Capítulo 12
Derivación de Programas

Índice del Capítulo

12.1. Introducción

Si bien un paso importante en la solución de un problema es escribir una especificación correcta que describa el problema, una vez hallada ésta, es necesario manipularla adecuadamente de modo de transformarla en la implementación de un programa. Veremos a continuación técnicas que nos permitirán llevar a cabo este objetivo.

12.2. Derivación de funciones recursivas

En esta sección mostraremos cómo obtener una definición recursiva de una función a partir de su especificación aplicando el principio de inducción. Este tipo de derivación de programas consiste en definir una función pudiendo usar para ello los valores de la función aplicada a elementos con menor estructura. Por ejemplo, en el caso de los números naturales, la función utilizará valores de la función sobre números más pequeños, y en el caso de listas utilizará valores de la función sobre listas de longitud menor.

(12.1) **Ejemplo.** Veamos un ejemplo sencillo de derivación. Dado un número natural se desea obtener una función que calcule su factorial. Primero daremos la especificación de esta función $fac : Nat \to Nat$:

$$\begin{array}{ll} pre: & true \\ post: & fac.n = (\prod\ i : 0 < i \leq n : i) \end{array}$$

Para hallar una definición recursiva que satisfaga esta especificación podemos pensar que la especificación de *fac* es una ecuación a resolver con incógnita *fac* . Luego el proceso de derivación consistirá en "despejar" *fac*. El proceso de despejar nos asegura que si ahora reemplazamos el valor obtenido en la ecuación original (la especificación) obtenemos un enunciado verdadero. Veamos cómo sería esto.

Caso base

$$\begin{array}{ll} & fac.0 \\ = & \quad \langle \text{especificación de } fac \rangle \\ & (\prod\ i : 0 < i \leq 0 : i) \\ = & \quad \langle \text{rango vacío} \rangle \\ & 1 \end{array}$$

Paso inductivo

$$\begin{array}{ll} & fac.(n+1) \\ = & \quad \langle \text{especificación de } fac \rangle \\ & (\prod\ i : 0 < i \leq n+1 : i) \\ = & \quad \langle \text{separación de un término} \rangle \\ & (\prod\ i : 0 < i \leq n : i) \times (n+1) \\ = & \quad \langle \text{hipótesis inductiva} \rangle \\ & fac.n \times (n+1) \end{array}$$

Por lo tanto definiendo recursivamente *fac* como:

$$\begin{array}{rcl} fac.0 & \doteq & 1 \\ fac.(n+1) & \doteq & (n+1) \times fac.n \end{array}$$

se satisface la especificación dada.

Además de haber construído un programa que satisface una especificación hemos obtenido una demostración de que lo hace. Esta demostración no es más que la anterior invirtiendo los pasos de atrás para adelante, a partir del segundo paso de la derivación y cambiando la justificación "especificación de *fac*" por "definición de *fac*". Veamos otro ejemplo.

(12.2) **Ejemplo.** Dada una lista de enteros se desea obtener una función que calcule la suma de sus elementos. Primero especificamos formalmente esta función y definimos su tipo $sum : [Int] \to Int$.

$$\begin{array}{ll} pre: & true \\ post: & sum.xs = (\sum\ i : 0 \leq i < \#xs : xs.i) \end{array}$$

A partir de esta especificación calcularemos una definición recursiva haciendo inducción sobre la lista.

Caso base $xs = [\,]$

$$\begin{array}{ll} & sum.[\,] \\ = & \quad \langle \text{especificación } sum \rangle \\ & (\sum\ i : 0 \leq i < \#[\,] : [\,].i) \\ = & \quad \langle \text{definición de\#; rango vacío} \rangle \\ & 0 \end{array}$$

En esta prueba podemos ver la diferencia entre una derivación y una verificación. El objetivo de la verificación es llegar a la fórmula (para el caso que se está probando), mientras que el objetivo de la derivación es simplificar esta fórmula hasta obtener una expresión del formalismo básico que pueda tomarse como definición, en este caso la constante 0.

Consideremos ahora el paso inductivo,

Paso inductivo

$$\begin{array}{ll} & sum.(x \triangleright xs)) \\ = & \quad \langle \text{especificación de } sum \rangle \\ & (\sum\ i : 0 \leq i < \#(x \triangleright xs) : (x \triangleright xs).i) \\ = & \quad \langle \text{definición de \#} \rangle \\ & (\sum\ i : 0 \leq i < 1 + \#xs : (x \triangleright xs).i) \\ = & \quad \langle \text{separación de un término} \rangle \\ & (x \triangleright xs).0\ +\ (\sum\ i : 1 \leq i < 1 + \#xs : (x \triangleright xs).i) \\ = & \quad \langle \text{cambio de variable dummy (f.i=i+1); definición de .} \rangle \\ & x\ +\ (\sum\ j : 1 \leq j + 1 < 1 + \#xs : (x \triangleright xs).(j+1)) \\ = & \quad \langle \text{definición de indexar, aritmética} \rangle \\ & x\ +\ (\sum\ j : 0 \leq j < \#xs : xs.j) \\ = & \quad \langle \text{hipótesis inductiva} \rangle \\ & x\ +\ sum.xs \end{array}$$

Por lo tanto, *sum* puede definirse recursivamente como:

$$\begin{array}{lcl} sum.[\,] & \doteq & 0 \\ sum.(x \triangleright xs) & \doteq & x + sum.xs \end{array}$$

(12.3) **Ejemplo.** En el capítulo 10 dimos una especificación de la función $ev : [Real] \to y \to Real$ que evalúa un polinomio en un valor dado. Para ello, decidimos representar a los polinomios como listas de números $[a_0, \ldots, a_n]$, donde cada a_i representa un coeficiente del polinomio $a^n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$.

Derivaremos una definición de esta función a partir de la especificación dada:

$$\begin{array}{ll} pre: & \#xs > 0 \land y \neq 0 \\ post: & ev.xs.y = (\sum\ i : 0 \leq i < \#xs : xs.i \times y^i) \end{array}$$

, y planteemos la propiedad sobre la cual aplicaremos inducción, para así calcular la definición de la función ev, la cual obtenemos a partir del planteo $pre \Rightarrow post$.

$$\begin{array}{ll} & pre \Rightarrow post \\ = & \quad \langle \text{Def. de pre y post de } ev \rangle \\ & \#xs > 0 \land y \neq 0 \Rightarrow ev.xs.y = (\sum\ i : 0 \leq i < \#xs : xs.i \times y^i) \\ = & \quad \langle \text{ Logica proposicional: traslación} \rangle \\ & \#xs > 0 \Rightarrow (y \neq 0 \Rightarrow ev.xs.y = (\sum\ i : 0 \leq i < \#xs : xs.i \times y^i)) \end{array}$$

Hemos modificado la expresión, a fin de que resulte explícito el argumento sobre el cual se hará inducción, en este caso, la lista real xs no vacía:

$$P.xs\ : \#xs > 0 \Rightarrow (y \neq 0 \Rightarrow ev.xs.y = (\sum\ i : 0 \leq i < \#xs : xs.i \times y^i))$$

aplicando inducción sobre la expresión:

$$(\forall\ xs : [Real]\ : \#xs > 0\ : y \neq 0 \Rightarrow ev.xs.y = (\sum\ i : 0 \leq i < \#xs : xs.i \times y^i))$$

Utilizaremos suposición del antecedente en la implicancia del término, para utilizar la igualdad en la derivación.

Caso base: $xs = [x]$, $y \neq 0$

$$\begin{array}{ll} & ev.[x].y \\ = & \quad \langle \text{especificación de } ev \rangle \\ & (\sum\ i : 0 \leq i < \#[x] : [x].i \times y^i) \\ = & \quad \langle \text{definición de \#;rango unitario} \rangle \\ & [x].0 \times y^0 \\ = & \quad \langle \text{aritmética con supuesto } y \neq 0 \text{ ; definición de .} \rangle \\ & x \end{array}$$

Paso inductivo: $(z \triangleright x \triangleright xs)$, $y \neq 0$

$$
\begin{array}{ll}
 & ev.(z \triangleright x \triangleright xs).y \\
= & \quad \langle \text{especificación de } ev \rangle \\
 & (\sum\, i : 0 \le i < \#(z \triangleright x \triangleright xs) : (z \triangleright x \triangleright xs).i \times y^i) \\
= & \quad \langle \text{definición de \#; separación de un término} \rangle \\
 & (z \triangleright x \triangleright xs).0 \times y^0 + (\sum\, i : 1 \le i < 1 + \#(x \triangleright xs) : (z \triangleright x \triangleright xs).i \times y^i) \\
= & \quad \langle \text{definición de indexar; cambio de variable dummy (f.i=i+1); aritmética} \rangle \\
 & z \times y^0 + (\sum\, i : 0 \le i < \#(x \triangleright xs) : (z \triangleright x \triangleright xs).(i+1) \times y^{i+1}) \\
= & \quad \langle \text{definición de indexar; aritmética: calculo de potencia con} y \neq 0 \rangle \\
 & z + (\sum\, i : 0 \le i < \#(x \triangleright xs) : (x \triangleright xs).i \times y \times y^i) \\
= & \quad \langle \text{distributividad de } \times \text{ respecto de } \sum \text{con rango no vacío} \rangle \\
 & z + y \times (\sum\, i : 0 \le i < \#(x \triangleright xs) : (x \triangleright xs).i \times y^i) \\
= & \quad \langle \text{hipótesis inductiva } \rangle \\
 & z + y \times ev.(x \triangleright xs).y
\end{array}
$$

Por lo tanto una definición recursiva para ev es:

$$
\begin{array}{lcl}
ev & :: & [Real] \to Real \to Real \\
ev.[x].y & \doteq & (y \neq 0 \to x) \\
ev.(z \triangleright x \triangleright xs).y & \doteq & (y \neq 0 \to z + y \times ev.(x \triangleright xs).y)
\end{array}
$$

Observemos como el supuesto que haciamos en la derivación sobre la variable $y \neq 0$ se transformó en nuestra derivación en una condición dentro de la guarda para la definición de la función ev. Es importante no olvidarnos que la validez de nuestro cálculo para esta función se sustentó en el supuesto que hacíamos, por lo tanto, deberá ser contemplado también en la definición de la función que demos.

12.3. Modularización

Una técnica muy utilizada en la construcción de programas que ahora utilizaremos en la derivación de los mismos es la *modularización*. Esta técnica se utiliza cuando la solución de un problema requiere la solución de un "subproblema", y consiste en no atacar ambos problemas simultáneamente sino por módulos, donde cada módulo es independiente del otro.

Veamos un ejemplo.

(12.4) Ejemplo. Queremos derivar la definición de una función $g :: [Int] \to [Int] \to Bool$ que toma 2 listas no vacías y determina si todos los elementos de la primer lista son mayores al mínimo valor de la segunda lista.

Especificamos esta función de la siguiente manera:

$$
\begin{array}{ll}
pre: & \#xs > 0 \ \wedge \ \#ys > 0 \\
post: & g.xs.ys = (\forall\, i : 0 \le i < \#xs : xs.i > (min\, j : 0 \le j < \#ys : ys.j))
\end{array}
$$

A partir de esta especificación derivamos g, haciendo inducción sobre xs:

Caso base: $xs = [x]$

$$
\begin{array}{ll}
 & g.[x].ys \\
= & \quad \langle \text{especificación de } g \rangle \\
 & (\forall\, i : 0 \le i < \#[x] : [x].i > (\mathit{min}\, j : 0 \le j < \#ys : ys.j)) \\
= & \quad \langle \text{definición de \#; rango unitario} \rangle \\
 & [x].0 > (\mathit{min}\, j : 0 \le j < \#ys : ys.j) \\
= & \quad \langle \text{definición de .} \rangle \\
 & x > (\mathit{min}\, j : 0 \le j < \#ys : ys.j)
\end{array}
$$

Paso inductivo: $(y \triangleright x \triangleright xs)$

$$
\begin{array}{ll}
 & g.(y \triangleright x \triangleright xs).ys \\
= & \quad \langle \text{especificación de } g \rangle \\
 & (\forall\, i : 0 \le i < \#(y \triangleright x \triangleright xs) : (y \triangleright x \triangleright xs).i > (\mathit{min}\, j : 0 \le j < \#ys : ys.j)) \\
= & \quad \langle \text{definición de \#; separación de un término} \rangle \\
 & (y \triangleright x \triangleright xs).0 > (\mathit{min}\, j : 0 \le j < \#ys : ys.j) \wedge \\
 & (\forall\, i : 1 \le i < \#(x \triangleright xs) + 1 : (y \triangleright x \triangleright xs).i > (\mathit{min}\, j : 0 \le j < \#ys : ys.j)) \\
= & \quad \langle \text{cambio de variable dummy (f.i=i+1); aritmética; definición de .} \rangle \\
 & y > (\mathit{min}\, j : 0 \le j < \#ys : ys.j) \wedge \\
 & (\forall\, k : 0 \le k < \#(x \triangleright xs) : (y \triangleright x \triangleright xs).(k+1) > (\mathit{min}\, j : 0 \le j < \#ys : ys.j)) \\
= & \quad \langle \text{definición de .} \rangle \\
 & y > (\mathit{min}\, j : 0 \le j < \#ys : ys.j) \wedge \\
 & (\forall\, k : 0 \le k < \#(x \triangleright xs) : (x \triangleright xs).k > (\mathit{min}\, j : 0 \le j < \#ys : ys.j)) \\
= & \quad \langle \text{hipótesis inductiva} \rangle \\
 & y > (\mathit{min}\, j : 0 \le j < \#ys : ys.j) \;\wedge\; g.(x \triangleright xs).ys
\end{array}
$$

Para obtener una definición de g agregaremos una definición de función que calcule el mínimo valor de una lista. Llamamos a esta nueva función m y la especificamos de la siguiente manera:

$$
\begin{array}{ll}
\mathit{pre}: & \#xs > 0 \\
\mathit{post}: & m.xs = (\mathit{min}\, i : 0 \le i < \#xs : xs.i)
\end{array}
$$

La independencia de los módulos significa que una vez definida m, debe quedar definida g. Derivamos ahora la función m:

Caso base: $xs = [x]$

$$
\begin{array}{ll}
& m.[x] \\
= & \quad \langle \text{especificación de } m \rangle \\
& (\mathit{min}\, i : 0 \leq i < \#[x] : [x].i) \\
= & \quad \langle \text{definición de \#; rango unitario} \rangle \\
& [x].\, 0 \\
= & \quad \langle \text{definición de .} \rangle \\
& x
\end{array}
$$

Paso inductivo: $(z \triangleright x \triangleright xs)$

$$
\begin{array}{ll}
& m.\,(z \triangleright x \triangleright xs) \\
= & \quad \langle \text{especificación de } m \rangle \\
& (\mathit{min}\, i : 0 \leq i < \#(z \triangleright x \triangleright xs) : (z \triangleright x \triangleright xs).i) \\
= & \quad \langle \text{definición de \#; separación de un término} \rangle \\
& \mathit{min}((z \triangleright x \triangleright xs).\, 0, (\mathit{min}\, i : 1 \leq i < \#(x \triangleright xs) + 1 : (z \triangleright x \triangleright xs).i)) \\
= & \quad \langle \text{definición de indexar; cambio de variable dummy (f.i=i+1); aritmética} \rangle \\
& \mathit{min}(z, (\mathit{min}\, j : 1 \leq j + 1 < \#(x \triangleright xs) + 1 : (z \triangleright x \triangleright xs).(j+1)) \\
= & \quad \langle \text{definición de indexar; aritmética } \rangle \\
& \mathit{min}(z, (\mathit{min}\, j : 0 \leq j < \#(x \triangleright xs) : (x \triangleright xs).j)) \\
= & \quad \langle \text{hipótesis inductiva} \rangle \\
& \mathit{min}(z, m.(x \triangleright xs))
\end{array}
$$

La definición recursiva de g es :

$$
\begin{array}{lcl}
g.[x].(y \triangleright ys) & \doteq & x > m.(y \triangleright ys) \\
g.(z \triangleright x \triangleright xs).(y \triangleright ys) & \doteq & z > m.(y \triangleright ys) \;\wedge\; g.(x \triangleright xs).(y \triangleright ys)
\end{array}
$$

$$
\begin{array}{lcll}
m.[x] & \doteq & x & \\
m.z \triangleright (x \triangleright xs) & \doteq & (\quad z \geq m.(x \triangleright xs) & \rightarrow \quad m.(x \triangleright xs) \\
 & & \Box \quad z < m.(x \triangleright xs) & \rightarrow \quad z \\
 & &) &
\end{array}
$$

12.4. Generalización por Abstracción

En la sección anterior hemos analizado ejemplos simples de derivación de programas. Si bien no hemos tenido inconvenientes en el proceso de derivación de estos programas, no siempre podremos construir un programa a partir de la especificación utilizando la técnica dada, encontraremos ejemplos en los cuales la hipótesis inductiva no puede aplicarse de manera directa. Una técnica utilizada para resolver este tipo de derivaciones es la *generalización por abstracción*. La idea de ésta consiste en buscar una especificación más general que la dada y que pueda derivarse en forma directa. La función obtenida tendrá como caso particular la función que se desea encontrar. Para encontrar la generalización adecuada se introducirán parámetros nuevos a la función.

Veamos un ejemplo.

(12.5) **Ejemplo.** Supogamos que queremos hallar la definición recursiva de una función que determina si todas las sumas parciales de una lista son mayores o iguales a 0. La especificación de esta función está dada por:

$$\begin{array}{ll} pre: & true \\ post: & p.xs = (\forall\, i : 1 \le i \le \#xs : sum.(xs \uparrow i) \ge 0) \end{array}$$

donde $p : [Int] \to Bool$. Derivemos ahora una definición recursiva para p, haciendo inducción sobre la lista.

Caso base: $xs = [\,]$

$$\begin{array}{ll} & p.[\,] \\ = & \quad \langle \text{especificación de } p \rangle \\ & (\forall\, i : 1 \le i \le \#[\,] : sum.([\,] \uparrow i) \ge 0) \\ = & \quad \langle \text{definición de \#; rango vacío} \rangle \\ & true \end{array}$$

Paso inductivo: $x \triangleright xs$

$$\begin{array}{ll} & p.(x \triangleright xs) \\ = & \quad \langle \text{especificación de } p \rangle \\ & (\forall\, i : 1 \le i \le \#(x \triangleright xs) : sum.((x \triangleright xs) \uparrow i) \ge 0) \\ = & \quad \langle \text{definición de \#; separación de un término} \rangle \\ & sum.((x \triangleright xs) \uparrow 1) \ge 0 \ \wedge\ (\forall\, i : 2 \le i \le 1 + \#xs : sum.((x \triangleright xs) \uparrow i) \ge 0) \\ = & \quad \langle \text{definición de } \uparrow \text{ ; cambio de variable dummy (f.i=i+1)} \rangle \\ & sum.[x] \ge 0 \ \wedge\ (\forall\, j : 2 \le j + 1 \le 1 + \#xs : sum.((x \triangleright xs) \uparrow (j+1)) \ge 0) \\ = & \quad \langle \text{definición de } sum \text{; aritmética} \rangle \\ & x \ge 0 \ \wedge\ (\forall\, j : 1 \le j \le \#xs : sum.((x \triangleright xs) \uparrow (j+1)) \ge 0) \\ = & \quad \langle \text{definición de } \uparrow \rangle \\ & x \ge 0 \ \wedge\ (\forall\, j : 1 \le j \le \#xs : sum.(x \triangleright (xs \uparrow j)) \ge 0) \\ = & \quad \langle \text{definición de } sum \rangle \\ & x \ge 0 \ \wedge\ (\forall\, j : 1 \le j \le \#xs : x + sum.(xs \uparrow j) \ge 0) \end{array}$$

En este punto notamos que podría aplicarse la hipótesis inductiva de no ser por la x que esta sumando. No parece haber ninguna forma de eliminar esta x por lo cual se propone derivar la definición de una función más general que p. Llamaremos a esta función generalizada $gp : Int \to [Int] \to Bool$ y la especificaremos de la siguiente forma.

$$\begin{array}{ll} pre: & true \\ post: & gp.n.xs = (\forall\, i : 1 \le i \le \#xs : n + sum.(xs \uparrow i) \ge 0) \end{array}$$

Esta nueva especificación está inspirada en la última expresión de la derivación de p . Que esta nueva derivación pueda llevarse adelante dependerá de las propiedades del dominio en

cuestión y puede no llegar a buen puerto o tener que volver a generalizarse a su vez. La programación es una actividad creativa y esto se manifiesta en este caso en la elección de las posibles generalizaciones.

Antes de comenzar la derivación de gp, debe determinarse si efectivamente gp generaliza a p, o lo que es lo mismo, si podemos definir p en términos de gp. En el ejemplo, ésto se cumple dado que $p.xs = gp.0.xs$, por lo tanto derivaremos gp directamente.

El caso base es similar al de p, el resultado es *true* por la aplicación de rango vacío sobre el cuantificador.

Paso inductivo: $x \triangleright xs$

$$
\begin{array}{ll}
 & gp.n.(x \triangleright xs) \\
= & \quad \langle \text{especificación de } gp \rangle \\
 & (\forall\, i : 1 \leq i \leq \#(x \triangleright xs) : n + sum.((x \triangleright xs) \uparrow i) \geq 0) \\
= & \quad \langle \text{definición de \#; separación de un término} \rangle \\
 & n + sum.((x \triangleright xs) \uparrow 1) \geq 0 \ \wedge \ (\forall\, i : 2 \leq i \leq 1 + \#xs : n + sum.((x \triangleright xs) \uparrow i) \geq 0) \\
= & \quad \langle \text{definición de } \uparrow \text{; cambio de variable dummy (f.i=i+1)} \rangle \\
 & n + sum.[x] \geq 0 \ \wedge \ (\forall\, j : 2 \leq j + 1 \leq 1 + \#xs : n + sum.((x \triangleright xs) \uparrow (j+1)) \geq 0) \\
= & \quad \langle \text{definición de } sum \text{; aritmética} \rangle \\
 & n + x \geq 0 \ \wedge \ (\forall\, j : 1 \leq j \leq \#xs : n + sum.((x \triangleright xs) \uparrow (j+1)) \geq 0) \\
= & \quad \langle \text{definición de } \uparrow \rangle \\
 & n + x \geq 0 \ \wedge \ (\forall\, j : 1 \leq j \leq \#xs : n + sum.(x \triangleright (xs \uparrow j)) \geq 0) \\
= & \quad \langle \text{definición de } sum \rangle \\
 & n + x \geq 0 \ \wedge \ (\forall\, j : 1 \leq j \leq \#xs : n + (x + sum.(xs \uparrow j)) \geq 0) \\
= & \quad \langle \text{asociatividad de } + \text{; hipótesis inductiva} \rangle \\
 & n + x \geq 0 \ \wedge \ gp.(x+n).xs
\end{array}
$$

El resultado completo de la derivación es el siguiente programa:

$$
\begin{array}{lcl}
p.xs & \doteq & gp.0.xs \\
gp.n.[\,] & \doteq & True \\
gp.n.(x \triangleright xs) & \doteq & n + x \geq 0 \ \wedge \ gp.(n+x).xs
\end{array}
$$

Veamos otro ejemplo:

(12.6) **Ejemplo.** En el capítulo 10 especificamos la función $bal : [Bool] \to Bool$, que determina si la lista que recibe como argumento contiene igual cantidad de elementos $True$ que $False$.

$$
\begin{array}{ll}
pre: & true \\
post: & bal.xs = ((N\, i : 0 \leq i < \#xs : xs.i) = (N\, i : 0 \leq i < \#xs : \neg xs.i))
\end{array}
$$

Derivemos una función recursiva para *bal* haciendo inducción sobre xs.

Caso base: $xs = [\,]$

$$
\begin{array}{ll}
 & bal.[\,] \\
= & \quad \langle \text{especificación de } bal \rangle \\
 & ((N\, i : 0 \le i < \#[\,] : [\,].i) = (N\, i : 0 \le i < \#[\,] : \neg[\,].i)) \\
= & \quad \langle \text{definición de \#; rango vacío} \rangle \\
 & 0 = 0 \\
= & \quad \langle \text{igualdad de enteros} \rangle \\
 & true
\end{array}
$$

Para probar el paso inductivo dividiremos la prueba en dos: trataremos primero el caso en que el primer elemento de la lista es $True$ y luego el caso en que es $False$. Es necesario hacer ésto para poder derivar una definición de función, la cual depende de este valor. La función derivada quedará definida por análisis por casos.

Paso inductivo: $x \triangleright xs$

Caso $x = True$

$$
\begin{array}{ll}
 & bal.(True \triangleright xs) \\
= & \quad \langle \text{especificación de } bal \rangle \\
 & ((N\, i : 0 \le i < \#(True \triangleright xs) : (True \triangleright xs).i) = \\
 & (N\, i : 0 \le i < \#(True \triangleright xs) : \neg(True \triangleright xs).i)) \\
= & \quad \langle \text{definición de \#; definición de } N \rangle \\
 & ((+\, i : 0 \le i < \#xs + 1 \wedge (True \triangleright xs).i : 1) = \\
 & (+\, i : 0 \le i < \#xs + 1 \wedge \neg(True \triangleright xs).i : 1)) \\
= & \quad \langle \text{aritmética} \rangle \\
 & ((+\, i : (i = 0 \vee 1 \le i < \#xs + 1) \wedge (True \triangleright xs).i : 1) = \\
 & (+\, i : (i = 0 \vee 1 \le i < \#xs + 1) \wedge \neg(True \triangleright xs).i : 1)) \\
= & \quad \langle \text{distributividad de } \wedge \text{ respecto de } \vee \text{; partición de rango} \rangle \\
 & ((+\, i : i = 0 \wedge (True \triangleright xs).i : 1) + (+\, i : 1 \le i < \#xs + 1 \wedge (True \triangleright xs).i : 1) = \\
 & (+\, i : i = 0 \wedge \neg(True \triangleright xs).i : 1) + (+\, i : 1 \le i < \#xs + 1 \wedge \neg(True \triangleright xs).i : 1)) \\
= & \quad \langle \text{leibniz; definición de .; definición de } False \rangle \\
 & ((+\, i : i = 0 \wedge True : 1) + (+\, i : 1 \le i < \#xs + 1 \wedge (True \triangleright xs).i : 1) = \\
 & (+\, i : i = 0 \wedge False : 1) + (+\, i : 1 \le i < \#xs + 1 \wedge \neg(True \triangleright xs).i : 1)) \\
= & \quad \langle \text{neutro de } \wedge \text{; elemento absorbente de } \wedge \rangle \\
 & ((+\, i : i = 0 : 1) + (+\, i : 1 \le i < \#xs + 1 \wedge (True \triangleright xs).i : 1) = \\
 & (+\, i : False : 1) + (+\, i : 1 \le i < \#xs + 1 \wedge \neg(True \triangleright xs).i : 1)) \\
= & \quad \langle \text{rango unitario; rango vacío, aritmética} \rangle \\
 & 1 + (+\, i : 1 \le i < \#xs + 1 \wedge (True \triangleright xs).i : 1) = \\
 & (+\, i : 1 \le i < \#xs + 1 \wedge \neg(True \triangleright xs).i : 1)) \\
= & \quad \langle \text{cambio de variable dummy (f.i=i+1); aritmética; definición de .} \rangle \\
 & 1 + (+\, j : 0 \le j < \#xs \wedge xs.j : 1) = (+\, j : 0 \le j < \#xs \wedge \neg xs.j : 1))
\end{array}
$$

Aquí notamos que no es posible aplicar la hipótesis a raíz del sumando 1. Para el caso en que $x = False$, tenemos el mismo problema, aparece un sumando en la segun-

da cuantificación. Decidimos entonces definir una función más general $gbal : Int \to [Bool] \to Bool$, cuya especificación es:

$$\begin{array}{ll} pre: & true \\ post: & gbal.y.xs = (y + (N\, i : 0 \leq i < \#xs : xs.i) = (N\, i : 0 \leq i < \#xs : \neg xs.i)) \end{array}$$

Notemos que es suficiente agregar una variable de tipo entera, ya que en el caso $x = False$ pasaremos restando el valor 1 a la segunda cuantificación para poder aplicar la hipótesis.

La derivación de *gbal* es similar a la de *bal* y la dejamos como ejercicio. El resultado obtenido de la derivación será el siguiente:

$$\begin{array}{lcl} bal.xs & \doteq & gbal.0.xs \\ gbal.y.[\,] & \doteq & (y = 0) \\ gbal.y.(x \triangleright xs) & \doteq & (\ x = True \ \to \ gbal.(n+1).xs \\ & & \Box\ x = False \ \to \ gbal.(n-1).xs \\ & &) \end{array}$$

12.5. Ejercicios

12.1 Derivar una definición recursiva para las siguientes funciones según las especificaciones dadas:

a) $sumesp : Nat \to Nat$
$pre: \ true$
$post: \ sumesp.n = (\sum\ i : 0 \leq i \leq n : 2^i + 1)$

b) $prod : [Int] \to Int$
$pre: \ true$
$post: \ prod.xs = (\prod\ i : 0 \leq i \leq \#xs : xs.i)$

c) $igualk : Nat \to [Nat] \to Bool$
$pre : \quad \#xs > 0$
$post : \quad igualk.k.xs = (\forall\, i : 0 \leq i < \#xs : xs.i = k)$

d) $estak : Nat \to [Nat] \to Bool$
$pre : \quad \#xs > 0$
$post : \quad estak.k.xs = (\exists\, i : 0 \leq i < \#xs : xs.i = k)$

e) $mayoresk : Int \to [Int] \to Bool$
$pre : \quad \#xs > 0$
$post : \quad mayoresk.k.xs = (\forall\, i : 0 \leq i < \#xs : xs.i > k)$

12.2 Derive una definición recursiva de la función $allEven : [Int] \to Bool$, que determina si todos los elementos de una lista son pares.

12.3 Derive una definición recursiva de la función $maxAtFirst : [Int] \to Bool$, que determina si el primer elemento de la lista contiene al máximo.

12.4 Derivar una definición recursiva para cada una de las especificaciones dadas de la función $iguales : [A] \longrightarrow Bool$, la cual determina si los elementos de una lista dada son todos iguales entre sí. Comparar los resultados obtenidos en cada caso.

a) $pre : \quad \#xs > 0$
$post : \quad iguales.xs = (\forall\, i : 0 \leq i < \#xs - 1 : xs.i = xs.(i + 1))$

b) $pre : \quad \#xs > 0$
$post : \quad iguales.xs = (\forall\, i : 0 \leq i < \#xs : xs.i = xs.\, 0)$

c) $pre : \quad \#xs > 0$
$post : \quad iguales.xs = (\forall\, i : 0 \leq i < \#xs : xs.i = last.xs)$

12.5 Derivar una definición recursiva de la función $creciente : [Int] \longrightarrow Bool$, que dada una lista no vacía de enteros, determina si los elementos de la lista están ordenados en forma creciente.

12.6 Derive una definición recursiva de la función $sumacot : Int \to [Int] \to Bool$, que dados un entero n y una lista de enteros xs, determine si la suma de los elementos de xs es menor a n.

12.7 Derivar una definición recursiva de la función $p : [Nat] \longrightarrow Bool$, que dada una lista no vacía de naturales, determina si algún elemento de la lista es igual a la suma de los demás elementos de la misma.

12.8 Derivar una definición recursiva de la función $f : Int \longrightarrow [Int] \longrightarrow Bool$, que determina si el k-ésimo elemento de una lista dada de enteros aloja el mínimo valor de la misma.

12.9 Derivar una definición recursiva para la función $consec\_dif : [Int] \to Bool$, la cual recibe una lista con al menos dos elementos y determina si para todo par de elementos consecutivos de la lista se cumple que la diferencia entre ambos valores es igual al índice de la primera posición considerada. Por ejemplo, $consec\_dif.[1, 1, 2, 4, 7] = True$, mientras que $consec\_dif.[1, 1000, 1001] = False$.

12.10 Derivar una función recursiva para la función $g : [Int] \to Int$, que toma una lista de enteros y devuelve el producto de la diferencia de cada uno de sus elementos con la longitud de la lista, satisfaciendo la siguiente especificación:

$$\begin{array}{ll} pre: & \#xs > 0 \\ post: & g.xs = (\prod i : 0 \leq i \leq \#xs : xs.i - \#xs) \end{array}$$

12.11 Derive una definición recursiva de la función $check : [Int] \to Bool$, que dada una lista determina si la cantidad de veces que aparece el mínimo elemento en la lista es igual a la cantidad de ceros de la lista.

12.12 En la sección 12.4 se derivó una definición recursiva para la función $gbal : [Bool] \to Bool$ usando la técnica generalización por abstracción. Otra manera de derivar una definición para esta función es utilizando la técnica modularización, donde se distinguen dos módulos independientes:

$$\begin{array}{ll} pre: & true \\ post: & cantTrue.xs = (\mathcal{N}\, i : 0 \leq i < \#xs : xs.i) \end{array}$$

$$\begin{array}{ll} pre: & true \\ post: & cantFalse.xs = (\mathcal{N}\, i : 0 \leq i < \#xs : \neg xs.i) \end{array}$$

Una vez que se obtienen las definiciones de estas 2 funciones se puede definir *gbal* como:

$$gbal.xs \doteq (cantTrue.xs = cantFalse.xs)$$

Completar esta definición con las definiciones de $cantTrue$ y $cantFalse$ obtenidas por derivación a partir de sus especificaciones.

12.13 Calcule las siguientes funciones según las especificaciones dadas empleando la técnica de modularización. Si fuera necesario, puede replantear la especificación de la función (como en el ejercicio anterior), para calcular la definición de la función requerida según la técnica indicada.

a) $sumult : [Real] \to [Real] \to Bool$
$pre : \quad true$
$post : \quad sumult.xs.ys = ((\sum\ i : 0 \leq i < \#xs : xs.i) = (\prod\ i : 0 \leq i < \#ys : ys.i))$

b) $cpi : [Int] \to Bool$
$pre : \quad true$
$post : \quad cpi.xs = ((\mathcal{N}\, i : 0 \leq i \leq \#xs : \mathbf{par}.(xs.i)) = (\mathcal{N}\, i : 0 \leq i \leq \#xs : \mathbf{impar}.(xs.i)))$

c) $mimax : [Int] \to [Int] \to Bool$
$pre : \quad \#xs > 0 \wedge \#ys > 0$
$post : \quad mimax.xs.ys = ((\text{mín}\ i : 0 \leq i < \#xs : xs.i) > (\text{máx}\ i : 0 \leq i < \#ys : ys.i))$

12.14 Especifique y derive una definición recursiva de la función $h0t1 : [Int] \to Bool$ que dada una lista decide si existe algún cero en ella, o si todos los elementos son iguales al valor uno.

12.15 Derive una definición recursiva para las siguientes funciones:

1. $abr : [Char] \to Nat$, que determina la cantidad de paréntesis izquierdos ("(") existentes en una lista de caracteres.
2. $cerr : [Char] \to Nat$, que determina la cantidad de paréntesis derechos (")") existentes en una lista de caracteres.
3. $ok : [Char] \to Bool$, que determina si la cantidad de paréntesis abiertos y cerrados coinciden.