Tarea 4
Programación declarativa

Peto Gutierrez Emmanuel
Ernesto Rubén Palacios Gómez

22 de mayo de 2018

1.- Por definición $a \leq b$ implica que $\exists c \in \mathbb{N}$ tal que $a + c = b$.

Voy a etiquetar las flechas de manera única con una terna, de forma que a la flecha $a \xrightarrow{f} b$ le corresponde la terna $f = (a, b, c)$. Por ejemplo $id_a = (a, a, 0)$ o si $b = s(a)$ entonces a la flecha $a \to b$ le corresponde $(a, b, 1)$. La composición se definirá así: Si tenemos que $a \leq b$ y $b \leq d$ y tenemos que $a + c = b$ y $b + e = d$, entonces $(b, d, e) \circ (a, b, c) = (a, d, e + c)$. La primera entrada de la terna de la izquierda debe ser igual a la segunda entrada de la terna de la derecha.

Veremos que se cumple la asociatividad y la identidad por la izquierda y derecha.

Asociatividad: Sean $a, b, c, d \in \mathbb{N}$ tal que $a \leq b \leq c \leq d$. Sean f, g, h flechas tal que $a \xrightarrow{f} b, b \xrightarrow{g} c, c \xrightarrow{h} d$, donde $f = (a, b, x), g = (b, c, y), h = (c, d, z)$.
$g \circ f = (a, c, x + y)$
$h \circ g = (b, d, y + z)$
Luego, $h \circ (g \circ f) = (a, d, x + y + z) = (h \circ g) \circ f$■

Identidad izquierda: $id_b \circ f$ donde $a \xrightarrow{f} b$.
$(b, b, 0) \circ (a, b, c) = (a, b, c + 0) = (a, b, c) = f$■

Identidad derecha: $f \circ id_a = (a, b, c) \circ (a, a, 0) = (a, b, c+0) = (a, b, c) = f$■

Un funtor $F : \mathbb{N} \to \mathbb{N}$ podría ser la función sucesor definida así:
$F(a) = S(a)$
$F(a, b, c) = (S(a), S(b), c)$

2.- Sean $f : x \to y, g : y \to z$ funciones de Haskell y x, y, z tipos de Haskell.
$g \circ f : x \to z$.
$F(x) = a \to x$

$F(y) = a \to y$
$F(z) = a \to z$
$F(f) = F(x) \to F(y) = (a \to x) \to (a \to y)$
$F(g) = F(y) \to F(z) = (a \to y) \to (a \to z)$
$F(g \circ f) = (a \to x) \to (a \to z) = F(g) \circ F(f)$■ Vemos que preserva la composición.

Sea $id_x = x \to x$ donde x es un tipo.
$F(x) = a \to x$
$F(id_x) = (a \to x) \to (a \to x) = id_{F(x)}$■ Preserva la identidad.

3.- Sean f, g, id_x definidos como en (2).
$F_1(x) = (a, b) \to x$
$F_1(y) = (a, b) \to y$
$F_1(z) = (a, b) \to z$
$F_1(f) = ((a, b) \to x) \to ((a, b) \to y)$
$F_1(g) = ((a, b) \to y) \to ((a, b) \to z)$
$F_1(g \circ f) = ((a, b) \to x) \to ((a, b) \to z)$
$F_1(g) \circ F_1(f) = ((a, b) \to x) \to ((a, b) \to z) = F_1(g \circ f)$■ Preserva la composición
$F_1(id_x) = ((a, b) \to x) \to ((a, b) \to x) = id_{F_1(x)}$■ Preserva identidades.

$F_2(f) = (a \to (b \to x)) \to (a \to (b \to y))$
$F_2(g) = (a \to (b \to y)) \to (a \to (b \to z))$
$F_2(g \circ f) = (a \to (b \to x)) \to (a \to (b \to z))$
$F_2(g) \circ F_2(f) = (a \to (b \to x)) \to (a \to (b \to z))$■ Preserva la composición.
$F_2(id_x) = (a \to (b \to x)) \to (a \to (b \to x)) = id_{F_2(x)}$■ Preserva la identidad.

4.-

$$\begin{array}{ccc} F_1(a) & \xrightarrow{F_1(f)} & F_1(b) \\ {\scriptstyle \eta_{2a}}\uparrow\downarrow{\scriptstyle \eta_{1a}} & & {\scriptstyle \eta_{2b}}\uparrow\downarrow{\scriptstyle \eta_{1b}} \\ F_2(a) & \xrightarrow{F_2(f)} & F_2(b) \end{array}$$

Las composiciones $\eta_1 \circ \eta_2$ y $\eta_2 \circ \eta_1$ son funciones identidad, entonces solo se cumple la propiedad $\eta_1 \circ \eta_2 = F_2$ y $\eta_2 \circ \eta_1 = F_1$ si F_1 y F_2 son funtores identidad.

5.-
$\eta : [] \Rightarrow []$
$\eta_a : [a] \to [a]$
$xs \mapsto sort\ xs$

η no es una transformación natural. Esto se debe a que sort solo se pue-

de aplicar a listas cuyos elementos sean comparables, es decir, que tengan definidas las operaciones $<, >, =$. Supongamos que tenemos una función de Haskell $f : Int \to Bool$ que decide si un número es par. Supongamos que ℓ es una lista de tipo Int, la operación $map\ f\ (sort\ \ell)$ está definida, pero $sort\ (map\ f\ \ell)$ no está definida, porque no podemos ordenar tipo $Bool$. Entonces $(map\ f)\ (sort\ \ell) \neq sort\ ((map\ f)\ \ell)$.

6.-
Definamos la función $Maybe\_fun$ así:
$Maybe\_fun :: (a \to b) \to Maybe\ a$
$Maybe\_fun\ f\ Nothing = Nothing$
$Maybe\_fun\ f\ (Just\ a) = Just\ (f\ a)$

Utilizaré el término $f(x)$ para describir el resultado de aplicar f a x en vez de $f\ x$, como sería en Haskell. Sea $f : a \to b$, veremos que el siguiente diagrama conmuta:

$$\begin{array}{ccc} Maybe\ a & \xrightarrow{Maybe\_fun f} & Maybe\ b \\ \downarrow{\scriptstyle \eta_a} & & \downarrow{\scriptstyle \eta_b} \\ [a] & \xrightarrow{map\ f} & [b] \end{array}$$

Caso $Nothing$:
$map\ f\ (\eta_a\ Nothing) = map\ f\ [] = []$
$\eta_b\ (Maybe\_fun\ f\ Nothing) = \eta_b\ Nothing = []$
Por lo tanto $map\ f\ (\eta_a\ Nothing) = \eta_b\ (Maybe\_fun\ f\ Nothing)$■

Caso x:
$map\ f\ (\eta_a\ (Just\ x)) = map\ f\ [x] = [f(x)]$
$\eta_b\ (Maybe\_fun\ f\ (Just\ x)) = \eta_b\ (Just\ f(x)) = [f(x)]$
Por lo tanto $map\ f\ (\eta_a\ (Just\ x)) = \eta_b\ (Maybe\_fun\ f\ (Just\ x))$■