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* :: $'a\ list \Rightarrow\ 'a\ list$ **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* :: $'a\ list \Rightarrow\ 'a\ list \Rightarrow\ 'a\ list$ **where**
$inversaAcAux\ []\ ys = ys$
$|\ inversaAcAux\ (x\#xs)\ ys = inversaAcAux\ xs\ (x\#ys)$

definition *inversaAc* :: $'a\ list \Rightarrow\ 'a\ list$ **where**
$inversaAc\ xs \equiv inversaAcAux\ xs\ []$

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 $\bigwedge$*ys*. *inversaAcAux* [] *ys* = (*inversa* [])@*ys* **by** *simp*
next
 fix *a xs*
 assume *HI*: $\bigwedge$*ys*. *inversaAcAux xs ys* = *inversa xs*@*ys*
 show $\bigwedge$*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* @ $(a \cdot ys)$ = *inversa* $(a \cdot xs)$ @ *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 $\ldots = (inversa\ xs)@([a]@ys)$ **by** (*simp only*:*append.append-Cons*)
 also have $\ldots = ((inversa\ xs)@[a])@ys$ **by** (*simp only*:*append-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 *fmn* 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 $'a\ arbol = Hoja \mid Nodo\ 'a\ 'a\ bosque$
and $'a\ bosque = Vacio \mid ConsB\ 'a\ arbol\ 'a\ bosque$

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

$$\frac{\begin{array}{cc} P1.0\ Hoja & \bigwedge a\ bosque.\ \dfrac{P2.0\ bosque}{P1.0\ (Nodo\ a\ bosque)} \\ P2.0\ Vacio & \bigwedge arbol\ bosque.\ \dfrac{P1.0\ arbol \qquad P2.0\ bosque}{P2.0\ (ConsB\ arbol\ bosque)} \end{array}}{P1.0\ arbol \wedge P2.0\ bosque}$$

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
$aplana\text{-}arbol :: 'a\ arbol \Rightarrow 'a\ list$ **and**
$aplana\text{-}bosque :: 'a\ bosque \Rightarrow 'a\ list$ **where**
$aplana\text{-}arbol\ Hoja = []$
$\mid aplana\text{-}arbol\ (Nodo\ x\ b) = x\#(aplana\text{-}bosque\ b)$
$\mid aplana\text{-}bosque\ Vacio = []$
$\mid aplana\text{-}bosque\ (ConsB\ a\ b) = (aplana\text{-}arbol\ a)\ @\ (aplana\text{-}bosque\ b)$

fun

map-arbol :: *'a arbol* $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ *'b arbol* **and**
map-bosque :: *'a bosque* $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ *'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*)
$\wedge$ *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*