Trabajo Práctico 3 Derivación con generalización y programas sobre segmentos de listas.

En esta guía se proponen ejercicios que requieren generalización, ya sea reemplazo de constantes por variables o por abstracción. Recordemos que descubrimos la necesidad de generalizar una especificación porque cuando estamos derivando el programa nos encontramos con que la hipótesis inductiva es demasiado rígida para aplicarla.

1. A partir de las siguientes especificaciones expresar en lenguaje natural qué devuelven las funciones, agregarles su tipo y derivarlas:

   a) $\text{psum}.xs = \langle \forall i\ : 0 \leq i \leq \#xs : \ sum.(xs{\uparrow}i) \geq 0 \rangle$

   b) $\text{sum\_ant}.xs = \langle \exists i\ : 0 \leq i < \#xs : \ xs\,!\,i = sum.(xs{\uparrow}i) \rangle$

a)

psum.xs : Todos los segmentos iniciales de la lista xs suman ≥ 0

psum :: [Num] -> Bool

psum.xs = ⟨ ∀ i : 0 ≤ i ≤ #xs : sum.(xs↑i) ≥ 0 ⟩

Hago inducción en listas

caso base xs= [ ]

psum.[ ]

≡ { especificación }

⟨ ∀ i : **0 ≤ i ≤ #[ ]** : sum.([ ]↑i) ≥ 0 ⟩

≡ { def de # y lógica }

⟨ ∀ i : i=0 : sum.([ ]↑i) ≥ 0 ⟩

≡ { rango unitario }

sum(**[ ]↑0)** ≥ 0

≡ { def de ↑ }

sum([ ]) ≥ 0

≡ { def de sum }

0 ≥ 0

≡ { lógica }

True

Caso inductivo

HI : psum.xs = ⟨ ∀ i : 0 ≤ i ≤ #xs : sum.(xs↑i) ≥ 0 ⟩

psum.(x:xs)

≡ { especificación }

⟨ ∀ i : 0 ≤ i ≤ #(x:xs) : sum.((x:xs)↑i) ≥ 0 ⟩

≡ {def de cardinal y lógica}

$\langle \forall\ i : i = 0 \vee\ 1 \leq i \leq \#xs + 1 : sum.((x:xs)\uparrow i) \geq 0 \rangle$

≡ {separación de término}

$\langle \forall\ i : i = 0 : sum.((x:xs)\uparrow i) \geq 0 \rangle \wedge \langle \forall\ i : 1 \leq i \leq \#xs + 1 : sum.((x:xs)\uparrow i) \geq 0 \rangle$

≡ {Rango Unitario y aritmética}

$sum.((x:xs)\uparrow 0) \geq 0 \wedge \langle \forall\ i : 0 \leq i \leq \#xs : sum.((x:xs)\uparrow(i+1)) \geq 0 \rangle$

≡ { def de ↑ }

$sum.((x:xs)\uparrow 0) \geq 0 \wedge \langle \forall\ i : 0 \leq i \leq \#(x:xs)-1 : sum.(x : (xs \uparrow i)) \geq 0 \rangle$

≡ { def de sum }

$sum.((x:xs)\uparrow 0) \geq 0 \wedge \langle \forall\ i : 0 \leq i \leq \#(x:xs)-1 : x + sum.(xs\uparrow i) \geq 0 \rangle$

No puedo llegar a HI… veo con una función mas general, con **especificación**

$gpsum.k.xs = \langle \forall\ i : 0 \leq i \leq \#xs : k + sum.(xs\uparrow i) \geq 0 \rangle$

caso base, xs = [ ]

gpsum.k.[ ]

≡ { especificación }

$\langle \forall\ i : 0 \leq i \leq \#[\,] : k + sum.([\,]\uparrow i) \geq 0 \rangle$

≡ { def de # y lógica }

$\langle \forall\ i : i = 0 : k + sum.([\,]\uparrow i) \geq 0 \rangle$

≡ { rango unitario }

$k + sum.([\,]\uparrow 0) \geq 0$

≡ { def de ↑ }

$k + sum.[\,] \geq 0$

≡ { def de sum }

$k \geq 0$

caso inductivo

HI : $\forall\ k$, $gpsum.k.xs = \langle \forall\ i : 0 \leq i \leq \#xs : k + sum.(xs\uparrow i) \geq 0 \rangle$

gpsum.n.(x:xs)

≡ { especificación }

$\langle \forall\ i : 0 \leq i \leq \#(x:xs) : n + sum.((x:xs)\uparrow i) \geq 0 \rangle$

≡ {def de # y lógica}

$\langle \forall\ i : i = 0 \vee 1 \leq i \leq \#xs + 1 : n + sum.((x:xs)\uparrow i) \geq 0 \rangle$

≡ {partición de rango}

$\langle \forall\ i : i = 0 : n + sum.((x:xs)\uparrow i) \geq 0 \rangle \wedge \langle \forall\ i : 1 \leq i \leq \#xs + 1 : n + sum.((x:xs)\uparrow i) \geq 0 \rangle$

≡ {cambio de variable i = i + 1}

$\langle \forall\ i : i = 0 : n + sum.((x:xs)\uparrow i) \geq 0 \rangle \wedge \langle \forall\ i : 1 \leq i + 1 \leq \#xs + 1 : n + sum.((x:xs)\uparrow(i+1)) \geq 0 \rangle$

≡ {aritmética}

$\langle\ \forall\ i : i = 0 : n + sum.((x:xs)\uparrow i) \geq 0\ \rangle \land \langle\ \forall\ i : 0 \leq i \leq \#xs : n + sum.(\underline{\mathbf{(x:xs)\uparrow(i+1)}}) \geq 0\ \rangle$

≡ { def de ↑ }

$\langle\ \forall\ i : i = 0 : n + sum.((x:xs)\uparrow i) \geq 0\ \rangle \land \langle\ \forall\ i : 0 \leq i \leq \#xs : n + \underline{\mathbf{sum.(x \triangleright (xs \uparrow i))}} \geq 0\ \rangle$

≡ { def de sum }

$\langle\ \forall\ i : i = 0 : n + sum.((x:xs)\uparrow i) \geq 0\ \rangle \land \langle\ \forall\ i : 0 \leq i \leq \#xs : \underline{\mathbf{n + (x + sum.(xs\uparrow i))}} \geq 0\ \rangle$

≡ { conmutatividad de + }

$\langle\ \forall\ i : i = 0 : n + sum.((x:xs)\uparrow i) \geq 0\ \rangle \land \underline{\mathbf{\langle\ \forall\ i : 0 \leq i \leq \#xs : (n + x) + sum.(xs\uparrow i) \geq 0\ \rangle}}$

≡ { HI para **k = (n+x)** }

$\langle\ \forall\ i : \underline{\mathbf{i = 0}} : n + sum.((x:xs)\uparrow i) \geq 0\ \rangle \land$ gpsum.(n+x).xs

≡ {Rango Unitario}

$n + sum.(\underline{\mathbf{(x:xs)\uparrow 0}}) \land$ gpsum.(n+x).xs

≡ { def de ↑ }

$n + \underline{\mathbf{sum.[\ ]}} \geq 0 \land$ gpsum.(n+x).xs

≡ { def de sum }

n ≥ 0 ∧ gpsum.(n+x).xs

=>

```
gpsum :: Num -> [Num] -> Bool
gpsum.n.[ ] = n ≥ 0
gpsum.n.(x:xs) = n ≥ 0 ∧ gpsum.(n+x).xs

psum.xs = gpsum.0.xs
```

b)

sum_ant.xs : Si existe un elemento que es igual a la suma de todos los elementos anteriores.

sum_ant.xs :: [Int] -> Bool

sum_ant.xs = $\langle\ \exists\ i : 0 \leq i < \#xs : xs\ !i = sum.(xs\uparrow i)\ \rangle$

Caso base, xs = [ ]

sum_ant.[ ]

≡ {especificación}

$\langle\ \exists\ i : 0 \leq i < \underline{\mathbf{\#[\ ]}} : [\ ]\ !i = sum.([\ ]\uparrow i)\ \rangle$

≡ {def de # y lógica}

$\langle\ \exists\ i : \underline{\mathbf{False}} : [\ ]\ !i = sum.([\ ]\uparrow i)\ \rangle$

≡ {rango vacío}

False

caso inductivo,

HI : sum_ant.xs = ⟨ ∃ i : 0 ≤ i < #xs : xs !i = sum.(xs↑i) ⟩

sum_ant.(x:xs)

≡ {especificación}

⟨ ∃ i : 0 ≤ i < **#(x:xs)** : (x : xs) !i = sum.((x : xs)↑i) ⟩

≡ {def de #}

⟨ ∃ i : **0 ≤ i < #xs+1** : (x : xs) !i = sum.((x : xs)↑i) ⟩

≡ {Logica}

⟨ ∃ i : **i = 0** ∨ **1 ≤ i < #xs+1** : (x:xs) !i = sum.((x : xs)↑i) ⟩

≡ {separación de término}

⟨ ∃ i : i = 0 : (x:xs) !i = sum.((x:xs)↑i) ⟩ ∨ ⟨ ∃ i : **1 ≤ i < #xs + 1** : **(x : xs) ! i = sum.((x : xs)↑i)** ⟩

≡ {cambio de variable **i = i + 1**}

⟨ ∃ i : i = 0 : (x:xs) !i = sum.((x:xs)↑i) ⟩ ∨ ⟨ ∃ i : **1 ≤ i + 1 < #xs + 1 :** (x : xs) ! (i+1) = sum.((x:xs) ↑ (i+1)) ⟩

≡ {aritmética resto 1 en cada término del rango}

⟨ ∃ i : i = 0 : (x:xs) !i = sum.((x:xs)↑i) ⟩ ∨ ⟨ ∃ i : 0 ≤ i < #xs : **(x:xs) !(i+1)** = sum.((x : xs)↑(i+1)) ⟩

≡ {def de !}

⟨ ∃ i : i = 0 : (x:xs) !i = sum.((x:xs)↑i) ⟩ ∨ ⟨ ∃ i : 0 ≤ i < #xs : xs! i = sum.(**(x : xs) ↑ (i + 1)**) ⟩

≡ {def de ↑}

⟨ ∃ i : i = 0 : (x:xs) !i = sum.((x:xs)↑i) ⟩ ∨ ⟨ ∃ i : 0 ≤ i < #xs : xs! i = **sum.(x : (xs ↑ (i + 1)))** ⟩

≡ {def de sum}

⟨ ∃ i : i = 0 : (x:xs) !i = sum.((x:xs)↑i) ⟩ ∨ **⟨ ∃ i : 0 ≤ i < #xs : xs! i = x + sum.(xs↑(i+1)) ⟩**

No puedo llegar a la HI, por lo que busco una función mas general

sum_ant.xs = ⟨ ∃ i : 0 ≤ i < #xs : xs !i = sum.(xs↑i) ⟩

gsum_ant.k.xs = ⟨ ∃ i : 0 ≤ i < #xs : xs !i + k = sum.(xs↑i) ⟩

Caso base, xs = [ ]

gsum_ant.k.[ ]

≡ { especificación }

⟨ ∃ i : 0 ≤ i < **#[ ]** : [ ] !i +k = sum.([ ]↑i) ⟩

≡ { def de # y lógica }

⟨ ∃ i : **False** : [ ] !i +k = sum.([ ]↑i) ⟩

≡ { rango vacío }

False

Caso recursivo,

HI : $\boldsymbol{\forall k ,\ \text{gsum\_ant}.k.xs = \langle \exists\, i : 0 \le i < \#xs : xs\,!i + k = \text{sum}.(xs{\uparrow}i) \rangle}$

$\text{gsum\_ant}.n.(x{:}xs)$

$\equiv$ { especificación }

$\langle \exists\, i : 0 \le i < \underline{\mathbf{\#(x{:}xs)}} : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle$

$\equiv$ { def de # }

$\langle \exists\, i : \underline{\mathbf{0 \le i < \#xs+1}} : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle$

$\equiv$ { separación de término }

$\langle \exists\, i : i = 0 : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle \vee \langle \exists\, i : \underline{\mathbf{1 \le i < \#xs+1}} : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle$

$\equiv$ { cambio de variable i = i + 1}

$\langle \exists\, i : i = 0 : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle \vee \langle \exists\, i : \underline{\mathbf{1 \le i + 1 < \#xs+1}} : (x : xs)\,!\,(i + 1) + n = \text{sum}.((x{:}xs) \uparrow (i + 1) \rangle$

$\equiv$ {aritmética}

$\langle \exists\, i : i = 0 : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle \vee \langle \exists\, i : 0 \le i < \#xs : \underline{\mathbf{(x : xs)\,!\,(i + 1)}} + n = \text{sum}.((x{:}xs) \uparrow (i + 1) \rangle$

$\equiv$ { def de ! }

$\langle \exists\, i : i = 0 : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle \vee \langle \exists\, i : 0 \le i < \#xs : xs!\, i + n = \text{sum}.\underline{\mathbf{((x{:}xs){\uparrow}(i+1))}} \rangle$

$\equiv$ { def de $\uparrow$ }

$\langle \exists\, i : i = 0 : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle \vee \langle \exists\, i : 0 \le i < \#xs : xs!\, i + n = \underline{\mathbf{\text{sum}.(x{:}(xs{\uparrow}(i+1)))}} \rangle$

$\equiv$ { def de sum }

$\langle \exists\, i : i = 0 : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle \vee \langle \exists\, i : 0 \le i < \#xs : \underline{\mathbf{xs!\, i + n = x + \text{sum}.(xs{\uparrow}(i+1))}} \rangle$

$\equiv$ { aritmética }

$\langle \exists\, i : i = 0 : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle \vee \underline{\boldsymbol{\langle \exists\, i : 0 \le i < \#xs : xs!\, i + (n - x) = \text{sum}.(xs{\uparrow}(i+1)) \rangle}}$

$\equiv$ { HI con **k = n-x** }

$\langle \exists\, i : \underline{\mathbf{i = 0}} : (x{:}xs)\,!i + n = \text{sum}.((x{:}xs){\uparrow}i) \rangle \vee \text{gsum\_ant}.(n-x).xs$

$\equiv$ {Rango Unitario}

$\underline{\mathbf{(x{:}xs)\,!0}} + n = \text{sum}.((x{:}xs){\uparrow}0) \vee \text{gsum\_ant}.(n-x).xs$

$\equiv$ { def de ! }

$x + n = \text{sum}.(\underline{\mathbf{(x{:}xs){\uparrow}0}}) \vee \text{gsum\_ant}.(n-x).xs$

$\equiv$ { def de $\uparrow$ }

$x + n = \underline{\mathbf{\text{sum}.(x{:}xs)}} \vee \text{gsum\_ant}.(n-x).xs$

$\equiv$ { def de sum }

$\underline{\mathbf{x + n = x + \text{sum}\ xs}} \vee \text{gsum\_ant}.(n-x).xs$

$\equiv$ { aritmética }

n = sum xs ⋁ gsum_ant.(n-x).xs

=>

```
gsum_ant :: Num -> [Num] -> Bool
gsum_ant.n.[ ] = False
gsum_ant.n.(x:xs) = n = sum xs ⋁ gsum_ant.(n-x).xs
sum_ant.xs = gsum_ant.0.xs
```

2. Especificar formalmente utilizando cuantificadores cada una de las siguientes funciones descriptas informalmente. Luego, *derivar* soluciones algorítmicas para cada una.

   a) esCuad : $Nat \to Bool$, dado un natural determina si es el cuadrado de un número.

   b) sumanOcho : $[Num] \to Nat$, dada una lista cuenta la cantidad de prefijos que suman 8.

   Reflexión: La primera parte de este ejercicio no parece tener que ver con generalización: efectivamente las especificaciones formales que surgirán serán las habituales. Sin embargo, cuando intentemos derivar los programas nos encontraremos con la necesidad de generalizar. Está bien que descubramos la necesidad de generalizar al derivar y que no intentemos anticiparnos.

   Recuerden que es una buena práctica *testear* las especificaciones que hicieron con ejemplos concretos. Por ejemplo, esCuad,4 $\equiv True$, esCuad,5 $\equiv False$. Para el segundo: sumanOcho.$[8, 0, -1, 1] = 3$, pero sumanOcho.$[8, 0, -1] = 2$. Para cerciorarse que las especificaciones son correctas, *evaluamos* las especificaciones en esos argumentos; es decir, manipulamos las expresiones hasta encontrar el valor final.

a) esCuad : Nat → Bool, dado un natural determina si es el cuadrado de un número.

esCuad.n = ⟨ ∃ x : 0 ≤ x ≤ n : x*x = n ⟩

inducción

paso inductivo

HI : esCuad.n = ⟨ ∃ x : 0 ≤ x ≤ n : x*x = n ⟩

<u>esCuad.(n+1)</u>

= { especificación }

⟨ ∃ x : **<u>0 ≤ x ≤ (n+1)</u>** : x*x = (n+1) ⟩

= { lógica }

⟨ ∃ x : **<u>0 ≤ x ≤ n ⋁ x = n+1</u>** : x*x = (n+1) ⟩

= { partición de rango }

⟨ ∃ x : 0 ≤ x ≤ n : **<u>x*x = (n+1)</u>** ⟩ ⋁ ⟨ ∃ x : x = n+1 : x*x = (n+1) ⟩

= no puedo aplicar HI

=> Hago una **generalización**, **gEsCuad.m.n = ⟨ ∃ x : 0 ≤ x ≤ n : x*x - m = n ⟩**

Inducción, empiezo por paso inductivo.

HI : gEsCuad.m.n = ⟨ ∃ x : 0 ≤ x ≤ n : x*x - m = n ⟩

<u>gEsCuad.m.(n+1)</u>

= { especificación }

〈 ∃ x : **<u>0 ≤ x ≤ (n+1)</u>** : x*x - m = (n+1) 〉

= { lógica }

〈 ∃ x : **<u>0 ≤ x ≤ n ∨ x = n+1</u>** : x*x - m = (n+1) 〉

= { partición de rango }

〈 ∃ x : 0 ≤ x ≤ n : **<u>x*x - m = (n+1</u>**)〉 ∨ 〈 ∃ x : x = n + 1 : x*x - m = (n+1) 〉

= { aritmética }

<u>〈 ∃ x : 0 ≤ x ≤ n : x*x - (m+1) = n〉</u> ∨ 〈 ∃ x : x = n+1 : x*x - m = (n+1) 〉

= { HI para gEsCuad.(m+1).n }

gEsCuad.(m+1).n ∨ **<u>〈 ∃ x : x = n+1 : x*x - m = (n+1) 〉</u>**

= { rango unitario }

gEsCuad.(m+1).n ∨ **<u>(n+1)*(n+1) - m = n+1</u>**

= { aritmética }

gEsCuad.(m+1).n ∨ n*n + n = m

Caso base

<u>gEsCuad.m.0</u>

= { especificación }

〈 ∃ x : **<u>0 ≤ x ≤ 0</u>** : x*x - m = 0 〉

= { lógica y rango unitario }

0*<u>0 - m = 0</u>

= { aritmética }

m = 0

gEsCuad.m.0 = (m=0)

gEsCuad.m.(n+1) = gEsCuad.(m+1).n ∨ (n*n + n = m)

esCuad.n = gEsCuad.0.n

Ej:

esCuad.4 = gEsCuad.0.4

<u>g.EsCuad.0.4</u>

= { caso recursivo m = 0 , (n+1) = (3+1) = 4}

<u>g.EsCuad.1.3</u> ∨ (3*3 + 3 = 0)

= { caso recursivo m = 1 , (n+1) = (2+1) = 3}

(**<u>g.EsCuad.2.2</u>** ∨ (2*2 + 2 = 1)) ∨ (3*3 + 3 = 0)

= { caso recursivo m = 2 , (n+1) = (1+1) = 2}

(**<u>g.EsCuad.3.1</u>** ∨ (1*1 + 1 = 2) ∨ (2*2 + 2 = 1)) ∨ (3*3 + 3 = 0)

= { caso recursivo m = 3 , (n+1) = (0+1) = 1}

(**<u>g.EsCuad.4.0</u>** ∨ (0*0 + 0 = 3) ∨ (1*1 + 1 = 2) ∨ (2*2 + 2 = 1)) ∨ (3*3 + 3 = 0)

= { caso base m = 4 }

<u>4=0</u> ∨ **<u>(0*0 + 0 = 3)</u>** ∨ **<u>(1*1 + 1 = 2)</u>** ∨ **<u>(2*2 + 2 = 1)</u>** ∨ **<u>(3*3 + 3 = 0)</u>**

= { lógica/ aritmética }

<u>False ∨ False ∨ True ∨ False ∨ False</u>

= { absorbente de ∨ }

<u>True</u>

=> esCuad.4 = True

b)

dada una lista cuenta la cantidad de prefijos que suman 8

sumanOcho : [Núm] -> Nat

sumanOcho.xs = 〈 N as,bs : xs=as++bs : sum.as=8 〉

Caso inductivo

HI : sumanOcho.xs = 〈 N as,bs : xs = as++bs : sum.as=8 〉

<u>sumanOcho.(x▸xs)</u>

= { especificación }

〈 N as,bs : (x▸xs) = **<u>as++bs</u>** : sum.as = 8 〉

= {tercero excluido }

〈 N as,bs : (x ▹ xs) = as ⧺ bs ∧ **<u>True</u>** : sum.as = 8 〉

≡ { sustituimos True ≡ as = [ ] ∨ as ≠ [ ] }

〈 N as,bs : (x ▹ xs) = **<u>as ⧺ bs ∧ (as = [ ] ∨ as ≠ [ ])</u>** : sum.as = 8 〉

= { distributividad }

〈 N as,bs : **<u>((x▸xs) =as++bs ∧ as = [ ]) ∨ ((x▸xs) =as++bs ∧ as ≠ [ ])</u>** : sum.as=8 〉

= { partición de rango }

〈 N as,bs : (x ▸ xs) = as++bs ∧ as = [ ] : sum.as = 8 〉 +

〈 N as,bs : (x ▸ xs) =as++bs ∧ **<u>as ≠ [ ]</u>** : sum.as = 8 〉

= { como as ≠ [ ], puedo hacer cambio de variable **as** ← **(a▸as)** }

〈 N as,bs : ((x▸ xs) =as++bs ∧ as = [ ]) : sum.as=8 〉 +

〈 N a,as,bs : ((x ▸xs) =(a ▸as)++bs ∧ **<u>(a▸as) /= [ ])</u>** : sum.as=8 〉

= { lógica y neutro }

〈 N as,bs : ((x ▸ xs) =as++bs ∧ as = [ ]) : sum.as=8 〉 +

〈 N a,as,bs : **<u>((x ▸ xs) =(a ▸ as)++bs</u>** : sum.(a▸as)=8 〉

= { def de ++ }

〈 N as,bs : ((x ▸ xs) =as++bs ∧ as = [ ]) : sum.as=8 〉 +

〈 N a,as,bs : **<u>((x ▸xs) = a ▸ (as++bs)</u>** : sum.(a▸as)=8 〉

= { prop de constructores y listas }

⟨ N as,bs : ((x▸xs) =as++bs ∧ as = [ ]) : sum.as = 8 ⟩ +

⟨ N a,as,bs : **x = a** ∧ xs = (as++bs): sum.(**a** ▸as) = 8 ⟩

= { eliminación de variable }

⟨ N as,bs : ((x▸xs) =as++bs ∧ as = [ ]) : sum.as=8 ⟩ +

⟨ N as,bs : xs = (as++bs) : **sum.(x ▸ as) = 8** ⟩

= { def de sum }

⟨ N as,bs : ((x▸xs) =as++bs ∧ as = [ ]) : sum.as=8 ⟩ +

⟨ N as,bs : xs=(as++bs): **x + sum.as=8** ⟩

hago especificación

gSumanOcho.k.xs = ⟨ N as,bs : xs=as++bs : k + sum.as=8 ⟩

Caso inductivo, hago los mismos pasos que sumanOcho.xs

gSumanOcho.k.(x▸xs)

…

⟨ N as,bs : ((x▸xs) =as++bs ∧ as = [ ]) : k + sum.as=8 ⟩ +

⟨ N as,bs : xs=(as++bs): k + x+ sum.as=8 ⟩

= { HI para gSumanOcho.(k+x).xs }

⟨ N as,bs : ((x▸xs) = **as**++bs ∧ **as = [ ]**) : k + sum.as=8 ⟩ + gSumanOcho.(k+x).xs

= { eliminación de variable }

⟨ N bs : (x▸xs) = bs : **k + sum.[ ] =8** ⟩ + gSumanOcho.(k+x).xs

= { def de sum y aritmética }

⟨ N bs : **(x▸xs) = bs** : k = 8 ⟩ + gSumanOcho.(k+x).xs

= { rango unitario }

(k = 8 → 1

☐ ¬ k = 8 → 0

) + gSumanOcho.(k+x).xs

Caso base,

gSumanOcho.k.[ ]

= { especificación }

⟨ N as,bs : **[ ] = as++bs** : k + sum.as=8 ⟩

= { prop de ++ }

⟨ N **as**,bs : **as = [ ]** ∧ bs = [ ] : k + sum.as=8 ⟩

= { eliminación de variable }

⟨ N bs : bs = [ ] : **k + sum.[ ]=8** ⟩

= { def de sum y aritmética }

⟨ N bs : **<u>bs = [ ]</u>** : k = 8 ⟩

= { rango unitario }

(k = 8 → 1

☐ ¬ k =8 → 0

)

=>

gSumanOcho.k.[ ] = (k =8 → 1

☐ ¬ k =8 → 0)

)

gSumanOcho.k.(x▸xs) = (k =8 → 1

☐ ¬ k =8 → 0

) + gSumanOcho.(k+x).xs

sumanOcho.xs = gsumanOcho.0.xs

3. Expresar en lenguaje natural cada una de las siguientes expresiones; para ello primero calcular los rangos, ya sea como conjunto de tuplas o una tabla, y evaluar las expresiones para $xs = [9, -5, 1, -3]$ e $ys = [9, -5, 3]$.

   a) $\langle \forall\, as, bs \ : xs = as +\!\!+ bs : \ sum.as \geq 0 \,\rangle$

   b) $\langle \text{Min} \ \ as, bs, cs \ : xs = as +\!\!+ bs +\!\!+ cs : \ sum.bs \,\rangle$

   c) $\langle \text{N}\, as, b, bs \ : xs = as +\!\!+ (b \triangleright bs) : \ b > sum.bs \,\rangle$

   d) $\langle \text{Max} \ \ as, bs, cs \ : xs = as +\!\!+ bs \wedge ys = as +\!\!+ cs : \ \#as \,\rangle$

   Reflexión: En los rangos que dividen la lista en dos o tres partes es conveniente pensar que la primera parte es un *prefijo* y la última un *sufijo*. Muchos problemas interesantes tienen que ver con encontrar el prefijo (o sufijo) de una lista con cierta propiedad. El último punto nos muestra que con segmentos es fácil especificar que dos listas tengan un segmento en común.

a)

“Para todo segmento inicial de la lista xs se debe cumplir que la suma de sus elementos es mayor o igual a cero”

as	bs	sum as >=0
[9,-5,1,-3]	[ ]	True
[9,-5,1]	[-3]	True
[9,-5]	[1,-3]	True
[9]	[-5,1,-3]	True
[ ]	[9,-5,1,-3]	True

b)

“Obtener el mínimo resultado de la suma de los elementos del segmento intermedio”

as	bs	cs	sum bs
[9,-5,1,-3]	[ ]	[ ]	0
[9,-5,1]	[-3]	[ ]	-3
[9,-5]	[1,-3]	[ ]	-2
[9]	[-5,1,-3]	[ ]	-7
[ ]	[9,-5,1,-3]	[ ]	2
[ ]	[9,-5,1]	[-3]	5
[ ]	[9,-5]	[1,-3]	4
[ ]	[9]	[-5,1,-3]	9
[ ]	[ ]	[9,-5,1,-3]	0
[9,-5]	[1]	[-3]	1
[9]	[-5,1]	[-3]	-4
[9]	[-5]	[1,-3]	-5

c)

“Contar cuantas veces la cabeza del segmento final es mayor a la suma de los demás elementos del segmento”

as	(b:bs)	b>sum bs
[9,-5,1,-3]	No es posible	
[9,-5,1]	-3:[]	Falso
[9,-5]	1:[-3]	Verdad
[9]	-5:[1,-3]	Falso
[ ]	9:[-5,1,-3]	Verdad

d)

“El máximo del largo de un segmento común entre xs e ys”

as	bs	=?	as	cs	#as
[9,-5,1,-3]	[ ]	No	[9,-5,-3]	[ ]	
[9,-5,1]	[-3]	–	–	–	
[9,-5]	[1,-3]	si	[9,-5]	[-3]	2
[9]	[-5,1,-3]	si	[9]	[-5,-3]	1
[ ]	[9,-5,1,-3]	si	[ ]	[9,-5,-3]	0

4. Expresar utilizando cuantificadores las siguientes sentencias del lenguaje natural; entre paréntesis está el nombre de la función.

   a) La lista xs es un segmento inicial de la lista ys (prefijo.$xs.ys$).

   b) La lista xs es un segmento de la lista ys (seg.$xs.ys$).

   c) La lista xs es un segmento final de la lista ys (sufijo.$xs.ys$).

   d) Las listas xs e ys tienen en común un segmento no vacío (segComun.$xs.ys$).

   e) La lista xs posee un segmento que no es ni **prefijo** ni **sufijo** y cuyo mínimo es mayor a los valores del prefijo y del sufijo (hayMeseta.xs).

   f) La lista xs de numeros enteros tiene la misma cantidad de elementos pares e impares (balanceada.xs).

   Reflexión: El último punto es de una naturaleza completamente distinta a los anteriores: valores pares e impares pueden aparecer dispersos en la lista, entonces los segmentos no serán útiles para especificarlo.

a) La lista xs es un segmento inicial de la lista ys (prefijo.xs.ys).

$prefijo.xs.ys \;=\; \langle \exists as, bs : ys = as \;++\; bs : \; xs = \; as \rangle$

b) La lista xs es un segmento de la lista ys (seg.xs.ys).

$seg.xs.ys \;=\; \langle \exists as, bs, cs : \; ys = as \;++\; bs \;++\; cs : \; xs = \; bs \rangle$

$seg'.xs.ys \;=\; \langle \exists as, bs, cs : \; ys = as \;++\; bs : \; prefijo.xs.bs \rangle$

c) La lista xs es un segmento final de la lista ys (sufijo.xs.ys).

$sufijo.xs.ys \;=\; \langle \exists as, bs : \; ys = as \;++\; bs : \; xs = \; bs \rangle$

d) Las listas xs e ys tienen en común un segmento no vacío (segComun.xs.ys).

$segComun.xs.ys \;=\; \langle \exists asx, asy, bs, csx, csy : \; xs = asx \;++\; bs \;++\; csx \;\wedge\; ys \;=\; asy \;++\; bs \;++\; csy :$
$bs \;\neq\; [] \rangle$

$segComun'.xs.ys = \langle \exists as, bs : \; ys \;=\; as \;++\; bs \;\wedge\; bs \neq [] : \; seg'.bs.ys \rangle$

e) La lista xs posee un segmento que no es ni prefijo ni sufijo y cuyo mínimo es mayor a los valores del prefijo y del sufijo (hayMeseta.xs).

$hayMeseta.xs \;=\; \langle \exists as, bs, cs : xs = as \;{+}{+}\; bs \;{+}{+}\; cs \;\wedge as \neq [] \;\wedge cs \neq []: \; min.bs \; > max.as$
$\wedge \; min.bs \; > max.cs \rangle$

f) La lista xs de numeros enteros tiene la misma cantidad de elementos pares e impares (balanceada.xs).

$balanceada.xs \;=\; \langle Ni: 0 \leq i < \#xs: (xs!\, i)\, `mod`\, 2 \;=\; 0 \rangle \;=\; \langle Ni: 0 \leq i < \#xs: (xs!\, i)\, `mod`\, 2 \;=\; 1 \rangle$

5. Derivar funciones recursivas para prefijo $(4a)$ y seg $(4b)$.
Reflexión: Al derivar seg tenga presente la especificación de prefijo. ¿Qué relación hay entre seg y prefijo?

a)

prefijo.xs.as = 〈 ∃ as,bs : ys = as ++ bs : xs = as 〉

prefijo :: [a] → [a] → Bool

CASO BASE: ys = []

prefijo.xs.[ ]

≡ {especificación}

〈 ∃ as,bs : **[ ] = as ++ bs** : xs = as 〉

≡ {prop de concatenación}

〈 ∃ as,bs : **as = [ ]** ∧ **bs = [ ]** : xs = as 〉

≡ {eliminación de variable as = [ ] }

〈 ∃ bs : **bs = [ ]** : xs = [ ] 〉

≡ {termino constante}

xs = [ ]

Segundo caso base : xs = [ ]

prefijo.[ ]. ys

≡ {especificación}

〈 ∃ as,bs : ys = **as ++ bs** : [ ] = as 〉

≡ {lógica}

〈 ∃ as,bs : **ys = as ++ bs** ∧ **(as = [ ]** ∨ **as ≠ [ ])** : [ ] = as 〉

≡ {distributiva y partición de rango}

〈 ∃ as,bs : **ys = as ++ bs** ∧ **as = [ ]** : [ ] = as 〉∨ 〈 ∃ as,bs : ys = as ++ bs ∧ as ≠ [ ] : [ ] = as 〉

≡ {eliminación de variable}

〈 ∃ bs : ys = **[ ] ++ bs** : [ ] = [ ] 〉∨ 〈 ∃ as,bs : ys = as ++ bs ∧ as ≠ [ ] : [ ] = as 〉

≡ {propiedad de concatenación}

〈 ∃ bs : ys = bs : **[ ] = [ ]** 〉∨ 〈 ∃ as,bs : ys = as ++ bs ∧ as ≠ [ ] : [ ] = as 〉

≡ {término constante}

[ ] = [ ] ∨ 〈 ∃ as,bs : ys = as ++ bs ∧ as ≠ [ ] : [ ] = as 〉

≡ {lógica}

True ∨ 〈 ∃ as,bs : ys = as ++ bs ∧ **as ≠ [ ]** : [ ] = as 〉

≡ {cambio de variable as = (a ▹ as)

True ∨ 〈 ∃ as,bs : ys = (a ▹ as) ++ bs ∧ **(a ▹ as) ≠ [ ]** : [ ] = (a ▹ as) 〉

≡ {lógica y elemento neutro del ∧ }

True ∨ 〈 ∃ as,bs : ys = **(a ▹ as) ++ bs** : [ ] = (a ▹ as) 〉

≡ {definición de concatenar}

True ∨ 〈 ∃ as,bs : ys = a ▹ (as ++ bs) : **[ ] = (a ▹ as)** 〉

≡ {término constante}

True ∨ False

≡ {absorción de V}

True

Derivacion:

H.I : prefijo.xs.ys = 〈 ∃ as,bs : ys = as ++ bs : xs = as 〉

Paso inductivo: (x▹xs) (y▹ys)

prefijo.(x▹xs).(y▹ys)

≡ {especificación}

〈 ∃ as,bs : (y ▹ ys) = as ++ bs : (x ▹ xs) = as 〉

≡ {lógica}

〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ (as = [ ] ∨ as ≠ [ ]) : (x ▹ xs) = as 〉

≡ {distributividad de ∧ con ∨ y partición de rango}

〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ as = [ ] : (x ▹ xs) = as 〉 ∨

〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ **as ≠ [ ]**) : (x ▹ xs) = as 〉

≡ {cambio de variable as = (a ▹ as)}

〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ as = [ ] : (x ▹ xs) = as 〉 ∨

〈 ∃ as,bs : (y ▹ ys) = (a ▹ as) ++ bs ∧ (**a ▹ as) ≠ [ ]**) : (x ▹ xs) = (a ▹ as) 〉

≡ {lógica}

〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ as = [ ] : (x ▹ xs) = as 〉 ∨

〈 ∃ as,bs : (y ▹ ys) = **(a ▹ as) ++ bs** : (x ▹ xs) = (a ▹ as) 〉

≡ {propiedades de concatenación}

〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ as = [ ] : (x ▹ xs) = as 〉 ∨
〈 ∃ as,bs : **(y ▹ ys) = a ▹ (as ++ bs)** : (x ▹ xs) = (a ▹ as) 〉
≡ {propiedad de constructores}
〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ as = [ ] : (x ▹ xs) = as 〉 ∨
〈 ∃ as,bs : **y = a** ∧ ys = as ++ bs : (x ▹ xs) = (**a** ▹ as) 〉
≡ {eliminación de variable}
〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ as = [ ] : (x ▹ xs) = as 〉 ∨
〈 ∃ as,bs : ys = as ++ bs : **(x ▹ xs) = (y ▹ as)** 〉
≡ {prop de constructores}
〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ as = [ ] : (x ▹ xs) = as 〉 ∨
〈 ∃ as,bs : ys = as ++ bs : x = y ∧ xs = as) 〉
≡ {distributividad }
〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ as = [ ] : (x ▹ xs) = as 〉 ∨
〈 ∃ as,bs : ys = as ++ bs : xs = as) 〉∧ x = y
≡ {h.i.}
〈 ∃ as,bs : (y ▹ ys) = as ++ bs ∧ **as = [ ]** : (x ▹ xs) = as 〉 ∨ prefijo.xs.ys ∧ x = y
≡ {eliminación de variable}
〈 ∃ as,bs : (y ▹ ys) = [ ]++ bs ∧ as = [ ] : **(x ▹ xs) = [ ]** 〉 ∨ prefijo.xs.ys ∧ x = y
≡ {término constante}
(x ▹ xs) = [ ] ∨ prefijo.xs.ys ∧ x = y
≡ {prop de constr}
false ∨ prefijo.xs.ys ∧ x = y
≡ {elemento absorbente V}
prefijo.xs.ys ∧ x = y

El programa es:

prefijo.xs.as = 〈 ∃ as,bs : ys = as ++ bs : xs = as 〉
prefijo :: [a] → [a] → Bool
prefijo.xs.[ ] = xs = [ ]
prefijo. [ ].ys = True
prefijo.(x ▹ xs).(y ▹ ys) = x = y ∧ prefijo.xs.ys

b)
seg.xs.ys = 〈 ∃ as, bs, cs : ys = as ++ bs ++ cs : xs = bs 〉
seg.xs.ys : [A] → [A] → Bool

inducción sobre ys

Caso recursivo

Hipótesis Inductiva: $\mathbf{seg.xs.ys} = \langle \exists\ \mathbf{as, bs, cs : ys = as ++ bs ++ cs : xs = bs} \rangle$

$\underline{\mathbf{seg.xs.(y \triangleright ys)}}$

$\equiv$ { especificación }

$\langle \exists\ as, bs, cs : \underline{\mathbf{y \triangleright ys = as ++ bs ++ cs}} : xs = bs \rangle$

$\equiv$ { lógica }

$\langle \exists\ as, bs, cs : \underline{\mathbf{y \triangleright ys = as ++ bs ++ cs \wedge (as = [] \vee as \neq [])}} : xs = bs \rangle$

$\equiv$ { distributiva $\wedge$ con él $\vee$ y partición de rango }

$\langle \exists\ as, bs, cs : y \triangleright ys = as ++ bs ++ cs \wedge as = [] : xs = bs \rangle \vee$

$\langle \exists\ as, bs, cs : y \triangleright ys = \underline{\mathbf{as}} ++ bs ++ cs \wedge \underline{\mathbf{as \neq []}} : xs = bs \rangle$

$\equiv$ { cambio de variable, $\mathbf{as} \leftarrow \mathbf{a \triangleright as}$ }

$\langle \exists\ as, bs, cs : y \triangleright ys = as ++ bs ++ cs \wedge as = [] : xs = bs \rangle \vee$

$\langle \exists\ as, bs, cs : y \triangleright ys = a \triangleright as ++ bs ++ cs \wedge \underline{\mathbf{a \triangleright as \neq []}} : xs = bs \rangle$

$\equiv$ { lógica }

$\langle \exists\ as, bs, cs : y \triangleright ys = as ++ bs ++ cs \wedge as = [] : xs = bs \rangle \vee$

$\langle \exists\ as, bs, cs : y \triangleright ys = \underline{\mathbf{a \triangleright as ++ bs ++ cs}} : xs = bs \rangle$

$\equiv$ { propiedades de concatenar}

$\langle \exists\ as, bs, cs : y \triangleright ys = as ++ bs ++ cs \wedge as = [] : xs = bs \rangle \vee$

$\langle \exists\ as, bs, cs : \underline{\mathbf{y \triangleright ys = a \triangleright (as ++ bs ++ cs)}} : xs = bs \rangle$

$\equiv$ { propiedades de listas }

$\langle \exists\ as, bs, cs : y \triangleright ys = as ++ bs ++ cs \wedge as = [] : xs = bs \rangle \vee$

$\langle \exists\ as, bs, cs : \underline{\mathbf{y = a}} \wedge ys = as ++ bs ++ cs : xs = bs \rangle$

$\equiv$ { eliminación de variable }

$\langle \exists\ as, bs, cs : y \triangleright ys = as ++ bs ++ cs \wedge as = [] : xs = bs \rangle \vee$

$\underline{\langle \exists\ \mathbf{as, bs, cs : ys = as ++ bs ++ cs : xs = bs} \rangle}$

$\equiv$ { H.I. }

$\langle \exists\ as, bs, cs : y \triangleright ys = \underline{\mathbf{as}} ++ bs ++ cs \wedge \underline{\mathbf{as = []}} : xs = bs \rangle \vee seg.xs.ys$

$\equiv$ { eliminación de variable }

$\langle \exists\ bs, cs : y \triangleright ys = \underline{\mathbf{[\ ] ++ bs ++ cs}} : xs = bs \rangle \vee seg.xs.ys$

$\equiv$ { propiedades de listas }

$\underline{\langle \exists\ \mathbf{bs, cs : y \triangleright ys = bs ++ cs : xs = bs} \rangle} \vee seg.xs.ys$

$\equiv$ { Modularización, especificación de prefijo }

$\mathbf{prefijo.xs.(y \triangleright ys)} \vee \mathbf{seg.xs.ys}$

Caso base

<u>seg.xs.[]</u>

≡ { especificación }

⟨ ∃ as, bs, cs : **<u>[] = as ++ bs ++ cs</u>** : xs = bs ⟩

≡ { propiedades de listas }

⟨ ∃ as, bs, cs : as = [] ∧ **<u>bs = []</u>** ∧ cs = [] : xs = **<u>bs</u>** ⟩

≡ { eliminación de variable }

⟨ ∃ as, bs, cs : as = [] ∧ cs = [] : **<u>xs = [ ]</u>**⟩

≡ {término constante}

<u>xs = []</u>

El programa resulta

seg.xs.ys : [A] → [A] → Bool

seg.xs.[] ≐ xs = []

seg.xs.(y▹ys) ≐ prefijo.xs.(y▹ys) ∨ seg.xs.ys

prefijo : [A] → [A] → Bool

prefijo.xs.[] ≐ xs = []

prefijo.[].(y▹ys) ≐ True

prefijo.(x▹xs).(y▹ys) ≐ prefijo.xs.ys ∧ x = y

8. Derivar funciones para:

   a) Suma mínima de un segmento:

$$\text{sumaMin}.xs = \langle \text{Min}\ \ as, bs, cs\ : xs = as +\!\!+ bs +\!\!+ cs :\ \ sum.bs \rangle$$

   b) Máxima longitud de elementos iguales a e:

$$\text{maxLongEq}.e.xs = \langle \text{Max}\ \ as, bs, cs\ : xs = as +\!\!+ bs +\!\!+ cs \wedge iga.e.bs :\ \ \#bs \rangle$$

   donde iga es la función del ejercicio *2.b* del práctico 2.

a)

sumaMin.xs = ⟨Min as, bs, cs : xs = as ++ bs ++ cs : sum.bs ⟩

Caso recursivo

Hipótesis Inductiva

sumaMin.xs = ⟨Min as, bs, cs : xs = as ++ bs ++ cs : sum.bs ⟩

sumaMin.(x▹xs)

≡ { especificación }

⟨Min as, bs, cs : x▹xs = as ++ bs ++ cs : sum.bs ⟩

≡ { lógica }

⟨Min as, bs, cs : x▹xs = **as ++ bs ++ cs ∧ (as = [ ] ∨ as ≠ [ ])** : sum.bs ⟩

≡ { distributiva ∧ con ∨ }

⟨Min as, bs, cs : **(x▹xs = as ++ bs ++ cs ∧ as = [ ]) ∨ (x▹xs = as ++ bs ++ cs ∧ as ≠ [ ])** : sum.bs ⟩

≡ { partición de rango }

⟨Min as, bs, cs : x▹xs = as ++ bs ++ cs ∧ as = [] : sum.bs ⟩ min

⟨Min as, bs, cs : x▹xs = as ++ bs ++ cs ∧ **as ≠ [ ]** : sum.bs ⟩

≡ { cambio de variable, as ← a▹as }

⟨Min a, as, bs, cs : x ▹ xs = as ++ bs ++ cs ∧ as = [ ] : sum.bs ⟩ min

⟨Min as, bs, cs : x ▹ xs = a ▹ as ++ bs ++ cs ∧ **a ▹ as ≠ [ ]** : sum.bs ⟩

≡ {lógica y elem. neutro del ∧}

⟨Min a, as, bs, cs : x ▹ xs = as ++ bs ++ cs ∧ as = [ ] : sum.bs ⟩ min

⟨Min as, bs, cs : x ▹ xs = **a ▹ as ++ bs ++ cs** : sum.bs ⟩

≡ { propiedades de listas }

⟨Min a, as, bs, cs : x ▹ xs = as ++ bs ++ cs ∧ as = [ ] : sum.bs ⟩ min

⟨Min as, bs, cs : **x ▹ xs = a ▹ (as ++ bs ++ cs)** : sum.bs ⟩

≡ { propiedades de listas }

⟨Min a, as, bs, cs : x▹xs = as ++ bs ++ cs ∧ as = [] : sum.bs ⟩ min

⟨Min as, bs, cs : **x = a** ∧ xs = as ++ bs ++ cs : sum.bs ⟩

≡ { eliminación de variable a }

⟨Min as, bs, cs : x▹xs = as ++ bs ++ cs ∧ as = [] : sum.bs ⟩ min

⟨Min as, bs, cs : xs = as ++ bs ++ cs : sum.bs ⟩

≡ { H.I. }

⟨Min **as,** bs, cs : x▹xs = **as** ++ bs ++ cs ∧ **as = [ ]** : sum.bs ⟩ min sumaMin.xs

≡ { eliminación de variable }

⟨Min bs, cs : x ▹ xs = **[ ] ++ bs ++ cs** : sum.bs ⟩ min sumaMin.xs

≡ { propiedades de listas }

⟨Min bs, cs : x▹xs = bs ++ cs : sum.bs ⟩ min sumaMin.xs

Modularización

Específico

<u>msumaMin.xs = ⟨Min bs, cs : xs = bs ++ cs : sum.bs ⟩</u>

Caso base xs = [ ]

<u>msumaMin.[ ]</u>

≡ { especificación }

⟨Min bs, cs : **<u>[ ] = bs ++ cs</u>** : sum.bs ⟩

≡ { propiedades de listas }

⟨Min bs, cs : **<u>bs = [ ] ∧ cs = [ ]</u>** : sum.bs ⟩

≡ { eliminación de variable cs, rango unitario }

<u>sum.[]</u>

≡ { definición de sum }

0

Caso recursivo

Hipótesis Inductiva: msumaMin.xs = ⟨Min bs, cs : xs = bs ++ cs : sum.bs ⟩

msumaMin.(x▹xs)

≡ { especificación }

⟨Min bs, cs : **<u>x▹xs = bs ++ cs</u>** : sum.bs ⟩

≡ { lógica }

⟨Min bs, cs : **<u>x▹xs = bs ++ cs ∧ (bs = [] ∨ bs ≠ [])</u>** : sum.bs ⟩

≡ { distributiva ∧ con ∨ }

⟨Min bs, cs : **<u>(x▹xs = bs ++ cs ∧ bs = []) ∨ (x▹xs = bs ++ cs ∧ bs ≠ [])</u>** : sum.bs ⟩

≡ { partición de rango }

⟨Min bs, cs : x ▹ xs = bs ++ cs ∧ bs = [ ] : sum.bs ⟩ min

⟨Min **<u>bs</u>**, cs : x ▹ xs = **<u>bs</u>** ++ cs ∧ **<u>bs ≠ [ ]</u>** : sum.bs ⟩

≡ { cambio de variable, bs ← b▹bs }

⟨Min bs, cs : x▹xs = bs ++ cs ∧ bs = [] : sum.bs ⟩ min

⟨Min bs, cs : x▹xs = b ▹ bs ++ cs ∧ **<u>b ▹ bs ≠ []</u>** : sum.(b▹bs) ⟩

≡ { lógica y neutro del ∧ }

⟨Min bs, cs : x▹xs = bs ++ cs ∧ bs = [] : sum.bs ⟩ min

⟨Min bs, cs : x ▹ xs = **b ▹ bs ++ cs** : sum.(b▹bs) ⟩

≡ { propiedades de listas }

⟨Min bs, cs : x▹xs = bs ++ cs ∧ bs = [] : sum.bs ⟩ min

⟨Min bs, cs : **x ▹ xs = b ▹ (bs ++ cs)** : sum.(b▹bs) ⟩

≡ { propiedades de listas }

⟨Min bs, cs : x▹xs = bs ++ cs ∧ bs = [] : sum.bs ⟩ min

⟨Min bs, cs : **x = b** ∧ xs = bs ++ cs : sum.(b▹bs) ⟩

≡ { eliminación de variable }

⟨Min bs, cs : x▹xs = bs ++ cs ∧ bs = [ ] : sum.bs ⟩ min

⟨Min bs, cs : xs = bs ++ cs : **sum.(x▹bs)** ⟩

≡ { definición de sum }

⟨Min bs, cs : x▹xs = bs ++ cs ∧ bs = [ ] : sum.bs ⟩ min

⟨Min bs, cs : xs = bs ++ cs : **x + sum.bs** ⟩

≡ { distributividad }

⟨Min bs, cs : x▹xs = bs ++ cs ∧ bs = [ ] : sum.bs ⟩ min

(**⟨ Min bs, cs : xs = bs ++ cs : sum.bs ⟩** + x)

≡ { H.I. }

⟨Min **bs**, cs : x ▹ xs = **bs** ++ cs ∧ **bs = [ ]** : sum.bs ⟩ min (msumaMin.xs + x)

≡ { eliminación de variable bs }

⟨Min cs : x ▹ xs = **[ ] ++ cs** : sum.[] ⟩ min (msumaMin.xs + x)

≡ { propiedades de listas }

⟨Min cs : x ▹ xs = cs : **sum.[ ]** ⟩ min (msumaMin.xs + x)

≡ { término constante }

sum.[ ] (min msumaMin.xs + x)

≡ { definición de sum }

0 min (msumaMin.xs + x)

Caso base de sumaMin

sumaMin.[ ]

= { especificación }

⟨Min as, bs, cs : **[ ] = as ++ bs ++ cs** : sum.bs ⟩

= { propiedades de listas }

⟨Min **as**, bs, **cs** : **as = [ ]** ∧ bs = [ ] ∧ **cs = [ ]** : **sum.bs** ⟩

= { eliminación de variables as y cs, rango unitario }

sum.[ ]

= { definición de sum }

0

El programa resulta

sumaMin : [Num] → Num

sumaMin.[] ≐ 0

sumaMin.(x▹xs) ≐ msumaMin.(x▹xs) min sumaMin.xs

msumaMin : [Num] → Num

msumaMin.[] ≐ 0

msumaMin.(x▹xs) ≐ 0 min (msumaMin.xs + x)

b)

maxLongEq.e.xs = ⟨Max as, bs, cs : xs = as ++ bs ++ cs ∧ iga.e.bs : #bs ⟩

Inducción en xs

Caso recursivo

Hipótesis Inductiva: maxLongEq.e.xs = ⟨Max as, bs, cs : xs = as ++ bs ++ cs ∧ iga.e.bs : #bs ⟩

<u>**maxLongEq.e.(x▹xs)**</u>

≡ { especificación }

⟨ Max as, bs, cs : x▹xs = <u>**as ++ bs ++ cs**</u> ∧ iga.e.bs : #bs ⟩

≡ { lógica }

⟨ Max as, bs, cs : x▹xs = as ++ bs ++ cs ∧ iga.e.bs ∧ (as = [] ∨ as ≠ []) : #bs ⟩

≡ { lógica }

⟨ Max as, bs, cs : <u>**(x ▹ xs = as ++ bs ++ cs ∧ iga.e.bs ∧ as = [ ]) ∨ (x▹xs = as ++ bs ++ cs ∧ iga.e.bs ∧ as ≠ [ ])**</u> : #bs ⟩

≡ { partición de rango }

⟨ Max as, bs, cs : x▹xs = as ++ bs ++ cs ∧ iga.e.bs ∧ as = [ ] : #bs ⟩ max

⟨ Max as, bs, cs : x▹xs = as ++ bs ++ cs ∧ iga.e.bs ∧ <u>**as ≠ [ ]**</u> : #bs ⟩

≡ { cambio de variable, **as** ← **a**▹**as** }

⟨ Max as, bs, cs : x▹xs = as ++ bs ++ cs ∧ iga.e.bs ∧ as = [] : #bs ⟩ max

⟨ Max <u>**a, as**</u>, bs, cs : x ▹ xs = a ▹ as ++ bs ++ cs ∧ iga.e.bs ∧ <u>**a ▹ as ≠ [ ]**</u> : #bs ⟩

≡ { lógica }

⟨ Max as, bs, cs : x▹xs = as ++ bs ++ cs ∧ iga.e.bs ∧ as = [] : #bs ⟩ max

⟨ Max a, as, bs, cs : x▹xs = **a ▹ as ++ bs ++ cs** ∧ iga.e.bs : #bs ⟩

≡ { propiedades de listas }

⟨ Max as, bs, cs : x▹xs = as ++ bs ++ cs ∧ iga.e.bs ∧ as = [] : #bs ⟩ max

⟨ Max a, as, bs, cs : **x ▹ xs = a ▹ (as ++ bs ++ cs)** ∧ iga.e.bs : #bs ⟩

≡ { propiedades de listas }

⟨ Max as, bs, cs : x▹xs = as ++ bs ++ cs ∧ iga.e.bs ∧ as = [] : #bs ⟩ max

⟨ Max **a**, as, bs, cs : **x = a** ∧ xs = as ++ bs ++ cs ∧ iga.e.bs : #bs ⟩

≡ { eliminación de variable a }

⟨Max as, bs, cs : x▹xs = as ++ bs ++ cs ∧ iga.e.bs ∧ as = [] : #bs ⟩ max

⟨ Max as, bs, cs : xs = as ++ bs ++ cs ∧ iga.e.bs : #bs ⟩

≡ { H.I. }

⟨Max **as**, bs, cs : x▹xs = **as** ++ bs ++ cs ∧ iga.e.bs ∧ **as = [ ]** : #bs ⟩ max maxLongEq.e.xs

≡ { eliminación de variable (as) }

⟨Max bs, cs : x ▹ xs = **[ ] ++ bs ++ cs** ∧ iga.e.bs : #bs ⟩ max maxLongEq.e.xs

≡ { propiedades de listas }

⟨Max bs, cs : x▹xs = bs ++ cs ∧ iga.e.bs : #bs ⟩ max maxLongEq.e.xs

Modularización

Especifico

mMaxLongEq.e.xs = ⟨Max bs, cs : xs = bs ++ cs ∧ iga.e.bs : #bs ⟩

Derivación (inducción en xs)

Caso recursivo

Hipótesis Inductiva: mMaxLongEq.e.xs = ⟨Max bs, cs : xs = bs ++ cs ∧ iga.e.bs : #bs ⟩

mMaxLongEq.e.(x▹xs)

≡ { especificación }

⟨Max bs, cs : x ▹ xs = **bs ++ cs** ∧ iga.e.bs : #bs ⟩

≡ { lógica }

⟨Max bs, cs : **x ▹ xs = bs ++ cs ∧ iga.e.bs ∧ (bs = [] ∨ bs ≠ [])** : #bs ⟩

≡ { lógica }

⟨Max bs, cs : **(x▹xs = bs ++ cs ∧ iga.e.bs ∧ bs = []) ∨ (x▹xs = bs ++ cs ∧ iga.e.bs ∧ bs ≠ [])** : #bs ⟩

≡ { partición de rango }

⟨Max **bs**, cs : x ▹ xs = **bs** ++ cs ∧ iga.e.bs ∧ **bs = [ ]** : #bs ⟩ max

⟨Max bs, cs : x ▹ xs = bs ++ cs ∧ iga.e.bs ∧ bs ≠ [] : #bs ⟩

≡ { eliminación de variable bs }

⟨Max cs : x ▹ xs = **[ ] ++ cs** ∧ iga.e.[ ] : #[ ] ⟩ max

⟨Max bs, cs : x ▹ xs = bs ++ cs ∧ iga.e.bs ∧ bs ≠ [ ] : #bs ⟩

≡ { propiedades de listas }

⟨Max cs : x ▹ xs = cs ∧ **iga.e.[ ]** : #[ ] ⟩ max

⟨Max bs, cs : x ▹ xs = bs ++ cs ∧ iga.e.bs ∧ bs ≠ [ ] : #bs ⟩

≡ { definición de iga, lógica }

⟨Max cs : x ▹ xs = cs : **#[ ]** ⟩ max

⟨Max bs, cs : x▹xs = bs ++ cs ∧ iga.e.bs ∧ bs ≠ [ ] : #bs ⟩

≡ { Rango Constante}

#[ ] max ⟨Max bs, cs : x▹xs = bs ++ cs ∧ iga.e.bs ∧ bs ≠ [] : #bs ⟩

≡ { definición de # }

0 max ⟨Max **bs**, cs : x ▹ xs = **bs** ++ cs ∧ iga.e.**bs** ∧ **bs ≠ []** : #bs ⟩

≡ { cambio de variable, **bs** ← **b**▹**bs** }

0 max ⟨Max **b, bs**, cs : x ▹ xs = b ▹ bs ++ cs ∧ iga.e.(b ▹ bs) ∧ **b**▹**bs ≠ [ ]** : #(b ▹ bs) ⟩

≡ { lógica }

0 max ⟨Max b, bs, cs : x▹xs = **b ▹ bs ++ cs** ∧ iga.e.(b ▹ bs) : #(b▹bs) ⟩

≡ { propiedades de listas }

0 max ⟨Max b, bs, cs : **x ▹ xs = b ▹ (bs ++ cs) ∧ iga.e.(b▹bs)** : #(b▹bs) ⟩

≡ { propiedades de listas }

0 max ⟨Max **b**, bs, cs : **x = b** ∧ xs = bs ++ cs ∧ iga.e.(b ▹ bs) : #(b ▹ bs) ⟩

≡ { eliminación de variable b }

0 max ⟨Max bs, cs : xs = bs ++ cs ∧ **iga.e.(x▹bs)** : #(x▹bs) ⟩

≡ { definición de iga }

0 max **⟨Max bs, cs : xs = bs ++ cs ∧ x = e ∧ iga.e.bs : #(x▹bs) ⟩**

Análisis por casos

x = e

0 max ⟨Max bs, cs : xs = bs ++ cs ∧ **x = e** ∧ iga.e.bs : #(x▹bs) ⟩

≡ { x = e ≡ True, lógica }

0 max ⟨Max bs, cs : xs = bs ++ cs ∧ iga.e.bs : **#(x ▹ bs)** ⟩

≡ { definición de # }

0 max ⟨Max bs, cs : xs = bs ++ cs ∧ iga.e.bs : **#bs+1** ⟩

≡ { distributividad }

0 max (**⟨Max bs, cs : xs = bs ++ cs ∧ iga.e.bs : #bs ⟩** + 1)

≡ { H.I. }

0 max (mMaxLongEq.e.xs + 1)

x ≠ e

0 max ⟨Max bs, cs : xs = bs ++ cs ∧ **x = e** ∧ iga.e.bs : #(x▹bs) ⟩

≡ { x = e ≡ False, lógica }

0 max ⟨Max bs, cs : **False** : #(x ▹ bs) ⟩

≡ { rango vacío, el neutro para Max es 0 cuando se considera **#** }

0 max 0

≡ { resuelvo }

0

Caso base para **mMaxLongEq**

mMaxLongEq.e.[]

= { especificación }

⟨Max bs, cs : **[ ] = bs ++ cs** ∧ iga.e.bs : #bs ⟩

= { propiedades de listas }

⟨Max bs, **cs** : bs = [ ] ∧ **cs = [ ]** ∧ iga.e.bs : #bs ⟩

= { eliminación de variable cs }

⟨Max bs : **bs = [ ] ∧ iga.e.bs** : #bs ⟩

= { Leibniz 2 }

⟨Max bs : bs = [ ] ∧ **iga.e.[ ]** : #bs ⟩

= { definición de iga, lógica }

⟨Max bs : **bs = [ ]** : #bs ⟩

= { rango unitario }

#[ ]

= { definición de # }

0

Caso base de maxLongEq

maxLongEq.e.[]

= { especificación }

⟨Max as, bs, cs : xs = as ++ bs ++ cs ∧ iga.e.bs : #bs ⟩

= { propiedades de listas }

$\langle$Max as, bs, cs : as = [] $\wedge$ bs = [] $\wedge$ cs = [] $\wedge$ iga.e.bs : #bs $\rangle$

= { eliminación de variable cs, as }

$\langle$Max bs : **bs = [ ] $\wedge$ iga.e.bs** : #bs $\rangle$

= { Leibniz 2 }

$\langle$Max bs : **bs = [ ] $\wedge$ iga.e.[ ]** : #bs $\rangle$

= { definición de iga, lógica }

$\langle$Max bs : **bs = [ ]** : #bs $\rangle$

= { rango unitario }

#[ ]

= { definición de # }

0

El programa resulta

maxLongEq.e.[] $\doteq$ 0

maxLongEq.e.(x▹xs) $\doteq$ mMaxLongEq.e.(x▹xs) max maxLongEq.e.xs

mMaxLongEq.e.[] $\doteq$ 0

mMaxLongEq.e.(x▹xs) $\doteq$ (x = e $\rightarrow$ 0 max (mMaxLongEq.e.xs + 1)

$\square$ x $\neq$ e $\rightarrow$ 0

)