ys*
have *inversaAc* (*a*#*xs*) *ys* = *inversaAc 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 *inversaAc* (*a* # *xs*) *ys* = *inversa* (*a* # *xs*) @ *ys* **by** *simp*
qed
qed

Nota 5.1.7. 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 `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

5.2 Recursión general. Corrección del algoritmo de Euclides

El objetivo de esta sección es mostrar el uso de las definiciones recursivas generales y sus esquemas de inducción. Además, como caso de estudio demostraremos la corrección del algoritmo de Euclides.

Definición 5.2.1 (Máximo común divisor)**.** *El número k es el máximo común divisor de m y n si k divide a m y a n y cualquier divisor común de m y n divide a k.*

constdefs *esMCD* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* (- *es mcd de* - *y* - [*40,40,40*] *39*)
k es mcd de m y n ≡ (*k*|*m* ∧ *k*|*n* ∧ (∀ *q*. *q*|*m* ∧ *q*|*n* ⟶ *q*|*k*))

Definición 5.2.2 (Ejemplo de definición recursiva general)**.** *La función calcula_mcd toma como argumento un par de números naturales y devuelve su máximo común divisor calculado mediante el algoritmo de Euclides.*

consts *calcula-mcd* :: *nat* × *nat* ⇒ *nat*
recdef *calcula-mcd measure*(λ (*m*,*n*). *n*)
calcula-mcd(*m*, *n*) = (*if n* = *0 then m else calcula-mcd*(*n*, *m mod n*))

Nota 5.2.3 (Definiciones recursivas generales)**.**

- Las definiciones recursivas generales se identifican mediante **recdef**.
- Para demostrar la terminación se proporciona una medida.
- La función de medida en el ejemplo anterior es *measure*(λ (*m*,*n*). *n*).
- Se prueba que en las llamadas de *calcula_mcd* el segundo argumento decrece.
- Al definir una función recursiva general se genera una regla de inducción. En la definición anterior, la regla generada es

$$(\mathit{calcula\_mcd.induct})\quad \frac{\bigwedge m\ n.\ \dfrac{n \neq 0 \longrightarrow P\ n\ (m\ mod\ n)}{P\ m\ n}}{P\ u\ v}$$

La usaremos en la demostración del teorema de corrección del algoritmo de Euclides (5.2.10).

Lema 5.2.4 (Ejemplo de cálculo)**.** *El máximo común divisor de 10 y 15 es 5.*

---

lemma *calcula-mcd*$(10,15) = 5$
by *simp*

Lema 5.2.5. *Si k divide a m y a n, entonces divide a* $m + n$.

---

lemma *divide-suma*:
fixes $k\ m\ n$ :: *nat*
assumes
1: $k|m$ **and**
2: $k|n$
shows $k|(m+n)$
proof −
from *1* **have** $\exists\ q.\ k*q = m$ **by** *auto*
then obtain q **where** *3*: $k*q = m$ **by** (*rule exE*)
from *2* **have** $\exists\ r.\ k*r = n$ **by** *auto*
then obtain r **where** *4*: $k*r = n$ **by** (*rule exE*)
hence $m+n = k*q+k*r$ **using** *3 4* **by** *simp*
also have $\ldots = k*(q+r)$ **by** (*rule add-mult-distrib2*[*symmetric*])
finally have $m+n = k*(q+r)$ **.**
hence $\exists\ p.\ m+n=k*p$ **by** (*rule exI*)
thus $k|(m+n)$ **by** *auto*
qed

Nota 5.2.6 (Comentarios a la demostración anterior)**.**

- El lema *add_mult_distrib2* es $k * (m + n) = k * m + k * n$
- Un lema ecuacional con el atributo *symmetric* se transforma en el simétrico; es decir, si el lema original es $t_1 = t_2$, el transformado es $t_2 = t_1$.

Lema 5.2.7. *Si k divide a m y a n, entonces divide a* $m - n$.

---

lemma *divide-resta*:
fixes $k\ m\ n$ :: *nat*
assumes
1: $k|m$ **and**
2: $k|n$

shows $k|(m-n)$
proof $-$
from *1* **have** $\exists\ q.\ k*q = m$ **by** *auto*
then obtain q **where** *3*: $k*q = m$ **by** (*rule exE*)
from *2* **have** $\exists\ r.\ k*r = n$ **by** *auto*
then obtain r **where** *4*: $k*r = n$ **by** (*rule exE*)
hence $m-n = k*q-k*r$ **using** *3 4* **by** *simp*
also have $\ldots = k*(q-r)$ **by** (*rule diff-mult-distrib2*[*symmetric*])
finally have $m-n = k*(q-r)$ **.**
hence $\exists\ p.\ m-n=k*p$ **by** (*rule exI*)
thus $k|(m-n)$ **by** *auto*
qed

Nota 5.2.8 (Comentarios a la demostración anterior)**.**

- La demostración es análoga a la del lema anterior.
- El lema *diff_mult_distrib2* es $k * (m - n) = k * m - k * n$

Lema 5.2.9. *Si m y r son congruentes módulo n (es decir, m es de la forma $q \times n + r$), entonces el máximo común divisor de m y n es igual al máximo común divisor de m y r.*

---

lemma *invariante*:
assumes *M*: $m = q*n + r$
shows (*x es mcd de m y n*) = (*x es mcd de n y r*)
proof $-$
{ **fix** k **assume** $k|m$ **and** $k|n$
hence $k|(m - q*n)$ **using** *divide-resta* **by** *auto*
hence $k|r$ **using** *M* **by** *simp* }
moreover
{ **fix** k **assume** $k|n$ **and** $k|r$
hence $k|(q*n + r)$ **using** *divide-suma* **by** *auto*
hence $k|m$ **using** *M* **by** *simp* }
ultimately show *?thesis* **using** *M* **by** (*simp add*:*esMCD-def*, *blast*)
qed

Teorema 5.2.10 (Corrección del algoritmo de Euclides)**.** *La función calcula_mcd devuelve el máximo común divisor de sus argumentos.*

---

theorem *calcula-mcd-calcula-mcd*:
calcula-mcd(*m*,*n*) *es mcd de m y n*
proof (*induct rule*: *calcula-mcd.induct*)
fix *m n*

assume *HI*: $n \neq 0 \longrightarrow$ *calcula-mcd*$(n, m \bmod n)$ *es mcd de n y* $(m \bmod n)$
show *calcula-mcd*(m,n) *es mcd de m y n*
proof $(case\text{-}tac\ n = 0)$
assume $n = 0$
thus *?thesis* **using** *esMCD-def* **by** *simp*
next
assume N: $n \neq 0$
have $m = (m\ div\ n)*n + (m\ mod\ n)$ **by** *auto*
with N *HI invariante* **have** *calcula-mcd*$(n, m \bmod n)$ *es mcd de m y n* **by** *blast*
with N **show** *?thesis* **by** *simp*
qed
qed

Nota 5.2.11 (Inducción sobre recursión)**.** El formato para iniciar una demostración por inducción en la regla inductiva correspondiente a la definición recursiva de la función f es

proof (*induct rule:f.induct*)

5.3 Recursión mutua e inducción

Nota 5.3.1 (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*

Nota 5.3.2 (Regla de inducción correspondiente a la recursión cruzada)**.** La regla de inducción sobre árboles y bosques es (*arbol_bosque.induct*)

$$\frac{\begin{array}{cc} P1\ Hoja & \bigwedge a\ bosque.\ \dfrac{P2\ bosque}{P1\ (Nodo\ a\ bosque)} \\ P2\ Vacio & \bigwedge arbol\ bosque.\ \dfrac{P1\ arbol \qquad P2\ bosque}{P2\ (ConsB\ arbol\ bosque)} \end{array}}{P1\ arbol \wedge P2\ bosque}$$

Nota 5.3.3 (Ejemplos de definición por recursión cruzada)**.**

1. *(aplana_arbol a)* es la lista obtenida aplanando el árbol a.
2. *(aplana_bosque b)* es la lista obtenida aplanando el bosque b.
3. *(map_arbol a h)* es el árbol obtenido aplicando la función h a todos los nodos del árbol a.
4. *(map_bosque b h)* es el bosque obtenido aplicando la función h a todos los nodos del bosque b.

---

consts
aplana-arbol :: *'a arbol* ⇒ *'a list*
aplana-bosque :: *'a bosque* ⇒ *'a list*
primrec
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*)

consts
map-arbol :: *'a arbol* ⇒ (*'a* ⇒ *'b*) ⇒ *'b arbol*
map-bosque :: *'a bosque* ⇒ (*'a* ⇒ *'b*) ⇒ *'b bosque*
primrec
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*)

Lema 5.3.4 (Ejemplo de inducción cruzada)**.**

1. *aplana-arbol* (*map-arbol a h*) = *map h* (*aplana-arbol a*)
2. *aplana-bosque* (*map-bosque b h*) = *map h* (*aplana-bosque b*)

---

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
```

Capítulo 6

Caso de estudio: Compilación de expresiones

El objetivo de esta sección es contruir un compilador de expresiones genéricas (construidas con variables, constantes y operaciones binarias) a una máquina de pila y demostrar su corrección.

6.1 Las expresiones y el intérprete

Definición 6.1.1. *Las expresiones son las constantes, las variables (representadas por números naturales) y las aplicaciones de operadores binarios a dos expresiones.*

---

types $'v\ binop = 'v \Rightarrow 'v \Rightarrow 'v$
datatype $'v\ expr =$
$\quad Const\ 'v$
$\mid Var\ nat$
$\mid App\ 'v\ binop\ 'v\ expr\ 'v\ expr$

Definición 6.1.2 (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.*

---

consts $valor :: 'v\ expr \Rightarrow (nat \Rightarrow 'v) \Rightarrow 'v$
primrec
$valor\ (Const\ b)\ ent = b$
$valor\ (Var\ x)\ ent = ent\ x$
$valor\ (App\ f\ e1\ e2)\ ent = (f\ (valor\ e1\ ent)\ (valor\ e2\ ent))$

Ejemplo 6.1.3. A continuación mostramos algunos ejemplos de evaluación con el intérprete.

lemma
valor (*Const 3*) *id* = *3* ∧
valor (*Var 2*) *id* = *2* ∧
valor (*Var 2*) (λ*x*. *x*+*1*) = *3* ∧
valor (*App* (*op* +) (*Const 3*) (*Var 2*)) (λ*x*. *x*+*1*) = *6* ∧
valor (*App* (*op* +) (*Const 3*) (*Var 2*)) (λ*x*. *x*+*4*) = *9*
by *simp*

6.2 La máquina de pila

Nota 6.2.1. 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*

Definición 6.2.2 (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.*

consts *ejec* :: *'v instr list* ⇒ (*nat*⇒*'v*) ⇒ *'v list* ⇒ *'v list*
primrec
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*))))

Ejemplo 6.2.3. 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*] (λ*x*. *x*+*4*) [*7*] = [*3*,*6*,*7*] ∧
ejec [*ILoad 2*, *IConst 3*, *IApp* (*op* +)] (λ*x*. *x*+*4*) [*7*] = [*9*,*7*]
by *simp*

6.3 El compilador

Definición 6.3.1. *El compilador* comp *traduce una expresión en una lista de instrucciones.*

consts *comp* :: *'v expr* ⇒ *'v instr list*
primrec
comp (*Const v*) = [*IConst v*]
comp (*Var x*) = [*ILoad x*]
comp (*App f e1 e2*) = (*comp e2*) @ (*comp e1*) @ [*IApp f*]

Ejemplo 6.3.2. 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*

6.4 Corrección del compilador

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

El teorema anterior no puede demostrarse por inducción en *e*. Para demostrarlo por inducción, lo generalizamos a

theorem ∀ *vs*. *ejec* (*comp e*) *ent vs* = (*valor e ent*)#*vs*
oops

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 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
```

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
```

La demostración del teorema es la siguiente

```
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
```

Capítulo 7

Conjuntos, funciones y relaciones

No busquéis el significado, buscad el uso.
L. WITTGENSTEIN

7.1 Conjuntos

7.1.1 Operaciones con conjuntos

Nota 7.1.1. La teoría elemental de conjuntos es `HOL/Set.thy`.

Nota 7.1.2. 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*).

Nota 7.1.3 (Reglas de la intersección)**.**

- $[\![c \in A; c \in B]\!] \Longrightarrow c \in A \cap B$ *(IntI)*
- $c \in A \cap B \Longrightarrow c \in A$ *(IntD1)*
- $c \in A \cap B \Longrightarrow c \in B$ *(IntD2)*

Nota 7.1.4. Propiedades del complementario:

- $(c \in -A) = (c \notin A)$ (*Compl_iff*)
- $-(A \cup B) = -A \cap -B$ (*Compl_Un*)

Nota 7.1.5. El conjunto **vacío** se representa por $\{\}$ y el **universal** por *UNIV*.

Nota 7.1.6. Propiedades de la **diferencia** y del complementario:

- $A \cap (B - A) = \emptyset$ (*Diff_disjoint*)

- $A \cup -A = UNIV$ (*Compl_partition*)

Nota 7.1.7. Reglas de la relación de **subconjunto**:

- $(\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$ (*subsetI*)
- $[\![A \subseteq B; c \in A]\!] \Longrightarrow c \in B$ (*subsetD*)

Nota 7.1.8. Ejemplo trivial.

lemma $(A \cup B \subseteq C) = (A \subseteq C \wedge B \subseteq C)$
by *blast*

Nota 7.1.9. Otro ejemplo trivial.

lemma $(A \subseteq -B) = (B \subseteq -A)$
by *blast*

Nota 7.1.10. Principio de extensionalidad de conjuntos:

- $(\bigwedge x.\ (x \in A) = (x \in B)) \Longrightarrow A = B$ (*set_ext*)

Nota 7.1.11. Reglas de la **igualdad** de conjuntos:

- $[\![A \subseteq B; B \subseteq A]\!] \Longrightarrow A = B$ (*equalityI*)
- $[\![A = B; [\![A \subseteq B; B \subseteq A]\!] \Longrightarrow P]\!] \Longrightarrow P$ (*equalityE*)

Lema 7.1.12 (Analogía entre intersección y conjunción)**.** *$x \in A \cap B$ syss $x \in A$ y $x \in B$.*

lemma $(x \in A \cap B) = (x \in A \wedge x \in B)$
by *simp*

Lema 7.1.13 (Analogía entre unión y disyunción)**.** *$x \in A \cup B$ syss $x \in A$ ó $x \in B$.*

lemma $(x \in A \cup B) = (x \in A \vee x \in B)$
by *simp*

Lema 7.1.14 (Analogía entre subconjunto e implicación)**.** *$A \subseteq B$ syss para todo x, si $x \in A$ entonces $x \in B$.*

lemma $(A \subseteq B) = (\forall\ x.\ x \in A \longrightarrow x \in B)$
by *auto*

Lema 7.1.15 (Analogía entre complementario y negación)**.** *x pertenece al complementario de A syss x no pertenece a A.*

lemma $(x \in -A) = (x \notin A)$
by *simp*

7.1.2 Notación de conjuntos finitos

Nota 7.1.16. La teoría de conjuntos finitos es `HOL/Finite_Set.thy`.

Nota 7.1.17. Los conjuntos finitos se definen por inducción a partir de las siguientes reglas inductivas:

- El conjunto vacío es un conjunto finito.
  $\mathit{finite}\ \{\}$ (*emptyI*)
- Si se le añade un elemento a un conjunto finito se obtiene otro conjunto finito.
  $\mathit{finite}\ A \Longrightarrow \mathit{finite}\ (\mathit{insert}\ a\ A)$ (*insertI*)

Nota 7.1.18. En la notación matemática, las reglas anteriores se representan como sigue:

- El conjunto vacío es un conjunto finito.
  $\mathit{finite}\ \emptyset$ (*emptyI*)
- Si se le añade un elemento a un conjunto finito se obtiene otro conjunto finito.
  $\mathit{finite}\ A \Longrightarrow \mathit{finite}\ (\{a\} \cup A)$ (*insertI*)

Ejemplo 7.1.19. Ejemplos de conjuntos finitos.

---

lemma
$\mathit{insert}\ 2\ \{\} = \{2\} \wedge$
$\mathit{insert}\ 3\ \{2\} = \{2,3\} \wedge$
$\mathit{insert}\ 2\ \{2,3\} = \{2,3\} \wedge$
$\{2,3\} = \{3,2,3,2,2\}$
by *auto*

Nota 7.1.20. Los conjuntos finitos se representan con la notación conjuntista habitual: los elementos entre llaves y separados por comas.

Nota 7.1.21. Ejemplo trivial.

lemma $\{a,b\} \cup \{c,d\} = \{a,b,c,d\}$
by *blast*

Nota 7.1.22. Conjetura falsa.

lemma $\{a,b\} \cap \{b,c\} = \{b\}$
refute
oops

Nota 7.1.23. Conjetura corregida.

lemma $\{a,b\} \cap \{b,c\} = (\textit{if } a{=}c \textit{ then } \{a,b\} \textit{ else } \{b\})$
by *auto*

Nota 7.1.24 (Definiciones recursivas sobre conjuntos finitos)**.** Se pueden definir funciones recursivas sobre conjuntos finitos usando la función de plegado *fold*.

Definición 7.1.25 (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.*

constdefs *sumaConj* :: *nat set* ⇒ *nat*
sumaConj S ≡ *fold* (λ *x y*. *x* + *y*) (λ *x*. *x*) *0 S*

constdefs *productoConj* :: *nat set* ⇒ *nat*
productoConj S ≡ *fold* (λ *x y*. *x* ∗ *y*) (λ *x*. *x*) *0 S*

constdefs *sumaCuadradosConj* :: *nat set* ⇒ *nat*
sumaCuadradosConj S ≡ *fold* (λ *x y*. *x* + *y*) (λ *x*. *x*∗*x*) *0 S*

Nota 7.1.26. 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*]

Ejemplo 7.1.27. Ejemplos de evaluación de las anteriores definiciones recursivas.

lemma

sumaConj {1,2,3,4} = 10 ∧
sumaCuadradosConj {1,2,3,4} = 30 ∧
productoConj {1,2,3} = *productoConj* {3,2}
by *simp*

Nota 7.1.28 (Inducción sobre conjuntos finitos)**.** Para demostrar que todos los conjuntos finitos tienen una propiedad P basta probar que

1. El conjunto vacío tiene la propiedad P.
2. 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

$$(\textit{finite\_induct}) \qquad \frac{\textit{finite } F \qquad P\ \emptyset \qquad \bigwedge x\ F.\ \dfrac{\textit{finite } F \qquad x \notin F \qquad P\ F}{P\ (\{x\} \cup F)}}{P\ F}$$

Lema 7.1.29 (Ejemplo de inducción sobre conjuntos finitos)**.** *Sea S un conjunto finito de números naturales. Entonces todos los elementos de S son menores o iguales que la suma de los elementos de S.*

Demostración automática:

lemma *finite S* $\Longrightarrow \forall\, x{\in}S.\ x \leq$ *sumaConj S*
by (*induct rule*: *finite-induct*) *auto*

Demostración estructurada:

lemma *sumaConj-acota*: *finite S* $\Longrightarrow \forall\, x{\in}S.\ x \leq$ *sumaConj S*
proof (*induct rule*: *finite-induct*)
show $\forall\, x{\in}\{\}.\ x \leq$ *sumaConj* $\{\}$ **by** *simp*
next
fix *x* **and** *F*
assume *fF*: *finite F*
and *xF*: $x \notin F$
and *HI*: $\forall\ x{\in}F.\ x \leq$ *sumaConj F*
show $\forall\, y{\in}$*insert x F*. $y \leq$ *sumaConj* (*insert x F*)
proof
fix *y*
assume $y \in$ *insert x F*
show $y \leq$ *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
```

7.1.3 Definiciones por comprensión

Nota 7.1.30. El conjunto de los elementos que cumple la propiedad P se representa por $\{x.\ P\}$.

Nota 7.1.31. Reglas de comprensión (relación entre colección y pertenencia):

- $(a \in \{x \mid P\ x\}) = P\ a$ *(mem_Collect_eq)*
- $\{x \mid x \in A\} = A$ *(Collect_mem_eq)*

Nota 7.1.32. Dos ejemplos triviales.

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

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

Nota 7.1.33. Ejemplo con la sintaxis general de comprehensión.

```
lemma
  {p*q | p q. p∈prime ∧ q∈prime} =
  {z. ∃p q. z = p*q ∧ p∈prime ∧ q∈prime}
by blast
```

Nota 7.1.34. En HOL, la notación conjuntista es azúcar sintáctica:

- $x{\in}A$ es equivalente a $A(x)$.
- $\{x.\ P\}$ es equivalente a $\lambda x.\ P$.

Definición 7.1.35 (Ejemplo de definición por comprensión)**.** *El conjunto de los pares es el de los números n para los que existe un m tal que* $n = 2 * m$.

constdefs *Pares* :: *nat set*
Pares $\equiv \{\ n.\ \exists\ m.\ n = 2{*}m\ \}$

Ejemplo 7.1.36. Los números 2 y 34 son pares.

lemma
$2 \in Pares \wedge$
$34 \in Pares$
by (*simp add*: *Pares-def*)

Definición 7.1.37. *El conjunto de los impares es el de los números n para los que existe un m tal que* $n = 2 * m + 1$.

constdefs *Impares* :: *nat set*
Impares $\equiv \{\ n.\ \exists\ m.\ n = 2{*}m + 1\ \}$

Lema 7.1.38 (Ejemplo con las reglas de intersección y comprensión)**.** *El conjunto de los pares es disjunto con el de los impares.*

lemma $x \notin (Pares \cap Impares)$
proof
fix x **assume** S: $x \in (Pares \cap Impares)$
hence $x \in Pares$ **by** (*rule IntD1*)
hence $\exists\ m.\ x = 2 * m$ **by** (*simp only*: *Pares-def mem-Collect-eq*)
then obtain p **where** p: $x = 2 * p$ **..**

from S **have** $x \in Impares$ **by** (*rule IntD2*)
hence $\exists\ m.\ x = 2 * m + 1$ **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

7.1.4 Cuantificadores acotados

Nota 7.1.39. Reglas de **cuantificador universal acotado** ("bounded"):

- $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \Longrightarrow \forall\, x{\in}A.\ P\ x$ (*ballI*)
- $[\![\forall\, x{\in}A.\ P\ x; x \in A]\!] \Longrightarrow P\ x$ (*bspec*)

Nota 7.1.40. Reglas de **cuantificador existencial acotado** ("bounded"):

- $[\![P\ x; x \in A]\!] \Longrightarrow \exists\, x{\in}A.\ P\ x$ (*bexI*)
- $[\![\exists\, x{\in}A.\ P\ x; \bigwedge x.\ [\![x \in A; P\ x]\!] \Longrightarrow Q]\!] \Longrightarrow Q$ (*bexE*)

Nota 7.1.41. Reglas de la **unión indexada**:

- $(b \in (\bigcup_{x\in A} B\ x)) = (\exists\, x{\in}A.\ b \in B\ x)$ (*UN_iff*)
- $[\![a \in A; b \in B\ a]\!] \Longrightarrow b \in (\bigcup_{x\in A} B\ x)$ (*UN_I*)
- $[\![b \in (\bigcup_{x\in A} B\ x); \bigwedge x.\ [\![x \in A; b \in B\ x]\!] \Longrightarrow R]\!] \Longrightarrow R$ (*UN_E*)

Nota 7.1.42. Reglas de la **unión de una familia**:

- $\bigcup S \equiv \bigcup_{x\in S} x$ (*Union_def*)
- $(A \in \bigcup C) = (\exists\, X{\in}C.\ A \in X)$ (*Union_iff*)

Nota 7.1.43. Reglas de la **intersección indexada**:

- $(b \in (\bigcap_{x\in A} B\ x)) = (\forall\, x{\in}A.\ b \in B\ x)$ (*INT_iff*)
- $(\bigwedge x.\ x \in A \Longrightarrow b \in B\ x) \Longrightarrow b \in (\bigcap_{x\in A} B\ x)$ (*INT_I*)
- $[\![b \in (\bigcap_{x\in A} B\ x); b \in B\ a \Longrightarrow R; a \notin A \Longrightarrow R]\!] \Longrightarrow R$ (*INT_E*)

Nota 7.1.44. Reglas de la **intersección de una familia**:

- $\bigcap S \equiv \bigcap_{x\in S} x$ (*Inter_def*)
- $(A \in \bigcap C) = (\forall\, X{\in}C.\ A \in X)$ (*Inter_iff*)

Nota 7.1.45. Abreviaturas:

- *Collect P* es lo mismo que $\{x.\ P\}$.
- *All P* es lo mismo que $\forall\, x.\ P\ x$.
- *Ex P* es lo mismo que $\exists\, x.\ P\ x$.
- *Ball P* es lo mismo que $\forall\, x{\in}A.\ P\ x$.
- *Bex P* es lo mismo que $\exists\, x{\in}A.\ P\ x$.

7.1.5 Conjuntos finitos y cardinalidad

Nota 7.1.46. El número de elementos de un conjunto finito A es el cardinal de A y se representa por *card A*.

Ejemplo 7.1.47. Ejemplos de cardinales de conjuntos finitos.

lemma
$card\ \{\} = 0 \wedge$
$card\ \{4\} = 1 \wedge$
$card\ \{4,1\} = 2 \wedge$
$x \neq y \Longrightarrow card\ \{x,y\} = 2$
by *simp*

Nota 7.1.48. Propiedades de cardinales:

- Cardinal de la unión de conjuntos finitos:
  $[\![finite\ A; finite\ B]\!] \Longrightarrow card\ A + card\ B = card\ (A \cup B) + card\ (A \cap B)$ (*card_Un_Int*)
- Cardinal del conjunto potencia:
  $finite\ A \Longrightarrow card\ (Pow\ A) = Suc\ (Suc\ 0)\ \hat{}\ card\ A$ (*card_Pow*)

7.2 Funciones

La teoría de funciones es *HOL/Fun.thy*.

7.2.1 Nociones básicas de funciones

Nota 7.2.1. Principio de **extensionalidad para funciones**:

- $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow f = g$ (*ext*)

Nota 7.2.2. Actualización de funciones

- $(f(x := y))\ z = (\mathsf{if}\ z = x\ \mathsf{then}\ y\ \mathsf{else}\ f\ z)$ (*fun_upd_apply*)
- $f(x := y, x := z) = f(x := z)$ (*fun_upd_upd*)

Nota 7.2.3. Función identidad

- $id = (\lambda x.\ x)$ (*id_def*)

Nota 7.2.4. Composición de funciones:

- $f \circ g = (\lambda x.\, f\ (g\ x))$ (*o_def*)

Nota 7.2.5. Asociatividad de la composición:

- $f \circ (g \circ h) = f \circ g \circ h$ (*o_assoc*)

7.2.2 Funciones inyectivas, suprayectivas y biyectivas

Nota 7.2.6. Función **inyectiva** sobre A:

- $\textit{inj-on}\ f\ A \equiv \forall\, x{\in}A.\ \forall\, y{\in}A.\ f\ x = f\ y \longrightarrow x = y$ (*inj_on_def*)

Nota 7.2.7. *inj f* es una abreviatura de *inj-on f UNIV*.

Nota 7.2.8. Función **suprayectiva**:

- $\textit{surj}\ f \equiv \forall\, y.\ \exists\, x.\ y = f\ x$ (*surj_def*)

Nota 7.2.9. Función **biyectiva**:

- $\textit{bij}\ f \equiv \textit{inj}\ f \wedge \textit{surj}\ f$ (*bij_def*)

Nota 7.2.10. Propiedades de las **funciones inversas**:

- $\textit{inj}\ f \Longrightarrow \textit{inv}\ f\ (f\ x) = x$ (*inv_f_f*)
- $\textit{surj}\ f \Longrightarrow f\ (\textit{inv}\ f\ y) = y$ (*surj_f_inv_f*)
- $\textit{bij}\ f \Longrightarrow \textit{inv}\ (\textit{inv}\ f) = f$ (*inv_inv_eq*)

Nota 7.2.11. Igualdad de funciones (por extensionalidad):

- $(f = g) = (\forall\, x.\ f\ x = g\ x)$ (*expand_fun_eq*)

Lema 7.2.12. *Una función inyectiva puede cancelarse en el lado izquierdo de la composición de funciones.*

lemma
 assumes *inj f*
 shows $(f \circ g = f \circ h) = (g = h)$
proof
 assume $f \circ g = f \circ h$
 thus $g = h$ **using** ⟨*inj f*⟩ **by** (*simp add*:*expand-fun-eq inj-on-def*)

next
assume $g = h$
thus $f \circ g = f \circ h$ **by** *auto*
qed

Una demostración más detallada es la siguiente

lemma
assumes *inj f*
shows $(f \circ g = f \circ h) = (g = h)$
proof
assume $f \circ g = f \circ h$
show $g = h$
proof
fix x
have $(f \circ g)(x) = (f \circ h)(x)$ **using** ⟨$f \circ g = f \circ h$⟩ **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 $f \circ g = f \circ h$
proof
fix x
have $(f \circ g)\ x = f(g(x))$ **by** *simp*
also have $\ldots = f(h(x))$ **using** ⟨$g = h$⟩ **by** *simp*
also have $\ldots = (f \circ h)\ x$ **by** *simp*
finally show $(f \circ g)\ x = (f \circ h)\ x$ **by** *simp*
qed
qed

Una demostración más automática es la siguiente

lemma
assumes *inj f*
shows $(f \circ g = f \circ h) = (g = h)$
by (*metis Un-UNIV-left assms id-o inj-iff inj-on-Un o-assoc*)

El desarrollo de la demostración automática es la siguiente

lemma
assumes *inj f*
shows $(f \circ g = f \circ h) = (g = h)$
proof (*neg-clausify*)

assume *0*: $(f \circ g \neq f \circ h) \vee (g \neq h)$
assume *1*: $(g = h) \vee (f \circ g = f \circ h)$
have *2*: $\bigwedge X1.\ inj\text{-}on\ f\ X1$
by (*metis assms inj-on-Un Un-UNIV-left*)
have *3*: $(inv\ f \circ (f \circ g) = h) \vee h = g \vee \neg\ inj\ f$
by (*metis 1 o-assoc inj-iff id-o*)
have *4*: $(inv\ f \circ (f \circ g) = h) \vee h = g$ **by** (*metis 2 3*)
have *5*: $h = g \vee \neg\ inj\ f$ **by** (*metis id-o o-assoc inj-iff 4*)
have *6*: $h = g$ **by** (*metis 5 2*)
have *7*: $h \neq g$ **by** (*metis 6 0*)
show *False* **by** (*metis 6 7*)
qed

Función imagen

Nota 7.2.13. Imagen de un conjunto mediante una función:

- $f\ `\ A \equiv \{y \mid \exists x{\in}A.\ y = f\ x\}$ (*image_def*)

Nota 7.2.14. Propiedades de la imagen:

- $(f \circ g)\ `\ r = f\ `\ g\ `\ r$ (*image_compose*)
- $f\ `\ (A \cup B) = f\ `\ A \cup f\ `\ B$ (*image_Un*)
- $inj\ f \Longrightarrow f\ `\ (A \cap B) = f\ `\ A \cap f\ `\ B$ (*image_Int*)

Nota 7.2.15. Ejemplos de demostraciones triviales de propiedades de la imagen.

lemma $f`A \cup g`A = (\bigcup x{\in}A.\ \{f\ x, g\ x\})$
by *auto*

lemma $f\ `\ \{(x,y).\ P\ x\ y\} = \{f(x,y) \mid x\ y.\ P\ x\ y\}$
by *auto*

Nota 7.2.16. El **rango** de una función (*range f*) es la imagen del universo (*f ` UNIV*).

Nota 7.2.17. Imagen inversa de un conjunto:

- $f\ -`\ B \equiv \{x \mid f\ x \in B\}$ (*vimage_def*)

Nota 7.2.18. Propiedad de la imagen inversa de un conjunto:

- $f\ -`\ (-\ A) = -\ f\ -`\ A$ (*vimage_Compl*)

7.3 Relaciones

7.3.1 Relaciones básicas

Nota 7.3.1. La teoría de relaciones es *HOL/Relation.thy*.

Nota 7.3.2. Las relaciones son conjuntos de pares.

Nota 7.3.3. Relación identidad:

- $Id \equiv \{p \mid \exists x.\, p = (x, x)\}$ (*Id_def*)

Nota 7.3.4. Composición de relaciones:

- $r\ O\ s \equiv \{(x, z) \mid \exists y.\, (x, y) \in s \wedge (y, z) \in r\}$ (*rel_comp_def*)

Nota 7.3.5. Propiedades:

- $R\ O\ Id = R$ (*R_O_Id*)
- $[\![r' \subseteq r; s' \subseteq s]\!] \Longrightarrow r'\ O\ s' \subseteq r\ O\ s$ (*rel_comp_mono*)

Nota 7.3.6. Imagen inversa de una relación:

- $((a, b) \in r^{-1}) = ((b, a) \in r)$ (*converse_iff*)

Nota 7.3.7. Propiedad de la imagen inversa de una relación:

- $(r\ O\ s)^{-1} = s^{-1}\ O\ r^{-1}$ (*converse_rel_comp*)

Nota 7.3.8. Imagen de un conjunto mediante una relación:

- $(b \in r\ ``\ A) = (\exists x{\in}A.\, (x, b) \in r)$ (*Image_iff*)

Nota 7.3.9. Dominio de una relación:

- $(a \in Domain\ r) = (\exists y.\, (a, y) \in r)$ (*Domain_iff*)

Nota 7.3.10. Rango de una relación:

- $(a \in Range\ r) = (\exists y.\, (y, a) \in r)$ (*Range_iff*)

Nota 7.3.11. La teoría de las potencias de relaciones es *HOL/Relation-Power.thy*.

Nota 7.3.12. Potencias de relaciones:

- $R\ \hat{}\ 0 = Id$
- $R\hat{}(Suc\ n) = R\ O\ (R\hat{}n)$

7.3.2 Clausura reflexiva y transitiva

Nota 7.3.13. La teoría de la clausura reflexiva y transitiva de una relación es *HOL/Transitive-Closure.thy*.

Nota 7.3.14. La **clausura reflexiva y transitiva** de la relación r es la menor solución de la ecuación:

- $r^* = Id \cup r\ O\ r^*$ (*rtrancl_unfold*)

Nota 7.3.15. Propiedades básicas de la clausura reflexiva y transitiva:

- $(a, a) \in r^*$ (*rtrancl_refl*)
- $p \in r \Longrightarrow p \in r^*$ (*r_into_rtrancl*)
- $[\![(a, b) \in r^*; (b, c) \in r^*]\!] \Longrightarrow (a, c) \in r^*$ (*rtrancl_trans*)

Nota 7.3.16. Inducción sobre la clausura reflexiva y transitiva

- $$\frac{(a, b) \in r^* \qquad P\ a \qquad \bigwedge y\ z.\ \dfrac{(a, y) \in r^* \qquad (y, z) \in r \qquad P\ y}{P\ z}}{P\ b}$$ (*rtrancl_induct*)

Nota 7.3.17. Idempotencia de la clausura reflexiva y transitiva:

- $(r^*)^* = r^*$ (*rtrancl_idemp*)

Nota 7.3.18. Reglas de introducción de la **clausura transitiva**:

- $(a, b) \in r \Longrightarrow (a, b) \in r^+$ (*r_into_trancl*)
- $[\![(a, b) \in r^+; (b, c) \in r^+]\!] \Longrightarrow (a, c) \in r^+$ (*trancl_trans*)

Nota 7.3.19. Ejemplo de propiedad:

- $(r^{-1})^+ = (r^+)^{-1}$ (*trancl_converse*)

7.3.3 Una demostración elemental

Nota 7.3.20. El teorema que se desea demostrar es que la clausura reflexiva y transitiva conmuta con la inversa (*rtrancl-converse*). Para demostrarlo introducimos dos lemas auxiliares: *rtrancl-converseD* y *rtrancl-converseI*.

lemma *rtrancl-converseD*: $(x, y) \in (r^{-1})^* \Longrightarrow (y, x) \in r^*$
proof (*induct rule*:*rtrancl-induct*)
show $(x, x) \in r^*$ **by** (*rule rtrancl-refl*)
next
fix $y\ z$
assume $(x, y) \in (r^{-1})^*$ **and** $(y, z) \in r^{-1}$ **and** $(y, x) \in r^*$
show $(z, x) \in r^*$
proof (*rule rtrancl-trans*)
show $(z, y) \in r^*$ **using** $\langle (y, z) \in r^{-1} \rangle$ **by** *simp*
next
show $(y, x) \in r^*$ **using** $\langle (y, x) \in r^* \rangle$ **by** *simp*
qed
qed

lemma *rtrancl-converseI*: $(y, x) \in r^* \Longrightarrow (x, y) \in (r^{-1})^*$
proof (*induct rule*:*rtrancl-induct*)
show $(y, y) \in (r^{-1})^*$ **by** (*rule rtrancl-refl*)
next
fix $u\ z$
assume $(y, u) \in r^*$ **and** $(u, z) \in r$ **and** $(u, y) \in (r^{-1})^*$
show $(z, y) \in (r^{-1})^*$
proof (*rule rtrancl-trans*)
show $(z, u) \in (r^{-1})^*$ **using** $\langle (u, z) \in r \rangle$ **by** *auto*
next
show $(u, y) \in (r^{-1})^*$ **using** $\langle (u, y) \in (r^{-1})^* \rangle$ **by** *simp*
qed
qed

theorem *rtrancl-converse*: $(r^{-1})^* = (r^*)^{-1}$
proof
show $(r^{-1})^* \subseteq (r^*)^{-1}$ **by** (*auto simp add*:*rtrancl-converseD*)
next
show $(r^*)^{-1} \subseteq (r^{-1})^*$ **by** (*auto simp add*:*rtrancl-converseI*)
qed

7.4 Relaciones bien fundamentadas e inducción

Nota 7.4.1. La teoría de las relaciones bien fundamentadas es *HOL/Wellfounded-Relations.thy*.

Nota 7.4.2. La relación-objeto *less-than* es el orden de los naturales que es bien funda-

mentada:

- $((x, y) \in \textit{less-than}) = (x < y)$ (*less_than_iff*)
- *wf less-than* (*wf_less_than*)

Nota 7.4.3. Notas sobre **medidas**:

- **Imagen inversa** de una relación mediante una función:
  $\textit{inv-image}\ r\ f \equiv \{(x, y) \mid (f\ x, f\ y) \in r\}$ (*inv-image-def*)
- Conservación de la buena fundamentación:
  $\textit{wf}\ r \Longrightarrow \textit{wf}\ (\textit{inv-image}\ r\ f)$ (*wf-inv-image*)
- Definición de la **medida**:
  $\textit{measure} \equiv \textit{inv-image less-than}$ (*measure-def*)
- Buena fundamentación de la medida:
  $\textit{wf}\ (\textit{measure}\ f)$ (*wf-measure*)

Nota 7.4.4. Notas sobre el **producto lexicográfico**:

- Definición del producto lexicográfico (*lex-prod-def*):
  $ra <*lex*> rb \equiv \{((a, b), (a', b')).\ (a, a') \in ra \vee (a = a' \wedge (b, b') \in rb)\}$
- Conservación de la buena fundamentación:
  $[\![\textit{wf}\ ra; \textit{wf}\ rb]\!] \Longrightarrow \textit{wf}\ (ra <*lex*> rb)$ (*wf-lex-prod*)

Nota 7.4.5. El orden de multiconjuntos está en la teoría *HOL/Library/Multiset.thy*.

Nota 7.4.6. Inducción sobre relaciones bien fundamentadas:

- $$\frac{\textit{wf}\ r \qquad \bigwedge x.\ \dfrac{\forall y.\ (y, x) \in r \longrightarrow P\ y}{P\ x}}{P\ a}$$ (*wf-induct*)