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 26 de mayo de 2009)

Índice

Capítulo 1

Isabelle como un lenguaje funcional

1.1 Introducción

Nota 1.1.1. Esta notas son una introducción a la demostración asistida utilizando el sistema Isabelle/HOL/Isar. La versión de Isabelle utilizada es la de 2009.

Nota 1.1.2. 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.

1.2 Números naturales, enteros y booleanos

Nota 1.2.1 (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 1.2.2 (Ejemplo de simplificación de números naturales)**.** *El siguiente del* 0 *es el* 1*.*

lemma *Suc 0 = 1*
by *simp*

Nota 1.2.3 (Suma y producto de números naturales)**.** En Isabelle están definida la suma y el producto de números naturales:

- $x+y$ es la suma de x e y
- $x*y$ es el producto de x e y

Lema 1.2.4 (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 1.2.5 (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 1.2.6 (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 1.2.7 (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 1.2.8 (Ejemplo de división)**.** *La división natural de 7 entre 3 es 2.*

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

Nota 1.2.9 (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 1.2.10 (Ejemplo de resto)**.** *El resto de dividir 7 entre 3 es 1.*

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

Nota 1.2.11 (Números enteros)**.** En Isabelle también están definidos los números enteros. El tipo de los enteros se representa por *int*.

Lema 1.2.12 (Ejemplo de operación con enteros)**.** *La suma de 1 y -2 es el número entero -1.*

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

Nota 1.2.13 (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 1.2.14 (Booleanos, conectivas y cuantificadores)**.** En Isabelle están definidos los valores booleanos (*True* y *False*), las conectivas ($\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$) y los cuantificadores ($\forall$, $\exists$). El tipo de los booleanos es *bool*.

Lema 1.2.15 (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*

1.3 Definiciones no recursivas

Definición 1.3.1 (Ejemplo de definición no recursiva)**.** *La disyunción exclusiva de A y B se verifica si una es verdadera y la otra no lo es.*

definition *xor* :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool* **where**
xor A B $\equiv$ (*A* $\wedge$ $\neg$ *B*) $\vee$ ($\neg$ *A* $\wedge$ *B*)

Lema 1.3.2 (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 1.3.3 (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*]

1.4 Definiciones locales

Nota 1.4.1 (Variables locales)**.** Se puede asignar valores a variables locales mediante 'let' y usarlo en las expresiones dentro de 'in'.

Lema 1.4.2 (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*

1.5 Pares

Nota 1.5.1 (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 1.5.2 (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*

1.6 Listas

Nota 1.6.1 (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 1.6.2 (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 1.6.3 (Primero y resto)**.**

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

Lema 1.6.4 (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 1.6.5 (Longitud)**.** `length l` es la longitud de la lista `l`.

Lema 1.6.6 (Ejemplo de cálculo de longitud)**.** *La lonngitud de la lista [1,2,3] es 3.*

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

Nota 1.6.7 (Referencias sobre listas)**.** En la sesión 38 de "HOL: The basis of Higher-Order Logic" se encuentran más definiciones y propiedades de las listas.

1.7 Registros

Nota 1.7.1 (Registro)**.** Un registro es una colección de campos y valores.

Definición 1.7.2 (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 1.7.3 (Ejemplo de definición de un registro)**.** *El punto* `pt` *tiene de coordenadas 3 y 7.*

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

Lema 1.7.4 (Ejemplo de propiedad de registro)**.** *La coordenada x del punto* `pt` *es 3.*

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

Lema 1.7.5 (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*)

1.8 Funciones anónimas

Nota 1.8.1 (Funciones anónimas)**.** En Isabelle pueden definirse funciones anónimas.

Lema 1.8.2 (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 (λ *x*. *x* + *x*) *1* = (*2*::*nat*)
by *simp*

1.9 Condicionales

Definición 1.9.1 (Ejemplo con el condicional *if*)**.** *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*)

Lema 1.9.2 (Ejemplo de simplificación con el condicional *if*)**.** *El valor absoluto de -3 es 3.*

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

Definición 1.9.3 (Ejemplo con el condicional *case*)**.** *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*)

Lema 1.9.4 (Ejemplo de simplificación con el condicional *case*)**.** *El número 3 es sucesor.*

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

1.10 Tipos de datos y recursión primitiva

Definición 1.10.1 (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 1.10.2 (Ejemplo de definición primitiva recursiva)**.** `conc xs ys` *e la concatenación de las lista* `xs` *e* `ys`*.*

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

Lema 1.10.3 (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 1.10.4 (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.

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

lemma *suma 3* = *6*
by (*simp add*: *suma-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 $P \wedge (Q \wedge P)$
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* $a * a < b * b$

```
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 ∨ B and ac: A ⟶ C and bc: B ⟶ 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	$[\![\neg P; \neg R \Longrightarrow P]\!] \Longrightarrow R$

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 ∧ (∃ 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: ∃ x. P ∧ Q(x)
 shows P ∧ (∃ x. Q(x))
proof (rule conjI)
 show P
  proof −
  from e obtain x where p: P ∧ Q(x) by (rule exE)
  from p show P by (rule conjunct1)
  qed
 show ∃ y. Q(y)
  proof −
  from e obtain x where p: P ∧ Q(x) by (rule exE)
  from p have q: Q(x) by (rule conjunct2)
  from q show ∃ 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* $x \mid y$*.*

```
definition divide :: nat ⇒ nat ⇒ bool (- | - [80,80] 80) where
 x | y ≡ ∃ 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 | b and bc: b | c
```

```
 shows a | 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

Ejemplo 3.1.2 (Demostración 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 3.1.3 (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 3.1.4 (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 3.1.5 (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 3.1.6 (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 3.1.7 (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.

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

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.*

primrec *suma-impares* :: *nat* ⇒ *nat* **where**
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-def*)

Nota 3.2.5. La suma de los 3 primero número impares se puede calcular mediante

value *suma-impares 3*

que devuelve el valor *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))

Lema 3.2.6 (Ejemplo de demostración por inducción matemática)**.** *La suma de los n primeros números impares es* n^2.

Nota 3.2.7. Demostración automática del lema 3.2.6.

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

Nota 3.2.8 (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 3.2.9. Demostración con patrones del lema 3.2.6.

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

Nota 3.2.11. Demostración con patrones y razonamiento ecuacional del lema 3.2.6.

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

Nota 3.2.12. Demostración por inducción y razonamiento ecuacional del lema 3.2.6.

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

Definición 3.2.13 (Números pares)**.** *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*

Lema 3.2.14 (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-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

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-1*: $xs \mathbin{@} (ys \mathbin{@} zs) = (xs \mathbin{@} ys) \mathbin{@} zs$
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.

```
primrec espejo :: 'a arbol ⇒ 'a arbol where
  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-1: 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.

```
primrec aplana :: 'a arbol ⇒ 'a list where
  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 $P \vee Q$
 thus $Q \vee P$
 proof (*rule disjE*)
  assume P
  thus *?thesis* **by** (*rule disjI2*)
 next
  assume Q
  thus *?thesis* **by** (*rule disjI1*)
 qed
qed

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

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

Nota 4.2.4. El lema anterior puede demostrarse automáticamente como se muestra a continuación.

lemma
fixes $x :: nat$
assumes *1*: $x * x + y = 13$

and *2*: $y \neq 4$
shows $x \neq 3$
using *assms*
by *auto*

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 $(rule\ disjE)$
assume P
thus Q **using** $\langle \neg P \rangle$ **by** *contradiction*
next
assume Q
thus Q **by** *assumption*
qed

4.4 Equivalencias

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

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

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 $(\mathit{rule\ iffI})$
 assume $(R \longrightarrow C) \wedge (S \longrightarrow C)$
 thus $R \vee S \longrightarrow C$ **by** *blast*
next
 assume $R \vee S \longrightarrow C$
 thus $(R \longrightarrow C) \wedge (S \longrightarrow C)$ **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
using *assms*
by $(\mathit{rule\ iffD1})$

lemma assumes $A = B$ **and** B **shows** A

using *assms*
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.*

primrec *inversa* :: *'a list* ⇒ *'a list* **where**
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.*

primrec *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* []

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.*

lemma *inversaAc* [*1*,*2*,*3*] = *inversa* [*1*,*2*,*3*]
by (*simp add*: *inversaAc-def*)

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 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

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*
inversaAcAux xs ys = inversa xs @ ys.

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

Corolario 5.1.7. *Para cualquier lista xs, se tiene que inversaAc xs = inversa xs.*

corollary *inversaAc xs = inversa xs*
by (*simp add*: *inversaAcAux-es-inversa inversaAc-def*)

Nota 5.1.8. 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`	$(xs \mathbin{@} ys) \mathbin{@} zs = xs \mathbin{@} (ys \mathbin{@} zs)$
`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 $(simp\ only{:}append.append\text{-}Nil)$
also have $\ldots = (inversa\ xs)@([a]@ys)$ **by** $(simp\ only{:}append.append\text{-}Cons)$
also have $\ldots = ((inversa\ xs)@[a])@ys$ **by** $(simp\ only{:}append\text{-}assoc)$
also have $\ldots = (inversa\ (a\#xs))@ys$ **by** $(simp\ only{:}inversa.simps(2))$
finally show $?thesis$ **.**
qed

5.2 Recursión general. La función de Ackermann

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 Wipipedia).

Definición 5.2.1. *La función de Ackermann se define por*

$$A(m,n) = \begin{cases} n+1, & si\ m = 0, \\ A(m-1,1) & si\ m > 0\ y\ n = 0, \\ A(m-1, A(m,n-1)), & si\ m > 0\ y\ n > 0 \end{cases}$$

para todo los números naturales. La función de Ackermann es recursiva, pero no es primitiva recursiva.

fun $ack :: nat \Rightarrow nat \Rightarrow nat$ **where**
$ack\ 0\ n = n{+}1$
$|\ ack\ (Suc\ m)\ 0 = ack\ m\ 1$
$|\ ack\ (Suc\ m)\ (Suc\ n) = ack\ m\ (ack\ (Suc\ m)\ n)$

Nota 5.2.2 (Definiciones recursivas generales)**.**

- Las definiciones recursivas generales se identifican mediante **fun**.

- Al definir una función recursiva general se genera una regla de inducción. En la definición anterior, la regla generada es

$$(ack.induct)\quad \frac{\bigwedge n.\ P\ 0\ n \qquad \bigwedge m.\ \dfrac{P\ m\ 1}{P\ (Suc\ m)\ 0} \qquad \bigwedge m\ n.\ \dfrac{P\ (Suc\ m)\ n \qquad P\ m\ (ack\ (Suc\ m)\ n)}{P\ (Suc\ m)\ (Suc\ n)}}{P\ a0.0\ a1.0}$$

Nota 5.2.3 (Ejemplo de cálculo)**.** El cálculo del valor de la función de Ackermann para 2 y 3 se realiza mediante

value *ack 2 3*

y se obtiene 9.

Lema 5.2.4. *Para todos m y n,* $A(m, n) > n$.

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

La demostración automática es

lemma *ack m n* $>$ *n*
by (*induct m n rule*: *ack.induct*) *simp-all*

Nota 5.2.5 (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*)

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*)

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

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*.

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)
```

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)
```

$= map\ h\ (aplana\text{-}bosque\ (ConsB\ a\ b))$ **by** *simp*
qed

lemma $aplana\text{-}arbol\ (map\text{-}arbol\ a\ h) = map\ h\ (aplana\text{-}arbol\ a)$
$\wedge\ aplana\text{-}bosque\ (map\text{-}bosque\ b\ h) = map\ h\ (aplana\text{-}bosque\ b)$
by (*induct-tac a* **and** *b*) *auto*

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.*

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

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.*

primrec *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*))))

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.*

primrec *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*]

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
```