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 *9*

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*∗(*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