Apunte de Introducción a los Algoritmos

Pedro Sánchez Terraf*

8 de abril de 2017

1. Guía de clases

1.1. Definiciones de funciones

La construcción básica en un programa funcional es la definición de funciones. Para definir una función, necesito decir qué "come" y qué "devuelve". Lo que "come" o consume una función se denominan **argumentos** y lo que "devuelve" o su resultado se denomina su **valor** para dichos argumentos. Como un ejemplo muy zonzo, consideremos la función *suma* que suma dos enteros.

$$\begin{aligned} &suma : Int \to Int \to Int \\ &suma.x.y \doteq x + y \end{aligned} \tag{1}$$

Notemos que una definición de función como la (1) introduce dos cosas:

1. Un nuevo símbolo que nombra la función (en este caso, f) y qué tipo tiene.
2. Una nueva igualdad, que es verdadera para todos los valores de las variables (i.e., la igualdad $suma.x.y = x + y$).

Decimos que la igualdad $suma.x.y = x + y$ introducida por la definición (1) es **válida**: es verdadera para todos los valores de sus variables. En tal sentido, funciona como un axioma o un teorema (por ejemplo, como la conmutatividad de la suma, $a + b = b + a$).

Como las definiciones introducen igualdades (y la igualdad es **simétrica**, i.e. $a = b$ implica $b = a$), podemos pasar de la expresión $suma.x.y$ a $x + y$ y viceversa. Sin embargo le pondremos dos nombres distintos a cada una de los procesos. Por ejemplo, si calculamos $suma.9.16$,

$$\begin{aligned} & suma.9.16 \\ = \{ \;& \text{Definición de } suma \; \} \\ & 9 + 16 \\ = \{ \;& \text{Aritmética} \; \} \\ & 25, \end{aligned}$$

estamos usando la igualdad introducida por (1) de izquierda a derecha. En tal caso, decimos que estamos **desplegando** la definición. Por otro lado, si la usamos de derecha a izquierda, "haciendo aparecer" la función *suma* como a continuación:

$$\begin{aligned} & 256 \\ = \{ \;& \text{Aritmética} \; \} \\ & 128 + \underline{64 + 64}, \\ = \{ \;& \text{Definición de } suma \; \} \\ & 128 + suma.64.64 \end{aligned}$$

decimos que estamos **plegando** la definición.

*CIEM-FAMAF.

1.2. Patrones

Una herramienta muy poderosa en programación funcional es la utilización de *patrones*.

Un **patrón** es la forma que tiene el argumento de una función. La utilidad de los patrones radica en que podemos estipular qué forma tienen los argumentos. Por ejemplo, para la función

$$\begin{aligned} &f : (Int, Int) \to Int \\ &f.(x, y) \doteq x * y \end{aligned} \tag{2}$$

que toma un par de enteros y devuelve la suma de sus componentes, el patrón es $\boldsymbol{(x, y)}$.

Comentario. La función f **no es la misma** que la función *multiplicar* de la Guía 1. Aparentemente hace lo mismo, pero su tipo es distinto. *multiplicar* tiene dos argumentos, y f tiene uno que es un par (tupla de dos componentes).

Cada vez que le demos algo de comer a f, deberemos "emparejarlo" (*match*, en inglés) con el patrón antes de aplicar la función. Por ejemplo, si queremos calcular $f.(4 * 2, 3)$, debemos entender cómo se corresponde el argumento $(4 * 2, 3)$ con el patrón $\boldsymbol{(x, y)}$:

$$\begin{array}{ccccc} (& 4*2 & , & 3 &) \\ & \downarrow & & \downarrow & \\ (& \boldsymbol{x} & , & \boldsymbol{y} &). \end{array}$$

Entonces, a la variable x le corresponde el valor $4 * 2$ y a y le corresponde 3. Entonces podemos desplegar la definición de f para obtener el resultado:

$$\begin{array}{ll} & \underline{f.(4 * 2, 3)} \\ = \{ & \text{Definición de } f \text{ (de acuerdo al patrón, } x = 4 * 2 \text{ e } y = 3) \ \} \\ & 4 * 2 * 3 \\ = \{ & \text{Aritmética } \} \\ & 24. \end{array}$$

Veamos un ejemplo ligeramente distinto. Queremos calcular $f.(4 + 1, 3)$. Si lo hacemos de la siguiente manera, estará **mal**:

$$\begin{array}{ll} & \underline{f.(4 + 1, 3)} \\ = \{ & \text{Definición de } f \ \} \\ & 4 + 1 * 3 \\ = \{ & \text{Aritmética } \} \\ & 7. \end{array}$$

El resultado debería haber sido 15. Lo que sucedió aquí es que siempre que emparejamos dos expresiones, conviene poner paréntesis para no introducir efectos secundarios indeseados. En este caso, la expresión que se empareja con x es $4 + 1$, pero para estar seguros que se considerará como un paquete cerrado, debemos ponerlo entre paréntesis:

$$\begin{array}{ll} & \underline{f.(4 + 1, 3)} \\ = \{ & \text{Definición de } f \ \} \\ & (4 + 1) * 3 \\ = \{ & \text{Aritmética } \} \\ & 15. \end{array}$$

Dos patrones que utilizaremos mucho son los que describen números naturales (que denominamos *Nat* y contienen al 0 como $\mathbb{N}_0$) y listas. Para los primeros, los patrones son 0 y n. Lo veamos con un ejemplo de función más.

$$g : Nat \to Int$$
$$g.0 \doteq 1 \tag{3}$$
$$g.(n+1) \doteq n+2. \tag{4}$$

De acuerdo a su definición, g sólo puede consumir 0 o algo que se corresponda con el patrón $(n+1)$.

Ejercicio 1.1. Convencerse que un número natural es o bien 0 o se puede emparejar con el patrón $n+1$.

Supongamos que queremos calcular $g.6$. No podemos aplicar la definición de g a esta expresión, porque no le estamos dando de comer ni 0 ni una expresión de la forma $(n+1)$. Pero hacer un poco de aritmética antes y luego podremos aplicar el caso (4) de la definición de g:

$$\begin{array}{ll} & g.\underline{6} \\ = \{ & \text{Aritmética } \} \\ & g.(5+1) \\ = \{ & \text{Definición de } g \ \} \\ & 5+2 \\ = \{ & \text{Aritmética } \} \\ & 7. \end{array}$$

En el medio de la prueba, al emparejar la expresión $(5+1)$ con el patrón $(n+1)$ deducimos que $n = 5$. Luego, al desplegar la definición de g pasamos al término derecho de la ecuación (4) obteniendo $5+2$.

Ejercicio 1.2. Deducir qué hace la función g.

En la Sección 1.4 analizaremos el caso de las listas, para las cuales también hay dos patrones paradigmáticos.

Técnica de definición de funciones: composición (*modularización*). Descomponer un problema en partes.

Trabajo en clase

1. Definir la función *esBisiesto* : $Num \to Bool$, que indica si un año es bisiesto. Un año es bisiesto si es divisible por 400 o es divisible por 4 pero no es divisible por 100. (usar mód).

2. Definir la función *max3* : $Num \to Num \to Num \to Num$, que dados tres números devuelve el mayor de los tres (usar máx).

Tarea

Entregar por escrito:

1. *mayor3* : $(Int, Int, Int) \to (Bool, Bool, Bool)$, que dada una una terna de enteros devuelve una terna de valores booleanos que indica si cada uno de los enteros es mayor que 3.

   Por ejemplo: $mayor3.(1,4,3) = (False, True, False)$ y $mayor3.(5,1984,6) = (True, True, True)$.

2. Definir la función *dispersion* : $Num \to Num \to Num \to Num$, que toma los tres valores y devuelve la diferencia entre el más alto y el más bajo. (*Ayuda*: usar *max3* y *min3*. De esa forma se puede definir *dispersion* sin hace análisis por casos.)

1.3. Patrones y Recursión

El siguiente paso es utilizar múltiples patrones para definir una función de manera *recursiva*. Esto quiere decir que, a diferencia de los ejemplos anteriores, el nuevo símbolo de función introducido va a aparecer de ambos lados del signo $\doteq$.

Damos como ejemplo una "misteriosa" función h.

$$h : Int \to Int$$
$$h.0 \doteq 0 \tag{5}$$
$$h.(n+1) \doteq 1 + 2 * n + h.n \tag{6}$$

Observemos cómo h aparece de ambos lados de la Ecuación (6). Sin embargo, nos daremos cuenta con ejemplos que esto no conlleva ningún problema, siempre que $h.x$ con x un entero mayor o igual a 0.

Como dijimos en la Sección 1.1, toda (línea de) definición introduce una ecuación válida nueva. Ahora podemos calcular los valores para h.

Teorema 1.3 ("0")**.** $h.0 = 0$.

Este "Teorema 0" es trivial, porque es exactamente la Ecuación (5). Nosotros escribimos su prueba de la siguiente manera.

$$\begin{array}{ll} & h.0 \\ = \{ & \text{Definición de } h \ \} \\ & 0 \end{array}$$

Esto no sólo es el "cálculo" de $h.0$, sino que también es a la vez la *demostración* de que $h.0 = 0$. Se refuerza entonces lo que dijimos al principio: un objetivo de esta materia es "ampliar nuestra capacidad de cálculo", especialmente ampliando los tipos de objetos sobre los cuales podemos "calcular". Por ejemplo, acabamos de calcular que el valor de la expresión $h.0 = 0$ de tipo *Bool* es *True*. Demostrar teoremas es esencialmente calcular con booleanos.

Prosigamos descubriendo qué hace h, calculando $h.1$:

$$\begin{array}{ll} & h.\underline{1} \\ = \{ & \text{Aritmética} \ \} \\ & h.(0+1) \\ = \{ & \text{Definición de } h \ \} \\ & 1 + 2 * 0 + \underline{h.0} \\ = \{ & \text{Teorema "0"} \ \} \\ & 1 + 2 * 0 + 0 \\ = \{ & \text{Aritmética} \ \} \\ & 1. \end{array}$$

Hemos demostrado el siguiente

Teorema 1.4 ("1")**.** $h.1 = 1$.

¿Cómo es el Teorema "n"?

Ejercicio 1.5. Calcular $h.2$, $h.3$ y $h.4$. ¿Cuánto vale, en general $h.n$?

1.4. Listas

Uno de los tipos más importantes en Haskell son las *listas*. Los tipos de listas se indican poniendo entre corchetes ("[" y "]") otro tipo, que será el tipo de los elementos de la lista. Así, $[Int]$ es el tipo de todas las listas de enteros. Las propiedades básicas de las listas son dos.

1. Toda lista se construye a partir de la lista vacía $[\,]$, agregando elementos por la izquierda:
$$[2,3] = 2 \triangleright \underline{[3]} = 2 \triangleright (3 \triangleright [\,]). \tag{7}$$
2. Los elementos de una lista deben ser del **mismo tipo**.

El triángulo $\triangleright$ se lee "seguido de" y junto con $[\,]$ son los **constructores de listas**, porque con ellos se arma cualquier lista. Para simplificar podemos pensar que el tipo del triángulo es

$$(\triangleright) : A \to [A] \to [A],$$

que significa que toma un "elemento" (de algún tipo A) y una lista (de tipo $[A]$), y le agrega el elemento, obteniendo una lista más larga de tipo $[A]$.

Ejemplo 1.6.
1. $[x, y+z]$ (de tipo $[Num]$, puesto que el resultado de una suma es de tipo Num);
2. $[True, p]$ (de tipo $[Bool]$);
3. [`"hola"`, `"chau"`] (de tipo $[String]$).

Observando las Ecuaciones (7), podemos darnos cuenta que toda lista es o bien la lista vacía $[\,]$, o resulta de agregar un elemento a otra lista. (De hecho, el elemento agregado es el primero de la lista original).

Un ejemplo de función que toma listas es *head*:

$$\begin{aligned} &head : [A] \to [A] \\ &head.(x \triangleright xs) \doteq x \end{aligned} \tag{8}$$

Ejercicio 1.7. Calcular $head.[1,2]$, $head.$[`"hola"`, `"chau"`] y $head.[[\,],[1,2]]$.

Hagamos parte del primer ejemplo como ayuda. Como está escrito, el argumento $[1,2]$ no es digerible por la función *head*. Para que lo pueda consumir, hay que ponerlo de acuerdo al patrón $(x \triangleright xs)$.

$$\begin{aligned} & head.\underline{[1,2]} \\ =\ \{ & \text{ Definición de lista } \} \\ & head.(1 \triangleright [2]) \\ =\ \{ & \text{ Definición de } head \ \} \\ & 1. \end{aligned}$$

Para poder entender el último paso, hay que tomar conciencia quién es el "x" y quién el "xs" en la expresión $(1 \triangleright [2])$. Una vez que sabemos eso, podemos emparejar con el patrón $(x \triangleright xs)$ (y el resultado de *head* será el "x").

Prosiguiendo ahora con recursión, usaremos ambos patrones en una definición.

Ejercicio 1.8. Sea la siguiente función

$$\begin{aligned} &uno : [A] \to [Int] \\ &uno.[\,] \doteq [\,] && (9) \\ &uno.(x \triangleright xs) \doteq 1 \triangleright uno.xs. && (10) \end{aligned}$$

Calcular $uno.[3,6,7]$.

En la definición de *uno*, la línea (9) se llama **caso baso**, y la segunda (donde aparece *uno* de ambos lados), **caso inductivo**.

Hacemos los primeros pasos como ayuda:

$$
\begin{array}{ll}
 & \underline{uno.[3,6,7]} \\
= \{ & \text{Definición de lista } \} \\
 & uno.(3 \triangleright [6,7]) \\
= \{ & \text{Definición de } uno \ \} \\
 & 1 \triangleright uno.[6,7],
\end{array}
$$

etcétera. La función *uno* toma cada elemento de la lista y lo transforma en algo distinto. Las funciones de lista que trabajan así se llaman **MAP** (o de "aplicación").

Una función de listas que se comporta distinto es la que suma los elementos de una lista de números. Se llama *sum*. Por suerte, todos sabemos cómo se suma una lista de números: $sum.[2,1,3] = 2+1+3 = 6$ (si no lo sabemos a esto, estamos en problemas). Las funciones como *sum* que repiten una operación con todos los elementos de la lista, se llaman **FOLD**. El objetivo ahora es *encontrar* una definición recursiva para *sum*, como la que tenemos de *uno*:

$$
\begin{aligned}
& sum : [Int] \to Int \\
& sum.[\,] \doteq ??? && (11) \\
& sum.(x \triangleright xs) \doteq ???. && (12)
\end{aligned}
$$

Supongamos que queremos calcular como en el principio del Ejercicio 1.8. Tendríamos algo como lo siguiente

$$
\begin{array}{ll}
 & \underline{sum.[2,1,3]} \\
= \{ & \text{Definición de lista } \} \\
 & sum.(2 \triangleright [1,3]) \\
= \{ & \text{Definición de } sum \ \} \\
 & (\text{algo con 2 y } sum.[1,3])
\end{array}
$$

Pero nosotros ya sabemos cómo queremos que se comporte *sum*: $sum.[2,1,3] = 6$ y $sum.[1,3] = 4$. Entonces tenemos lo siguiente:

$$
\begin{array}{ll}
 & 6 \\
= \{ & \text{valor de } sum.[2,1,3] \ \} \\
 & \underline{sum.[2,1,3]} \\
= \{ & \text{Definición de lista } \} \\
 & sum.(2 \triangleright [1,3]) \\
= \{ & \text{Definición de } sum \ \} \\
 & (\text{algo con 2 y } sum.[1,3])
\end{array}
$$

¿Cómo obtenemos 6 a partir de 2 y la suma del resto de los elementos, $sum.[1,3] = 4$? Simple, sumando. Luego, en este ejemplo particular, tenemos

$$
\begin{array}{ll}
 & 6 \\
= \{ & \text{valor de } sum.[2,1,3] \ \} \\
 & \underline{sum.[2,1,3]} \\
= \{ & \text{Definición de lista } \} \\
 & sum.(2 \triangleright [1,3]) \\
= \{ & \text{Definición de } sum \ (*) \ \} \\
 & 2 + sum.[1,3] \\
= \{ & \text{valor de } sum.[1,3] \ \} \\
 & 2+4
\end{array}
$$

El paso clave, indicado con (*),

$$sum.(2 \triangleright [1,3]) = 2 + sum.[1,3]$$

nos permite deducir qué hace *sum* cuando le damos de comer el patrón $(x \triangleright xs)$:

$$sum.(x \triangleright xs) \doteq x + sum.xs.$$

Ésta es la parte inductiva de la definición de *sum*. Para establecer el caso base, sigamos aplicando esta definición en nuestro ejemplo, calculando $sum.[1,3]$:

$$\begin{array}{ll} & sum.\underline{[1,3]} \\ = \{ & \text{Definición de lista } \} \\ & sum.(1 \triangleright [3]) \\ = \{ & \text{Definición de } sum \ \} \\ & 1 + sum.\underline{[3]} \\ = \{ & \text{Definición de lista } \} \\ & 1 + \underline{sum.(3 \triangleright [\,])} \\ = \{ & \text{Definición de } sum \ (*) \ \} \\ & 1 + 4 + sum.[\,] \end{array}$$

Ejercicio 1.9. Deducir a partir de este ejemplo cuánto debe valer $sum.[\,]$.

Ejercicio 1.10. Definir por recursión la función $duplica : [Int] \to [Int]$, que multiplica por 2 todos los elementos de una lista de enteros.

La última clase de funciones simples de listas son las **FILTER**. Estas funciones seleccionan los elementos de la lista que cumplen con algún criterio. En nuestro caso, un criterio será un **predicado**, es decir, una función que devuelve un *Bool*.

Ejercicio 1.11. Definir la función $esMultiplo2 : Int \to Bool$ que dado un entero devuelve *True* si y sólo si es par. (Ayuda: usar mód).

Ahora podemos dar un ejemplo de función FILTER (o "filtro"):

$$soloPares : [Int] \to [Int] \tag{13}$$

$$soloPares.[\,] \doteq [\,] \tag{14}$$

$$\begin{array}{lll} soloPares.(x \triangleright xs) \doteq & (& esMultiplo2.x \quad \to x \triangleright soloPares.xs \\ & \square & \neg esMultiplo2.x \quad \to soloPares.xs \\ &) & \end{array}$$

Ejercicio 1.12. Calcular $soloPares.[1,2,3]$.

Podemos resumir las características de las funciones de clases FILTER, MAP y FOLD para poder reconocerlas:

1. Una función FILTER toma una lista y devuelve otra *del mismo tipo*, y tiene la forma general:

$$\begin{array}{lll} filtro : [A] \to [A] & & \\ filtro.[\,] \doteq [\,] & & \\ filtro.(x \triangleright xs) \doteq & (& condicion.x \quad \to x \triangleright filtro.xs \\ & \square & \neg condicion.x \quad \to filtro.xs \\ &) & \end{array}$$

   De hecho, el resultado de una FILTER es una lista cuyos elementos estaban en la lista original, y son exactamente los que satisfacen el predicado $condicion : A \to Bool$.

2. Una función MAP toma una lista y devuelve otra *de la misma longitud*. Tiene la forma general

$$\begin{aligned} &aplicar : [A] \to [B] \\ &aplicar.[\,] \doteq [\,] \\ &aplicar.(x \triangleright xs) \doteq funcion.x \triangleright (aplicar.xs) \end{aligned}$$

   A cada elemento de la lista original le aplicamos la $funcion : A \to B$.

3. Una función FOLD toma una lista (el tipo del resultado puede no ser una lista). Su forma general es

$$\begin{aligned} &operar : [A] \to B \\ &operar.[\,] \doteq b \\ &operar.(x \triangleright xs) \doteq operacion.x.(operar.xs), \end{aligned}$$

   donde todos los elementos de la lista se los combina usando la $operacion : A \to B \to B$. En el caso de *sum*, esta operación es $(+) : Num \to Num \to Num$ (es decir, $A = B = Num$).

La función cardinal o **longitud** # no es ni MAP, ni FILTER, ni FOLD.[1]

Ejercicio 1.13. Escribir a # como la composición de una función MAP con una FOLD. (Ayuda: ya las vimos a esas dos funciones).

A continuación, se incluyen todas las definiciones de los operadores de listas. Estas funciones se pueden usar libremente (en un examen por ejemplo) para definir otras funciones nuevas.

longitud

$$\#[\,] \doteq 0$$
$$\#(x \triangleright xs) \doteq 1 + \#xs$$

head (cabeza)

$$head.(x \triangleright xs) \doteq x$$

tail (cola)

$$tail.(x \triangleright xs) \doteq xs$$

concatenar

$$[\,] +\!\!+ ys \doteq ys$$
$$(x \triangleright xs) +\!\!+ ys \doteq x \triangleright (xs +\!\!+ ys)$$

pegar por la derecha

$$[\,] \triangleleft y \doteq y \triangleright [\,]$$
$$(x \triangleright xs) \triangleleft y \doteq x \triangleright (xs \triangleleft y)$$

índice

$$(x \triangleright xs)\,!\,0 \doteq x$$
$$(x \triangleright xs)\,!\,(n+1) \doteq xs\,!\,n$$

tomar

$$xs \uparrow 0 \doteq [\,]$$
$$[\,] \uparrow n \doteq [\,]$$
$$(x \triangleright xs) \uparrow (n+1) \doteq x \triangleright (xs \uparrow n)$$

[1] En realidad esto es un poco mentira, sí es una función FOLD, pero es un poco más complicado verla así.

tirar

$$(\downarrow) : [A] \to Int \to [A]$$

$$\begin{aligned} xs \downarrow 0 &\doteq xs \\ [\,] \downarrow n &\doteq [\,] \\ (x \triangleright xs) \downarrow (n+1) &\doteq xs \downarrow n \end{aligned}$$

suma

$$sum : [Num] \to Num$$

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

Ejercicio 1.14. Calcular $[3,4] +\!\!+ [5]$ y $[3,4,5] \uparrow 2$ usando las definiciones.

Las funciones que siguen las llamo "funciones tabú", porque (en esta materia) se debe escribir su definición para poder usarlas.

map

$$map : (A \to B) \to [A] \to [B]$$

$$\begin{aligned} map.f.[\,] &\doteq [\,] \\ map.f.(x \triangleright xs) &\doteq f.x \triangleright map.f.xs \end{aligned}$$

Ejemplo: $duplica = map.((*).2)$

filter

$$filter : (A \to Bool) \to [A] \to [A]$$

$$\begin{aligned} filter.p.[\,] &\doteq [\,] \\ filter.p.(x \triangleright xs) &\doteq (\ p.x \longrightarrow x \triangleright filter.p.xs \\ &\quad \Box\ \neg p.x \longrightarrow filter.p.xs \\ &\quad) \end{aligned}$$

Ejemplo: $soloPares = filter.esMultiplo2$

foldl

$$foldl : (A \to B \to A) \to A \to [B] \to A$$

$$\begin{aligned} foldl.f.z.[\,] &\doteq z \\ foldl.f.z.(x \triangleright xs) &\doteq foldl.f.(f.z.x).xs \end{aligned}$$

Ejemplos: $sum = foldl.(+).0$
$rev = foldl.(\triangleleft).[\,]$

1.5. Trabajo en clase

Definir las siguientes funciones y evaluarlas manualmente sobre los ejemplos dados:

1. $incPrim : [(Int, Int)] \to [(Int, Int)]$, que dada una lista de pares de enteros, le suma 1 al primer número de cada par.
   Ejemplos: $incPrim.[(20,5),(50,9)] = [(21,5),(51,9)]$, $incPrim.[(4,11),(3,0)] = [(5,11),(4,0)]$.

2. $expandir : String \to String$, pone espacios entre cada letra de una palabra.
   Ejemplo: $expandir$.`"hola"` = `"h o l a"` (¡sin espacio al final!).

1.6. Inducción

Retomemos la función h del comienzo de la Sección 1.3. A esta altura ya sabemos que el Teorema "n" es $h.n = n^2$. Para demostrar propiedades de funciones definidas recursivamente, utilizamos pruebas por **inducción**.

Comparemos variantes de las pruebas del Teorema "1" y el Teorema "2", ahora sabiendo que hay un cuadrado dando vueltas por ahí.

$$
\begin{array}{ll}
 & h.\underline{1} \\
= \{ & \text{Aritmética } \} \\
 & h.(0+1) \\
= \{ & \text{Definición de } h \ \} \\
 & 1+2*0+\underline{h.0} \\
= \{ & \text{Teorema "0" } \} \\
 & 1+2*0+0^2 \\
= \{ & \text{Aritmética (binomio cuadrado) } \} \\
 & (1+0)^2 \\
= \{ & \text{Aritmética } \} \\
 & 1^2.
\end{array}
\qquad
\begin{array}{ll}
 & h.\underline{2} \\
= \{ & \text{Aritmética } \} \\
 & h.(1+1) \\
= \{ & \text{Definición de } h \ \} \\
 & 1+2*1+\underline{h.1} \\
= \{ & \text{Teorema "1" } \} \\
 & 1+2*1+1^2 \\
= \{ & \text{Aritmética (binomio cuadrado) } \} \\
 & (1+1)^2 \\
= \{ & \text{Aritmética } \} \\
 & 2^2.
\end{array}
$$

Son casi iguales. Observemos, de todos modos, que la prueba del Teorema "2" requiere haber probado el Teorema "1" previamente. Por esto, demostrar el Teorema "487" (que enuncia que $h.487 = 487^2$) es posible, pero antes tendríamos que escribir las 487 pruebas de los Teoremas "n" para $n = 0, \ldots, 486$. Muy aburrido, considerando que las últimas 486 son lo mismo.

El salto de abstracción está encontrar una "forma general" de dicha demostración. Hay un numerito que va cambiando, y lo podemos llamar n. Entonces, la versión general de esta prueba queda así:

$$
\begin{array}{ll}
 & h.(n+1) \\
= \{ & \text{Definición de } h \ \} \\
 & 1+2*n+\underline{h.n} \\
= \{ & \text{Teorema "}n\text{" } \} \\
 & 1+2*n+n^2 \\
= \{ & \text{Aritmética (binomio cuadrado) } \} \\
 & (1+n)^2 \\
= \{ & \text{Aritmética } \} \\
 & (n+1)^2.
\end{array}
$$

Habiéndonos dado cuenta que existe esta forma general (para el Teorema "$n+1$"), es trivial dar una *receta* para escribir la prueba del Teorema "487":

- Escribir la prueba del Teorema "0".
- Copiar la prueba general cambiando el n por todos los valores entre 0 y 486.

En lugar de hacer la (todavía inhumana) tarea del segundo ítem, podemos buscar un atajo. Esto es, suponer que ya hemos escrito la prueba del Teorema "n" y simplemente dar la del Teorema "$n+1$". En resumidas cuentas, tenemos los siguientes pasos:

Elegir variable en la cual haremos la inducción (en este ejemplo, sólo hay una, la n).

Caso Base Probar el Teorema "0".

Caso Inductivo Suponiendo que hemos probado el Teorema "n" escribimos la prueba del Teorema "$n+1$"

Esto es una prueba por **inducción en** Nat.

También se puede hacer inducción en listas. A diferencia con los naturales, donde los casos son 0 y $(n + 1)$ (o bien 1 y $(n + 1)$), los casos de listas son $[\,]$ y $(x \triangleright xs)$. Para trabajar un ejemplo, demostraremos que

$$sum.(xs +\!\!+ ys) = sum.xs + sum.ys \tag{15}$$

El esquema para hacer inducción en listas es exactamente el mismo que los naturales, pero lo describiremos en más detalle para este ejemplo. Los pasos a seguir son:

1. **Elegir una variable** para hacer la inducción.

   En el ejemplo que elegimos, los patrones relevantes aparecen siempre en la variable xs, así que elegimos esa para hacer la inducción.

2. **Caso Base**: reemplazamos la variable elegida por $[\,]$ y probamos lo que obtenemos:

   $$sum.([\,] +\!\!+ ys) = sum.[\,] + sum.ys. \tag{16}$$

3. **Caso inductivo**: Copiamos el teorema tal como venía (Ecuación (15)), y le llamamos **Hipótesis Inductiva** (HI):

   $$\text{(HI)} \qquad sum.(xs +\!\!+ ys) = sum.xs + sum.ys,$$

   reemplazamos ahora $(x \triangleright xs)$ en lugar de xs en todos lados:

   $$sum.((x \triangleright xs) +\!\!+ ys) = sum.(x \triangleright xs) + sum.ys \tag{17}$$

   y demuestramos lo que obtuvimos usando las definiciones y la HI.

Probemos el caso base.

Ejercicio 1.15. Escribir las justificaciones que faltan en la demostración que sigue, subrayando dónde se aplican los cambios.

$$\begin{array}{ll} & sum.([\,] +\!\!+ ys) \\ = \{ & \qquad\qquad \} \\ & sum.ys \\ = \{ & \qquad\qquad \} \\ & 0 + sum.ys \\ = \{ & \qquad\qquad \} \\ & sum.[\,] + sum.ys \end{array}$$

Para el caso inductivo, debemos probar la igualdad (14) usando eventualmente la HI. Para hacerlo, tenemos tres caminos básicos:

1. salir del lado izquierdo $(sum.((x \triangleright xs) +\!\!+ ys))$ y llegar al derecho $(sum.(x \triangleright xs) + sum.ys)$;
2. al revés, de derecha a izquierda; o bien
3. tomar toda la expresión booleana (17) y probar que es equivalente a $True$.

Usualmente, las primeras dos maneras resultan en pruebas más cortas, pero requieren un poco más de ingenio. Las pruebas que toman todo y llegan a $True$ suelen ser al revés.

Haremos un ejemplo en el que probamos un caso particular de la hipótesis inductiva, suponiendo que ya probamos el caso base. Es decir, supongamos que sabemos

$$\text{(HI)} \qquad sum.([\,] +\!\!+ ys) = sum.[\,] + sum.ys.$$

Usando esto, vamos a probar que $sum.([4] +\!\!+ ys) = sum.[4] + sum.ys$

$$
\begin{array}{ll}
 & sum.([4] +\!\!+ ys) \\
= \{ & \text{Definición de lista } \} \\
 & sum.((4 \triangleright [\,]) +\!\!+ ys) \\
= \{ & \text{Definición de } +\!\!+ \ \} \\
 & sum.(4 \triangleright ([\,] +\!\!+ ys)) \\
= \{ & \text{Definición de } sum \ \} \\
 & 4 + sum.([\,] +\!\!+ ys) \\
= \{ & \text{HI } \} \\
 & 4 + sum.[\,] + sum.ys \\
= \{ & \text{Definición de } sum \ \} \\
 & sum.(4 \triangleright [\,]) + sum.ys \\
= \{ & \text{Definición de lista } \} \\
 & sum.[4] + sum.ys
\end{array}
$$

Ejercicio 1.16. Completar la prueba por inducción de este teorema, siguiendo la receta de más arriba.

La solución está al final del apunte.

Ejercicio 1.17. Considerando la función $quitarCeros : [Num] \to [Num]$ definida de la siguiente manera

$$
\begin{array}{lcll}
quitarCeros.[\,] & \doteq & [\,] & \\
quitarCeros.(x \triangleright xs) & \doteq & (& x \neq 0 \to x \triangleright quitarCeros.xs \\
 & & \square & x = 0 \to quitarCeros.xs \\
 & &) &
\end{array}
$$

demostrá que

$$sum.(quitarCeros.xs) = sum.xs$$

2. Soluciones

Ejercicio 1.16

Prueba 1. **Elegimos una variable**: hacemos inducción en xs.

Caso Base: reemplazo la variable elegida por $[\,]$:

$$sum.([\,] +\!\!+ ys) = sum.[\,] + sum.ys$$

y pruebo lo que obtengo:

$$\begin{array}{ll} & \underline{sum.([\,] +\!\!+ ys)} \\ = \{ & \text{Definición de } +\!\!+ \ \} \\ & sum.ys \\ = \{ & \text{Aritmética } \} \\ & \underline{0} + sum.ys \\ = \{ & \text{Definición de } sum \ \} \\ & sum.[\,] + sum.ys \end{array}$$

Caso inductivo: Copio el teorema tal como venía, y le llamo Hipótesis Inductiva:

$$\text{(HI)} \qquad sum.(xs +\!\!+ ys) = sum.xs + sum.ys,$$

reemplazo ahora $(x \triangleright xs)$ en lugar de xs en todos lados:

$$sum.((x \triangleright xs) +\!\!+ ys) = sum.(x \triangleright xs) + sum.ys$$

y demuestro lo que obtuve usando las definiciones y la HI.

$$\begin{array}{ll} & \underline{sum.((x \triangleright xs) +\!\!+ ys)} \\ = \{ & \text{Definición de } +\!\!+ \ \} \\ & sum.(x \triangleright (xs +\!\!+ ys)) \\ = \{ & \text{Definición de } sum \ \} \\ & x + \underline{sum.(xs +\!\!+ ys)} \\ = \{ & \text{HI } \} \\ & \underline{x + sum.xs} + sum.ys \\ = \{ & \text{Definición de } sum \ \} \\ & sum.(x \triangleright xs) + sum.ys \quad \square \end{array}$$

Prueba 2. Puede ocurrir que no se nos ocurran algunos pasos de la demostración de arriba, así que hay otra forma de escribir esta prueba, un poco más repetitiva pero que hace más fácil imaginarse qué hacer. Por lo demás, la prueba es la misma (también por inducción).

Caso Base: $sum.([\,] +\!\!+ ys) = sum.[\,] + sum.ys$.

En vez de salir de un lado y llegar al otro, tomo todo.

$$\begin{array}{ll} & \underline{sum.([\,] +\!\!+ ys)} = sum.[\,] + sum.ys \\ \equiv \{ & \text{Definición de } +\!\!+ \ \} \\ & sum.ys = \underline{sum.[\,]} + sum.ys \\ \equiv \{ & \text{Definición de } sum \ \} \\ & sum.ys = 0 + sum.ys \\ \equiv \{ & \text{Aritmética } \} \\ & True \end{array}$$

Caso inductivo: Usando

$$\text{(HI)} \qquad sum.(xs +\!\!+ ys) = sum.xs + sum.ys,$$

pruebo:

$$sum.((x \triangleright xs) +\!\!+ ys) = sum.(x \triangleright xs) + sum.ys.$$

Tomo todo y opero:

$\underline{sum.((x \triangleright xs) +\!\!+ ys)} = sum.(x \triangleright xs) + sum.ys$
$\equiv$ { Definición de $+\!\!+$ }
$\underline{sum.(x \triangleright (xs +\!\!+ ys))} = sum.(x \triangleright xs) + sum.ys$
$\equiv$ { Definición de sum }
$x + \underline{sum.(xs +\!\!+ ys)} = sum.(x \triangleright xs) + sum.ys$
$\equiv$ { HI }
$x + sum.xs + sum.ys = \underline{sum.(x \triangleright xs)} + sum.ys$
$\equiv$ { Definición de sum }
$x + sum.xs + sum.ys = x + sum.xs + sum.ys$
$\equiv$ { Reflexividad de $=$ }
$True$ $\square$

Ejercicio 1.17

Demostración. Lo probamos por inducción en xs.

Caso Base: Debemos probar

$$sum.(quitarCeros.[\,]) = sum.[\,].$$

$\underline{sum.(quitarCeros.[\,])}$
$=$ { Definición de $quitarCeros$ }
$sum.[\,].$

Caso inductivo: Usando

$$\text{(HI)} \qquad sum.(quitarCeros.xs) = sum.xs$$

probamos

$$sum.(quitarCeros.(x \triangleright xs)) = sum.(x \triangleright xs).$$

Dividimos la prueba en dos casos.

Si $x = 0$:

$\underline{sum.(quitarCeros.(0 \triangleright xs))}$
$=$ { Definición de $quitarCeros$ }
$sum.(quitarCeros.xs)$
$=$ { HI }
$sum.xs$
$=$ { Aritmética }
$0 + sum.xs$
$=$ { Definición de sum }
$sum.(0 \triangleright xs).$

Si $x \neq 0$:

$$
\begin{array}{ll}
 & sum.(\underline{quitarCeros.(x \triangleright xs)}) \\
= \{ & \text{Definición de } quitarCeros \ \} \\
 & sum.(x \triangleright quitarCeros.xs) \\
= \{ & \text{Definición de } sum \ \} \\
 & \underline{x + sum(quitarCeros.xs)} \\
= \{ & \text{HI} \ \} \\
 & x + sum.xs \\
= \{ & \text{Definición de } sum \ \} \\
 & sum.(x \triangleright xs). \quad \square
\end{array}
$$