Evaluación perezosa. Redes de procesos

Evaluación perezosa

- Evaluación impaciente (*eager*): el evaluador hace *todo lo que puede*. Corresponde a llamada–por–valor.
- Evaluación perezosa (*lazy*): el evaluador hace *solamente lo preciso*. Corresponde a llamada–por–necesidad.

Argumentos estrictos

$$f\ x\ y\ =\ \textbf{if}\ x < 10\ \textbf{then}\ x\ \textbf{else}\ y$$

- La función es estricta en x
- Problemas cuando no termina: $g\ (-3)$,

$$g\ x\ =\ \textbf{if}\ x < 0\ \textbf{then}\ g\ (x-1)\ \textbf{else}\ x * x$$

- Un evaluador impaciente tampoco terminaría de evaluar la expresión $f\ 4\ (g\ (-3))$
- Un evaluador perezoso daría el resultado 4.
- Cualquier evaluador no terminaría de evaluar la expresión $f\ (g\ (-3))\ 8$ y ello es debido a que la función es estricta en el primer argumento.

$$\begin{aligned} estaEl3\ [\,] &= False \\ estaEl3\ (x : xs) &= (3{==}\ x)\ ||\ estaEl3\ xs \end{aligned}$$

- Con evaluador perezoso:

$$\begin{aligned} & estaEl3\ [3,4,5] \\ \Longrightarrow\ & \\ & (3{==}\ 3)\ ||\ estaEl3\ [4,5] \\ \Longrightarrow\ & \\ & True \end{aligned}$$

- Con un evaluador impaciente:

$$\begin{aligned} & estaEl3\ [3,4,5] \\ \Longrightarrow\ & \\ & 3{==}\ 3\ ||\ estaEl3\ [4,5] \\ \Longrightarrow\ & \\ & 3{==}\ 3\ ||\ 3{==}\ 4\ ||\ estaEl3\ [5] \\ \Longrightarrow\ & \\ & 3{==}\ 3\ ||\ 3{==}\ 4\ ||\ 3{==}\ 5\ ||\ estaEl3\ [\,] \\ \Longrightarrow\ & \\ & 3{==}\ 3\ ||\ 3{==}\ 4\ ||\ 3{==}\ 5\ ||\ False \\ \Longrightarrow\ & \\ & 3{==}\ 3\ ||\ 3{==}\ 4\ ||\ False \\ \Longrightarrow\ & \\ & 3{==}\ 3\ ||\ False \\ \Longrightarrow\ & \\ & True \end{aligned}$$

Procesando estructuras infinitas

```
desde :: Integer → [Integer]
desde x = x : desde (x + 1)
```

$$[x,\ (1+)\ x,\ (1+)\ (\ (1+)\ x\ ),\ \ldots\ ]$$

- *iterate* de PRELUDE:

```
iterate :: (a → a) → a → [a]
iterate f x = x : iterate f (f x)
```

- $iterate\ (1+)\ x \implies x\ :\ iterate\ (1+)\ (x+1)$

$$desde \equiv iterate\ (1+)$$

```
suma :: Integer → [Integer] → Integer
suma n (x : xs) = if n== 0 then 0 else x + suma (n − 1) xs
```

$$\begin{array}{l}
suma\ 2\ (desde\ 1) \\
\Longrightarrow \\
suma\ 2\ (1\ :\ desde\ 2) \\
\Longrightarrow \\
1\ +\ suma\ 1\ (desde\ 2) \\
\ldots \\
\Longrightarrow \\
1\ +\ (2\ +\ 0) \\
\Longrightarrow \\
3
\end{array}$$

```
selec n (x : xs) = if n== 1 then x else selec (n − 1) xs
```

$$\begin{array}{l}
selec\ 3\ (desde\ 4) \\
\Longrightarrow \\
selec\ 3\ (4\ :\ desde\ 5) \\
\Longrightarrow \\
selec\ 2\ (desde\ 5) \\
\Longrightarrow \\
selec\ 2\ (5\ :\ desde\ 6) \\
\Longrightarrow \\
selec\ 1\ (desde\ 6) \\
\Longrightarrow \\
selec\ 1\ (6\ :\ desde\ 7) \\
\Longrightarrow \\
6
\end{array}$$

- La evaluación perezosa significa: *haz sólo lo que te pida un patrón a la izquierda de una ecuación o cualificador* (**where** o **let**)

Listas parciales y listas infinitas

Aproximaciones o listas parciales

$$filter\ (<10)\ (map\ (+6)\ [1..])\ ?$$

- $[1..]$ es vista como el límite de las aproximaciones

$$\bot,\ 1:\bot,\ 1:2:\bot,\ \ldots$$

$s_n \uparrow [1..]$ ($\uparrow$ se lee *tiende a*), donde

$$s_n = 1:2:\ \ldots\ :n:\bot$$

- Si f es una función computable entonces es continua; es decir,

$$xs_n \uparrow xs \quad \Rightarrow f\ xs_n \uparrow f\ xs$$

- Si f es estricta en su argumento, entonces $f\ \bot = \bot$

$$\begin{array}{ll} map\ (+6)\ \bot & = \bot \\ map\ (+6)\ (1:\bot) & = 7:\bot \\ map\ (+6)\ (1:2:\bot) & = 7:8:\bot \\ \ldots & \end{array}$$

y entonces, por ser computable la función $map\ (+6)$:

$$map\ (+6)\ [1..] = [7..]$$

- Si filtramos tomando los menores que 10:

$$\begin{array}{ll} filter\ (<10)\ (map\ (+6)\ \bot\ ) & = \bot \\ filter\ (<10)\ (map\ (+6)\ (1:\bot)) & = 7:\bot \\ filter\ (<10)\ (map\ (+6)\ (1:2:\bot)) & = 7:8:\bot \\ filter\ (<10)\ (map\ (+6)\ (1:2:3:\bot)) & = 7:8:9:\bot \\ filter\ (<10)\ (map\ (+6)\ (1:2:3:4:\bot)) & = 7:8:9:\bot \\ \ldots & \end{array}$$

$$filter\ (<10)\ (map\ (+6)\ [1..]) = 7:8:9:\bot$$

Inducción sobre listas parciales

- Constructores $[\,]$, $\bot$ y $(:)$ Si a es un tipo base, sea $[a]'$ el tipo formado por las listas parciales sobre el tipo base a.
- Una propiedad p sobre listas parciales

$$p \;::\; [a]' \to \mathit{Bool}$$

puede ser demostrada por inducción estructural:

$$\begin{array}{l} \forall\, u \,\centerdot\, u \;::\; [a]' \,\centerdot\, p\; u \\ \equiv \\ p\,[\,], \\ p\,\bot, \\ \forall\, x, u \,\centerdot\, x :: \;\; a,\; u :: \;\; [a]' \,\centerdot\, p\; u \qquad \Rightarrow\; p\,(x : u) \end{array}$$

- Ejemplo.- Asociatividad en listas parciales.

$$u +\!\!+ (\, v +\!\!+ w \,) \;\equiv\; (u +\!\!+ v \,) +\!\!+ w$$

$\bot +\!\!+ (v +\!\!+ w)$
$\equiv$! $+\!\!+$ es estricta es su primer argumento
$\bot$
$\equiv$! $+\!\!+$ es estricta es su primer argumento
$\bot +\!\!+ w$
$\equiv$! $+\!\!+$ es estricta es su primer argumento
$(\bot +\!\!+ v) +\!\!+ w$

- No todas las igualdades son demostrables por inducción; por ejemplo:

$$\mathit{iterate}\; f\; x \;\equiv\; x : \mathit{map}\; f\; (\mathit{iterate}\; f\; x)$$

Lema.-*Siendo las listas xs e ys infinitas o parciales:*

$$xs \;\equiv\; ys \;\Leftrightarrow\; \forall\, n \,\centerdot\, n > 0 \,\centerdot\, \mathit{aprox}\; n\; xs \;\equiv\; \mathit{aprox}\; n\; ys$$

$$\begin{array}{ll} \mathit{aprox}\; (n+1)\; [\,] & = [\,] \\ \mathit{aprox}\; (n+1)\; (x : xs) & = x : \mathit{aprox}\; n\; xs \end{array}$$

¡ no existe ecuación para el caso$\mathit{aprox}\;\; 0\;\; _$! Luego

$$\begin{array}{l} \textsc{Main}\!> \;\mathit{aprox}\; 3\; [1..] \\ 1 : 2 : 3 : \bot \end{array}$$

y por tanto el resultado es una lista parcial cuando la lista sea infinita.

Procesando estructuras infinitas

La criba de Eratóstenes

- La función *criba* toma una lista y selecciona solamente aquellos números que son primos con la cabeza de la lista

$$criba\ [2..] \Longrightarrow [3,5,7,9,11,13,15,17,19,21,...]$$

```
criba :: [Int] → [Int]
criba (p : xs) = [ x | x ← xs, p `noDivideA` x ]
  where m `noDivideA` n = mod n m ≠ 0
```

$$\begin{array}{ll}
[2..] & \Longrightarrow [2,3,4,5,6,7,8,9,10,11,12,...] \\
criba\ [2..] & \Longrightarrow [3,5,7,9,11,13,15,17,19,21,...] \\
criba\ (criba\ [2..]) & \Longrightarrow [5,7,11,13,17,19,...] \\
criba\ (criba\ (criba\ [2..])) & \Longrightarrow [7,11,13,17,19,...] \\
... &
\end{array}$$

pero tales listas se obtienen iterando

$$\begin{array}{l}
iterate\ criba\ [2..] \\
\Longrightarrow \quad \text{! definición de } iterate \\
[2..]\ :\ iterate\ criba\ [3,5,7,9,11,13,15,17,19,21,...] \\
\Longrightarrow \\
[2..]\ :\ [3,...]\ :\ iterate\ criba\ [5,7,11,13,17,19,...] \\
\Longrightarrow \\
\quad ...
\end{array}$$

y tendremos la lista de los primos tomando las cabezas de tales listas:

```
primos :: [Integer]
primos = map head (iterate criba [2..])
```

Redes de procesos

- Podemos considerar los procesos como funciones que:
  - consumen datos (argumentos)
  - producen valores para otras funciones
- La evaluación perezosa corresponde a corrutinas particulares de los procesos
  - los procesos son suspendidos hasta que se solicita la evaluación (por algún proceso) de cierta expresión.
  - la petición de evaluación la hace la comparación de patrones y una expresión se evalúa sólo parcialmente.

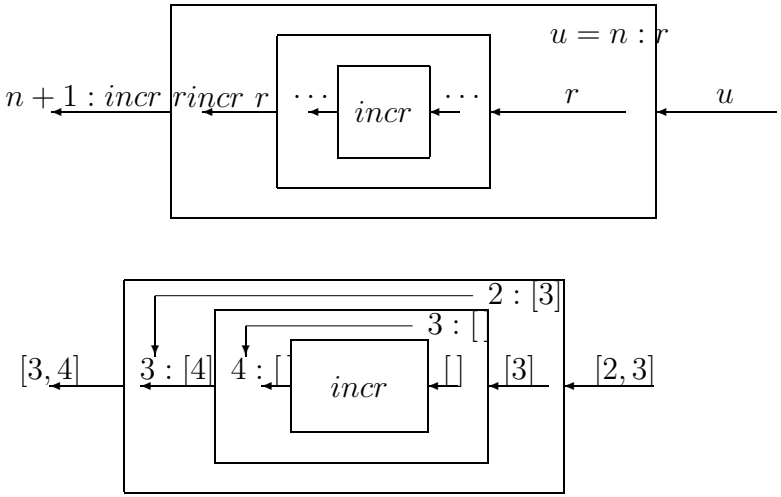
$$incr :: [Int] \to [Int]$$

$$\begin{aligned} incr\ [\,] &= [\,] \\ incr\ (n : ns) &= (n+1) : incr\ ns \end{aligned}$$

$$pos :: [Int]$$

$$pos = 0 : incr\ pos$$

la llamada $selec\ 3\ pos$ crea los procesos solamente cuando son necesarios:



$selec\ 3\ pos$

$\Longrightarrow$! crea una versión de pos

$selec\ 3\ (0 :\ incr\ pos)$

$\Longrightarrow$

$selec\ 2\ (incr\ pos)$

$\Longrightarrow$! invoca a pos para separar el canal

$selec\ 2\ (incr\ (0 : incr\ pos))$

$\Longrightarrow$

$selec\ 2\ (1 : incr(incr\ pos))$

$\Longrightarrow$

$selec\ 1\ (incr(incr\ pos))$

$\ldots$

$\Longrightarrow$

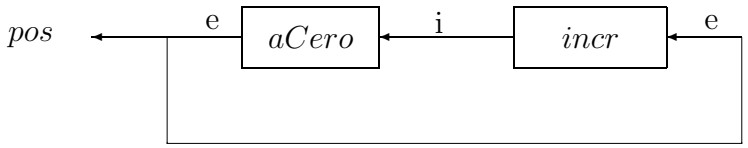
2

Los procesos mueren al terminar la evaluación.

Redes finitas de procesos

$$aCero :: [Int] \to [Int]$$

$$aCero = (0 :)$$



$$pos = e \textbf{ where } (i, e) = (\ incr\ e,\ aCero\ i\ )$$

$$\begin{aligned}
& pos = e \textbf{ where } (i, e) = (\ incr\ e,\ aCero\ i\ ) \\
\equiv\ & \text{! eliminando } e \\
& pos = aCero\ i \textbf{ where } i = incr\ pos \\
\equiv\ & \text{! eliminando } i \\
& pos = aCero\ (incr\ pos) \\
\equiv\ & \text{! def. } aCero \\
& pos = 0 : incr\ pos
\end{aligned}$$

Sucesiones en general

- Sea la sucesión definida por:

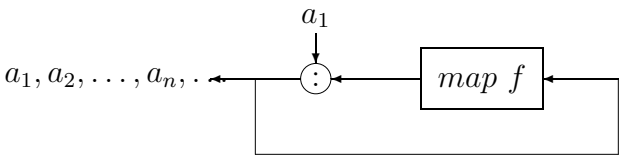
$$a_1 = k$$

$$a_n = f(a_{n-1})$$

- Puede construirse por medio de:

$$s = a_1 : map\ f\ s$$

- Y como red sería:



La red que calcula el Triángulo de Pascal

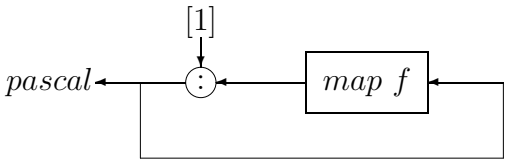
$$
\begin{array}{c}
1 \\
1\ 1 \\
1\ 2\ 1 \\
1\ 3\ 3\ 1 \\
\ldots
\end{array}
$$

Como una sucesión general:

$$pascal = [1] : map\ f\ pascal$$
$$\textbf{where}\ f\ c = zipWith\ (+)\ (0 : c)\ (c +\!\!+ [0])$$

o también

$$pascal = [1] : u\ \textbf{where}\ \{u = map\ f\ pascal;\ f = ...\}$$



La red que calcula la sucesión de números primos

- Ya se ha visto que:

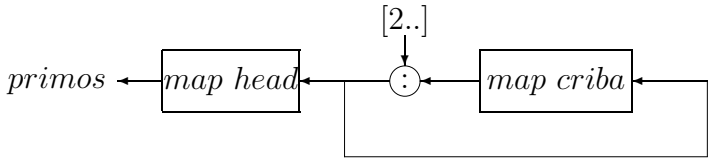
$$primos = map\ head\ (iterate\ criba\ [2..])$$

y considerando que

$$iterate\ criba\ [2..]$$
$$\equiv\ !$$
$$lprimos\ \textbf{where}\ lprimos = [2..] : map\ criba\ lprimos$$

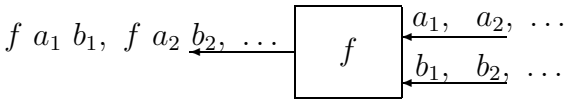
podemos escribir la solución como

$$primos = map\ head\ lprimos$$
$$\textbf{where}\ lprimos = [2..] : map\ criba\ lprimos$$



Procesos con varias entradas

- Un proceso puede tener:
  - varios canales de entrada
  - uno de salida.
- Sea el proceso que toma la información de dos canales de entrada y devuelve sobre el canal de salida la aplicación de cierta función f dos a dos:



que puede definirse en la forma:

$$zipWith\ f\ [a_1,\ a_2,\ \ldots]\ [b_1,\ b_2,\ \ldots]$$

donde

$$zipWith\ ::\ (a \to b \to c) \to [a] \to [b] \to [c]$$

$$zipWith\ f\ (x : xs)\ (y : ys) = f\ x\ y\ :\ zipWith\ f\ xs\ ys$$
$$zipWith\ \_\ \_\ \_\ = [\,]$$

- Ejemplo.- A partir de las listas

$$\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \quad \ldots \\ 2 & 4 & 6 & 8 & 10 & \ldots \end{array}$$

multiplicando columna a columna tenemos la sucesión s_1

$$s_1 = zipWith\ (*)\ [1..]\ [2, 4..]$$

La lista de factoriales

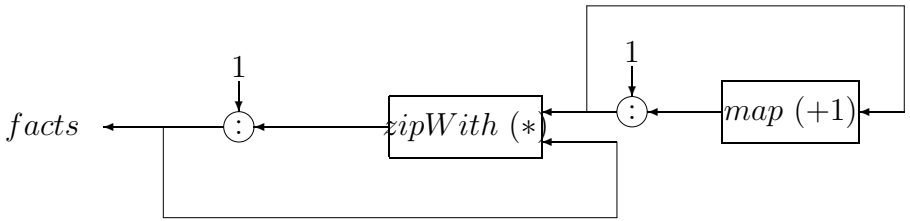
$$\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & \ldots \\ 0! & 1! & 2! & 3! & 4! & \ldots \end{array}$$

si multiplicamos columna a columna obtenemos

$$1! \quad 2! \quad 3! \quad 4! \quad 5! \quad \ldots$$

donde falta $0!$ para ser igual a la segunda fila:

$$facts = 1 : zipWith\ (*)\ [1..]\ facts$$



Números de Fibonacci

$$\begin{array}{ccccccccc} 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & \ldots \\ 1 & 1 & 2 & 3 & 5 & 8 & 13 & \ldots \end{array}$$

y ahora sumamos columna a columna:

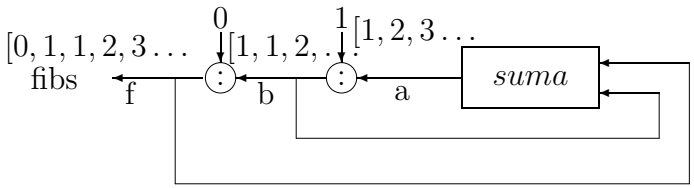
$$1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad \ldots$$

donde faltan los dos primeros elementos para ser igual a la primera fila:

$$fibs \equiv 0\ 1\ 1\ 2\ 3\ 5\ 8\ 13\ \ldots$$

tendremos (tomando $suma = zipWith\ (+)$)

$$fibs = 0 : 1 : suma\ fibs\ (tail\ fibs)$$



$$fibs = f\ \textbf{where}\ (a, b, f) = (suma\ b\ f,\ 1 : a,\ 0 : b\ )$$

Es fácil demostrar que las dos soluciones son la misma por transformación.

Sucesiones en general

• Sean $k > 0$, las constantes $a_0, a_1, \ldots, a_{k-1}$ y la sucesión $\{c_n\}_{n \geq k}$. Consideremos ahora la sucesión dada por las ecuaciones

$$\begin{aligned} x_0 &= a_0 \\ \ldots &= \ldots \\ x_{k-1} &= a_{k-1} \\ x_n &= f(c_n, x_{n-1}, \ldots, x_{n-k}), \qquad n \geq k \end{aligned}$$

donde los k primeros términos de la sucesión son constantes conocidas y el resto sigue el esquema descrito.

• Escribimos las filas

$$\begin{array}{lllll} c_k & c_{k+1} & \ldots & c_i & c_{i+1} & \ldots \\ x_0 & x_1 & \ldots & x_i & x_{i+1} & \ldots \\ x_1 & x_2 & \ldots & x_{i+1} & x_{i+2} & \ldots \\ \ldots & & & & & \\ x_{k-1} & x_k & \ldots & x_{i+k-1} & x_{i+k} & \ldots \end{array}$$

y aplicando f con los elementos de cada columna como argumentos, se obtiene la nueva fila

$$x_k \quad x_{k+1} \quad \ldots \quad x_{i+k} \quad x_{i+k+1} \quad \ldots$$

que nuevamente resulta ser la sucesión buscada a partir del término k-ésimo.

Ahora es fácil construir una función HASKELL que la reproduzca (en *pseudo-código*)

$$\begin{aligned} s = a_0 : a_1 : \ldots : a_{k-1} : zipWithK1\, f\, [c_n \mid n \leftarrow [k..]] \\ s \\ (drop\ 1\ s) \\ (drop\ 2\ s\ ) \\ \ldots \\ (drop\ (k-1)\ s) \end{aligned}$$

• La función $zipWithK1$ no existe, existen $zipWith$, $zipWith3$ y en el módulo LIST hasta $zipWith7$.

• En cualquier caso, un $zipWith3$ puede descomponerse en dos $zipWith$, etc.

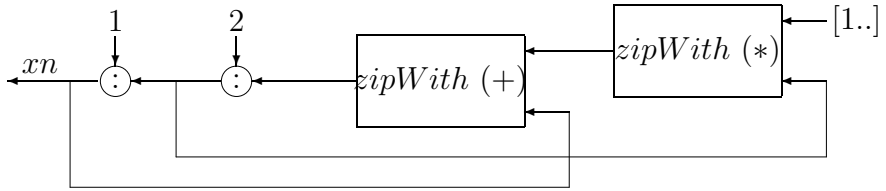
• Ejemplo.- Sea la sucesión x_n definida como:

$$x_0 = 1,\ x_1 = 2,\ x_{n+2} = (n+1) * x_{n+1} + x_n$$

1	2	3	4	...
1	2	3	8	...
2	3	8	27	...

Si ahora multiplicamos la primera por la tercera y sumamos la segunda, el resultado que se obtiene es:

3	8	27	116	...



$$xn = 1 : 2 : zipWith3\ (\lambda\ n\ x\ x' \rightarrow n * x' + x\ )\ [1..]\ xn\ (tail\ xn)$$

donde, como vemos, la sucesión c_n del caso general aquí queda reducida a la sucesión $[1..]$.

• Descomponiendo el $zipWith3$:

$$xn = 1 : 2 : zipWith\ (+)\ (zipWith\ (*)\ [1..]\ (tail\ xn))\ xn$$

• A veces queda más claro nombrando cada parte de la red:

$$\begin{aligned} xn &= sol\ \textbf{where} \\ sol &= 1 : sol1 \\ sol1 &= 2 : sol2 \\ sol2 &= zipWith\ (+)\ sol3\ sol \\ sol3 &= zipWith\ (*)\ [1..]\ sol1 \end{aligned}$$

• Ejercicio.- Escribir una red de procesos y la función HASKELL correspondiente para la sucesión

$$y_0 = 1,\ y_1 = 1,\ y_{n+2} = n * y_{n+1} + 3 * y_n$$

Los números de Hamming

• Sea $\mathcal{H}$ (los números de Hamming) el menor subconjunto de $\mathbb{N}$ verificando los axiomas:

$$Ax1: \ 1 \in \mathcal{H}$$
$$Ax2: \ \forall x \ . \ x \in \mathcal{H} \ \Rightarrow \ 2x, 3x, 5x \in \mathcal{H}$$

Se trata de obtener la secuencia ordenada de elementos de $\mathcal{H}$:

$$1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 8,\ 9,\ 10,\ 12,\ 15,\ \ldots$$

Si denotamos con h la lista de elementos de $\mathcal{H}$, es trivial que mezclando convenientemente los flujos

$$map\ (2*)\ h \quad map\ (3*)\ h \quad map\ (5*)\ h$$

se obtienen números de Hamming;

$$mezcla\ u\ v\ w\ =\ (u\ <|>\ v)\ <|>\ w$$

where

$$\begin{aligned}(x:xs)\ <|>\ (y:ys)\ |\ x == y\ &=\ x\ :\ (xs\ <|>\ ys)\\ |\ x < y\ &=\ x\ :\ (xs\ <|>\ (y:ys))\\ |\ y < x\ &=\ y\ :\ ((x:xs)\ <|>\ ys)\end{aligned}$$

y finalmente, si añadimos el 1 como elemento inicial.

$$h\ =\ 1\ :\ mezcla\ (map\ (2*)\ h)\ (map\ (3*)\ h)\ (map\ (5*)\ h)$$