# 21 SOLUCIONES A EJERCICIOS

## 21.1. INTRODUCCIÓN A HASKELL

**Solución al Ejercicio 2.14** (pág. 33).– Nuestra función deberá verificar

$$\forall\, n\ sep\ n\ \geq\ 0\ \centerdot\ aEntero\,[a_n,\ a_{n-1},\ \ldots,\ a_0]\ =\ \textstyle\sum_{i=0}^{n} a_i\ 10^i$$

Para $n = 0$ tenemos trivialmente

$$aEntero\,[d]\ =\ d$$

pero también deberá tenerse, para $n > 0$

$$\begin{array}{ll}
 & aEntero\,[a_{n+1},\ a_n,\ a_{n-1},\ \ldots,\ a_0] \\
\equiv & \\
 & \displaystyle\sum_{i=0}^{n+1} a_i\ 10^i \\
\equiv & \\
 & \displaystyle a_{n+1}10^{n+1} + a_n 10^n\ +\ \sum_{i=0}^{n-1} a_i\ 10^i \\
\equiv & \\
 & \displaystyle (a_{n+1}10 + a_n)10^n + \sum_{i=0}^{n-1} a_i\ 10^i \\
\equiv & \\
 & aEntero\,[a_{n+1}10 + a_n,\ a_{n-1},\ \ldots,\ a_0]
\end{array}$$

de donde tenemos la función

$$\begin{array}{ll}
aEntero\,[d] & = d \\
aEntero\,(d : m : r) & = aEntero\ (\ (10 * d + m) : r\ )
\end{array}$$

cuya corrección es inmediata. Obsérvese que la función utiliza directamente la cabeza de la lista como acumulador (ver Capítulo 16). Podríamos haber obtenido otra versión utilizando un acumulador directamente

$$
\begin{array}{lll}
aEntero\ xs & = & aEntero'\ xs\ 0 \\
 & & \\
aEntero'\ [\,] \quad p & = & p \\
aEntero'\ (x : xs)\ p & = & aEntero'\ xs\ (10 * p + x)
\end{array}
$$

que verifica trivialmente la misma ecuación, ya que se tiene (por inducción)

$$\forall\ n\ \centerdot\ n\ \geq\ 0\ \centerdot\ aEntero'\ [a_n, a_{n-1},\ \ldots,\ a_0]\ p\ =\ p\ 10^n\ +\ \textstyle\sum_{i=0}^{n}\ a_i\ 10^i$$

Veamos ahora la recíproca

$$
\begin{array}{lll}
aLista\ 0 & = & [0] \\
aLista\ m & = & aLista'\ m\ [\,] \\
aLista'\ m\ xs & = & \textbf{if}\ m \leq 0\ \textbf{then} \\
 & & \quad xs \\
 & & \textbf{else} \\
 & & \quad aLista'\ (m\ `div`\ 10)\ (m\ `mod`\ 10\ :\ xs)
\end{array}
$$

para la cual se demuestra

$$\forall\, n \centerdot\ n\ \geq\ 0\ ,\ a_n\ >\ 0\ ,\ (\forall\, k\ \centerdot\ 0\ \leq\ k\ \leq\ n\ \centerdot\ 0\ \leq\ a_k \leq\ 9)\centerdot$$
$$aLista'\ (\sum_{i=0}^{n}\ a_i\ 10^i) xs\ =\ [a_n\ , a_{n-1},\ \ldots,\ a_0] +\!\!+\ xs$$

por inducción sobre $n$ (donde $(+\!\!+)$ es la concatenación de listas). Para $n = 0$ es trivial; el paso inductivo sería:

$$
\begin{array}{ll}
 & aLista'\ (\sum_{i=0}^{n+1}\ a_i\ 10^i)\ xs \\
\equiv & \{\, 2) aLista' \,\} \\
 & aLista'\ (\sum_{i=0}^{n}\ a_{i+1}\ 10^i)\ (a_0\ :\ xs) \\
\equiv & \{\, \text{hipótesis de inducción, y ya que } (\forall k\ \centerdot\ 0 \leq k \leq n \centerdot 0 \leq a_k \leq 9) \,\} \\
 & [a_{n+1},\ a_n,\ \ldots,\ a_1] +\!\!+\ (a_0\ :\ xs) \\
\equiv & \{\, \text{propiedad de } (+\!\!+) \text{ y } (:) \,\} \\
 & [a_{n+1},\ a_n,\ \ldots,\ a_1,\ a_0] +\!\!+\ xs
\end{array}
$$

**Solución al Ejercicio 2.28** (pág. 46).–

$$
\begin{array}{lll}
(|>) & :: & [Integer\ \to\ Integer]\ \to\ Integer\ \to\ [Integer] \\
[\,] \quad |>\ \_ & = & [\,] \\
(f : fs)\ |>\ x & = & f\ x\ :\ (fs\ |>\ x)
\end{array}
$$

**Solución al Ejercicio 2.29** (pág. 46).–

```
esMúltiploDe       :: Integer → Integer → Bool
a `esMúltiploDe` b  = (a `mod` b == 0)

esBisiesto    :: Integer → Bool
esBisiesto x  = x `esMúltiploDe` 4 &&
                (not(x `esMúltiploDe` 100) || x `esMúltiploDe` 400)
```

**Solución al Ejercicio 2.31** (pág. 46).–

```
aLaDerechaDe       :: Integer → Integer → Integer
x `aLaDerechaDe` y  =  x * 10 + y
```

**Solución al Ejercicio 2.32** (pág. 47).–

```
resto                  :: Integer → Integer → Integer
resto x y | x < y      =  x
          | otherwise  =  resto (x - y) y
```

**Solución al Ejercicio 2.33** (pág. 47).–

```
cociente                  :: Integer → Integer → Integer
cociente x y | x < y      =  0
             | otherwise  =  1 + cociente (x - y) y
```

**Solución al Ejercicio 2.34** (pág. 47).–

```
sumDesdeHasta       :: Integer → Integer → Integer
sumDesdeHasta a b   = sumAux (min a b) (max a b)
  where sumAux x y | x == y     =  x
                   | otherwise  =  x + sumAux (x + 1) y
```

**Solución al Ejercicio 2.38** (pág. 47).– La siguiente es una definición directa aunque poco eficiente (realiza dos llamadas recursivas):

```
fibonacci          :: Integer → Integer
fibonacci 0        = 0
fibonacci 1        = 1
fibonacci (n + 2)  = fibonacci n + fibonacci (n + 1)
```

**Solución al Ejercicio 2.41** (pág. 48).–

```
esCapicúa                  :: Integer → Bool
esCapicúa x
  | x ≥ 1000 && x ≤ 9999   = d1 == d4&& d2 == d3
  | otherwise              = error "número de cifras incorrecto"
  where
    d1 = x `mod` 10
    d2 = (x `div` 10) `mod` 10
    d3 = (x `div` 100) `mod` 10
    d4 = (x `div` 1000)
```

**Solución al Ejercicio 2.44** (pág. 48).– Combinando adecuadamente las soluciones al Ejercicio 2.32 y al Ejercicio 2.33 se obtiene:

```
-- trocear x = (cociente x 10, resto x 10)
trocear :: Integer → (Integer, Integer)
trocear x | x < 10   = (0, x)
          | otherwise = (1 + c, r) where (c, r) = trocear (x − 10)
```

Obsérvese que

```
MAIN> trocear 5
(0,5) :: (Integer, Integer)
```

**Solución al Ejercicio 2.45** (pág. 48).– Para definir *concatenar* se puede usar la función *trocear* del Ejercicio 2.44:

```
concatenar     :: Integer → Integer → Integer
concatenar x 0 = x
concatenar x y = r `aLaDerechaDe` (concatenar x c) where (c, r) = trocear y
```

## 21.2. DEFINICIONES DE TIPO

**Solución al Ejercicio 4.4** (pág. 78).– Definimos solo la primera. La segunda se define de modo similar:

```
área                  :: Figura → Float
área (Círculo r)      = pi * r * r
área (Cuadrado l)     = l * l
área (Rectángulo b h) = b * h
área Punto            = 0
```

**Solución al Ejercicio 4.5** (pág. 78).–

```
raíces          :: Float → Float → Float → Resultado
raíces a b c
  | a == 0      = error"Ecuación de primer grado"
  | disc == 0   = UnaReal x
  | disc > 0    = DosReales x1 x2
  | disc < 0    = DosComplejas c1 c2
  where
    disc    =b * b - 4 * a * c
    rDisc   =sqrt (abs disc)
    coc     =2 * a
    x       = - b/coc
    x1      =(-b + rDisc)/coc
    x2      =(-b - rDisc)/coc
    c1re    = - b/coc
    c1im    =rDisc/coc
    c2re    =c1re
    c2im    = - c1im
    c1      =c1re :- c1im
    c2      =c2re :- c2im
```

**Solución al Ejercicio 4.25** (pág. 103).– Se trata se probar que los 6 valores

$$\begin{array}{lll} max3\ x\ y\ z & max3\ x\ z\ y & max3\ y\ z\ x \\ max3\ y\ x\ z & max3\ z\ x\ y & max3\ z\ y\ x \end{array}$$

son todos iguales; las igualdades (que notaremos $(3f)$) que involucran mantener fijo el tercer argumento son fáciles de demostrar, pues:

$$\begin{array}{ll} & max3\ x\ y\ z \\ \equiv & \{\text{ definición de } max3\ \} \\ & max\ (max\ x\ y)\ z \\ \equiv & \{\text{ simetría de } max\ \} \\ & max\ (max\ y\ x)\ z \\ \equiv & \{\text{ definición de } max3\ \} \\ & max3\ y\ x\ z \end{array}$$

con lo cual:

$$\begin{array}{lcl} max3\ x\ y\ z & = & max3\ y\ x\ z \\ max3\ x\ z\ y & = & max3\ z\ x\ y \\ max3\ y\ z\ x & = & max3\ z\ y\ x \end{array}$$

y solamente sería necesario probar la igualdad que involucra mantener fijo el primer argumento y la que involucra mantener el segundo

$$(1f) \qquad max3\ x\ y\ z \quad = \quad max3\ x\ z\ y$$

$$(2f) \qquad max3\ x\ y\ z \quad = \quad max3\ z\ y\ x$$

ya que, por la transitividad de la relación de igualdad, se obtendría la igualdad restante:

$$max3\ x\ z\ y \quad = \quad max3\ z\ y\ x$$

La igualdad $(1f)$ se puede demostrar distinguiendo los posibles valores, mientras que finalmente la $(2f)$ es fácil:

$$\begin{array}{ll} & max3\ x\ y\ z \\ \equiv & \{\,(3f)\,\} \\ & max3\ y\ x\ z \\ \equiv & \{\,(1f)\,\} \\ & max3\ y\ z\ x \\ \equiv & \{\,(3f)\,\} \\ & max3\ z\ y\ x \end{array}$$

**Solución al Ejercicio 4.26** (pág. 103).– Una primera posibilidad es:

```
-- Suma de complejos
('c', x, y)  >+< ('c', x', y')  = ('c', x + x', y + y')
('c', x, y)  >+< ('p', m, a)    = ('c', x + m * cos a, y + m * sin a)
('p', m, a)  >+< ('c', x, y)    = ('c', x + m * cos a, y + m * sin a)
('p', m, a)  >+< ('p', m', a')  =
   ('c', m * cos a + m' * cos a', m * sin a + m' * sin a')

-- Producto de complejos
('p', m, a)  >*< ('p', m', a')  = ('p', m * m', a + a')
('p', m, a)  >*< ('c', x, y)    =
   ('p', m * sqrt (x ↑ 2 + y ↑ 2), a + atan2 y x)
-- o bien ('c', x * x' - y * y', x * y' + y * x')
-- where {x' = m * cos a; y' = m * sin a}
('c', x, y)  >*< ('p', m, a)  =
   ('p', m * sqrt (x ↑ 2 + y ↑ 2), a + atan2 y x)
-- o bien ('c', x * x' - y * y', x * y' + y * x')
-- where {x' = m * cos a; y' = m * sin a}
('c', x, y)  >*< ('c', x', y')  =
   ('p', sqrt (x ↑ 2 + y ↑ 2) * sqrt (x' ↑ 2 + y' ↑ 2),
         atan2 y x + atan2 y' x')
-- o bien ('c', x * x' - y * y', x * y' + y * x')

-- Diferencia y Cociente de complejos
...
```

También podría utilizarse una función de conversión:

```
conver :: Complejo → Complejo
conver ('c', x, y)  = ('p', sqrt (x ↑ 2 + y ↑ 2), atan2 y x)
conver ('p', m, a)  = ('c', m * cos a, m * sin a)
```

y a partir de ella definir:

```
-- Suma de complejos
('c', x, y)  >+< ('c', x', y')  = ('c', x + x', y + y')
('c', x, y)  >+< ('p', m, a)    = ('c', x, y) >+< conver ('p', m, a)
('p', m, a)  >+< ('c', x, y)    = conver ('p', m, a) >+< ('c', x, y)
('p', m, a)  >+< ('p', m', a')  = conver ('p', m, a) >+< conver ('p', m', a')

-- Producto, Diferencia y Cociente de complejos
...
```

**Solución al Ejercicio 4.27** (pág. 103).– Utilizando una función de conversión:

```
conver :: Complejo → Complejo
conver (Carte x y)  = Polar (sqrt (x ↑ 2 + y ↑ 2)) (atan2 y x)
conver (Polar m a)  = Carte (m * cos a)             (m * sin a)
```

se pueden definir:

```
-- Suma de complejos
c1@(Carte x y)  >+< c2@(Carte x' y')  = Carte (x + x') (y + y')
c1@(Carte x y)  >+< c2@(Polar m a)    = c1 >+< conver c2
c1@(Polar m a)  >+< c2@(Carte x y)    = conver c1 >+< c2
c1@(Polar m a)  >+< c2@(Polar m' a')  = conver c1 >+< conver c2

-- Producto, Diferencia y Cociente de complejos
...
```

**Solución al Ejercicio 4.28** (pág. 103).–

```
total             :: Número → Int
total (Dígito n)  =  n
total (i :↑: d)   =  total i + total d
```

## 21.3. EL SISTEMA DE CLASES DE HASKELL

**Solución al Ejercicio 5.5** (pág. 129).–

```
instance Ord Nat where
  Cero ≤ _       =  True
  Suc _ ≤ Cero   =  False
  Suc n ≤ Suc m  =  n ≤ m
```

**Solución al Ejercicio 5.6** (pág. 129).– Como la igualdad es la estructural y el orden es el predeterminado por la enumeración, en HASKELL bastaría con:

$$\begin{aligned}&\textbf{data}\ ColSimp\ =\ Violeta \mid Añil \mid Azul \mid Verde \mid\\ &\qquad\qquad Amarillo \mid Naranja \mid Rojo\\ &\qquad\qquad \textbf{deriving}\ (Eq,\ Ord)\end{aligned}$$

**Solución al Ejercicio 5.7** (pág. 129).– Se introduce un tipo $Marcador$ para el que se sobrecargan las funciones $(+)$ y $(==)$:

$$\textbf{data}\ Marcador\ =\ M\ (Int, Int, Int, Int, Int, Int, Int)$$

$$\begin{aligned}&\textbf{instance}\ Num\ Marcador\ \textbf{where}\\ &\ M\ (a1, a2, a3, a4, a5, a6, a7)\ +\ M\ (b1, b2, b3, b4, b5, b6, b7)\ =\\ &\qquad M\ (a1 + b1, a2 + b2, a3 + b3, a4 + b4, a5 + b5, a6 + b6, a7 + b7)\end{aligned}$$

$$\begin{aligned}&\textbf{instance}\ Eq\ Marcador\ \textbf{where}\\ &M\ (a1, a2, a3, a4, a5, a6, a7) == M\ (b1, b2, b3, b4, b5, b6, b7)\ =\\ &\qquad (a1== b1)\ \&\&\ (a2== b2)\ \&\&\ (a3== b3)\ \&\&\ (a4== b4)\ \&\&\\ &\qquad (a5== b5)\ \&\&\ (a6== b6)\ \&\&\ (a7== b7)\end{aligned}$$

De esta forma dos colores (simples o compuestos) serán iguales si lo son sus marcadores asociados:

$$\begin{aligned}&\textbf{instance}\ Eq\ Color\ \textbf{where}\\ &\ c1 == c2\ =\ (creaMarcador\ c1) == (creaMarcador\ c2)\end{aligned}$$

donde la función $creaMarcador$ que a cada color le asocia un marcador puede definirse como:

$$\begin{aligned}&creaMarcador\ ::\ Color\ \rightarrow\ Marcador\\ &creaMarcador\ Violeta\ &&=\ M\ (1,0,0,0,0,0,0)\\ &creaMarcador\ Añil\ &&=\ M\ (0,1,0,0,0,0,0)\\ &creaMarcador\ Azul\ &&=\ M\ (0,0,1,0,0,0,0)\\ &creaMarcador\ Verde\ &&=\ M\ (0,0,0,1,0,0,0)\\ &creaMarcador\ Amarillo\ &&=\ M\ (0,0,0,0,1,0,0)\\ &creaMarcador\ Naranja\ &&=\ M\ (0,0,0,0,0,1,0)\\ &creaMarcador\ Rojo\ &&=\ M\ (0,0,0,0,0,0,1)\\ &creaMarcador\ (Mezcla\ c1\ c2)\ &&=\ creaMarcador\ c1\ +\ creaMarcador\ c2\end{aligned}$$

**Solución al Ejercicio 5.8** (pág. 130).– Una mochila puede representarse mediante una lista de pares donde cada par consta del elemento en cuestión junto con el número de veces que dicho elemento está en la mochila, con lo cual

$$\textbf{data}\ Mochila\ a\ =\ M\ [(a,\ Int)]$$

Para definir $añadir$ hay que suponer que $a$ es instancia de las clases $Eq$ y $Ord$:

```
añadir :: (Eq a, Ord a) ⇒ a → Mochila a → Mochila a
añadir x (M ps) = M (añadir' x ps)
añadir' e []                    = [(e,1)]
añadir' e ((f,n) : ps) | e == f = (f, n+1) : ps
                       | e < f  = (e,1)      : (f,n) : ps
                       | e > f  = (f,n)      : añadir' e ps
```

Las funciones *extraer*, *esVacía* y *unión* son fáciles de definir:

```
extraer :: Mochila a → (a, Mochila a)
extraer (M ps) = extraer' ps
extraer' []                 = error "Mochila vacía"
extraer' ((e,1) : ps)       = (e, M ps)
extraer' ((e,n+1) : ps) = (e, M ((e,n) : ps))

esVacía :: Mochila a → Bool
esVacía (M []) = True
esVacía _      = False

unión :: Mochila a → Mochila a → Mochila a
unión (M [])                m = m
unión (M ((e,1) : ps))      m = unión (M ps) (añadir e m)
unión (M ((e,n+1) : ps)) m = unión (M ((e,n) : ps)) (añadir e m)
```

Una función para crear una mochila vacía es

```
crearMochila :: Mochila a
crearMochila = M []
```

Finalmente, como la igualdad entre mochilas que se desea es la estructural, dicha igualdad se puede obtener directamente por ser una mochila una lista de pares cuyas componentes ya disponen de la igualdad

```
data (Ord a) ⇒ Mochila a = M [(a, Int)] deriving Eq
```

**Solución al Ejercicio 5.9** (pág. 130).–

```
data Color = Azul | Amarillo | Añil | ... | Mezcla Color Color
             deriving (Eq, Ord, Show)

aMochila (Mezcla x y) m = aMochila x (aMochila y m)
aMochila c            m = añadir c m

instance Eq Color where
  c1 == == c2 = (aMochila c1 v == aMochila c2 v)
  where v = crearMochila
```

## 21.4. PROGRAMACIÓN CON LISTAS

**Solución al Ejercicio 6.28** (pág. 156).– Tanto si se usa $(>)$ como si se usa $(\geq)$ la lista que se obtiene estará ordenada de forma decreciente: la diferencia entre ambas se produce a la hora de insertar un elemento que está repetido, pues con $(>)$ lo insertaría al final de todas sus repeticiones (más ineficiente) mientras que con $(\geq)$ lo insertaría al principio de las mismas:

$$\begin{array}{l}\text{MAIN}> \ insertar\ 3\ [3,3,3,3,3,2,1] \text{ -- con } (>)\\ [3,3,3,3,3,3,2,1]\\ \qquad\quad \uparrow\end{array}$$

$$\begin{array}{l}\text{MAIN}> \ insertar\ 3\ [3,3,3,3,3,2,1] \text{ -- con } (\geq)\\ [3,3,3,3,3,3,2,1]\\ \uparrow\end{array}$$

Usando $(\neq)$ la lista que se obtiene **no** verifica que todos los elementos de la lista sean distintos, ni siquiera que todo elemento tenga a su lado un elemento distinto a él: lo único que hace es insertar el elemento delante del primero que se encuentre (de izquierda a derecha) distinto suyo o bien al final de todas sus repeticiones si la lista empieza por un elemento igual al elemento a insertar:

$$\begin{array}{l}\text{MAIN}> \ insertar\ 3\ [3,3,3,3,5,2,1,3,3] \text{ -- con } (\neq)\\ [3,3,3,3,3,5,2,1,3,3]\\ \qquad\quad \uparrow\end{array}$$

$$\begin{array}{l}\text{MAIN}> \ insertar\ 3\ [5,2,1,3,3] \text{ -- con } (\neq)\\ [3,5,2,1,3,3]\\ \uparrow\end{array}$$

**Solución al Ejercicio 6.29** (pág. 156).– Se considera el predicado

$$x \ << \ ys \quad \equiv \quad \langle x \text{ es menor o igual que todos los de la lista } ys\rangle$$

definido mediante:

$$\begin{array}{lcl} x \ << \ [\,] & = & True \\ x \ << \ (y : ys) & = & (x \leq y) \ \&\& \ (x \ << \ ys) \end{array}$$

Entonces, siendo $ord \equiv \ ordenadaAsc$, $ins \equiv \ insertar$, se verifica

$$\begin{array}{lll} (1) & ord\ (x : xs) & = \ ord\ xs \ \wedge \ x \ << \ xs \\ (2) & x << \ ys \ \wedge \ x \leq y & \Rightarrow x \ << \ (ins\ y\ ys) \end{array}$$

Veamos $(1)$ por inducción sobre $xs$:

— *Caso Base*: ($xs \equiv [\,]$) (trivial)

— *Paso Inductivo*: Supongamos

$$(HI) \qquad ord\ (x : xs) \qquad = \qquad ord\ xs\ \wedge\ x\ <<\ xs$$

y vamos a demostrar

$$\begin{array}{ll}
& ord\ (x : y : ys)\ =\ ord\ (y : ys)\ \wedge\ x\ <<\ (y : ys) \\
\equiv & \{\ 3)ord\ \} \\
& x \leq y\ \wedge\ ord\ (y : ys)\ =\ ord\ (y : ys)\ \wedge\ x\ <<\ (y : ys) \\
\equiv & \{\ \text{cálculo proposicional}\ \} \\
& x \leq y\ \wedge\ ord\ (y : ys)\ \Leftarrow\ ord\ (y : ys)\ \wedge\ x\ <<\ (y : ys) \qquad (trivial) \\
\wedge & \\
& x \leq y\ \wedge\ ord\ (y : ys)\ \Rightarrow\ ord\ (y : ys)\ \wedge\ x\ <<\ (y : ys) \\
\equiv & \{\ \text{cálculo proposicional}\ \} \\
& x \leq y\ \wedge\ ord\ (y : ys)\ =\ ord\ (y : ys) \qquad (trivial) \\
\wedge & \\
& x \leq y\ \wedge\ ord\ (y : ys)\ \Rightarrow\ x\ <<\ (y : ys) \\
\equiv & \{\ \text{hipótesis de inducción, definición de}\ (<<)\ \} \\
& x \leq y\ \wedge\ ord\ ys\ \wedge\ y\ <<\ ys\ \Rightarrow\ x \leq y\ \wedge\ x\ <<\ ys \\
\Leftarrow & \\
& x \leq y\ \wedge\ y\ <<\ ys\ \Rightarrow\ x\ <<\ ys
\end{array}$$

donde la certeza de la última proposición (que es una especie de relación de transitividad) se demuestra por inducción estructural sobre $ys$:

- ✓ *Caso Base*: ($ys \equiv [\,]$) (trivial)
- ✓ *Paso Inductivo*: ($ys \equiv\ z : zs$)

$$\begin{array}{ll}
& x \leq y\ \wedge\ y\ <<\ z : zs\ \Rightarrow\ x\ <<\ z : zs \\
\equiv & \{\ 2)(<<)\ \text{dos veces}\ \} \\
& x \leq y\ \wedge\ y \leq z\ \wedge\ y\ <<\ zs\ \Rightarrow\ x \leq z\ \wedge\ x\ <<\ zs \\
\Leftarrow & \{\ \text{hipótesis de inducción}\ \} \\
& x\ <<\ zs\ \wedge\ x \leq y\ \wedge\ y \leq z\ \Rightarrow\ x \leq z\ \wedge\ x\ <<\ zs \\
\equiv & \{\ \text{transitividad de}\ (\leq)\ \} \\
& Cierto
\end{array}$$

Veamos ahora $(2)$ por inducción estructural sobre $ys$:

$$x\ <<\ ys\ \wedge\ x \leq y \qquad \Rightarrow \qquad x\ <<\ (ins\ y\ ys)$$

— *Caso Base*: ($ys \equiv [\,]$) (trivial)

— *Paso Inductivo*: ($ys \equiv z : zs$)

$$
\begin{array}{ll}
& x \ << \ (z : zs) \ \wedge \ x \leq y \ \Rightarrow \ x \ << \ (ins \ y \ (z : zs)) \\
\equiv & \{ \text{cálculo proposicional} \} \\
& x \ << \ (z : zs) \ \wedge \ x \leq y \ \wedge \ \underline{y \leq z} \ \Rightarrow \ x \ << \ (ins \ y \ (z : zs)) \ \wedge \ \underline{y \leq z} \\
\wedge & \\
& x \ << \ (z : zs) \ \wedge \ x \leq y \ \wedge \ \underline{y > z} \ \Rightarrow \ x \ << \ (ins \ y \ (z : zs)) \ \wedge \ \underline{y > z} \\
\equiv & \{ \text{cálculo proposicional, 2)} ins \} \\
& x \ << \ (z : zs) \ \wedge \ x \leq y \ \wedge \ y \leq z \ \Rightarrow \ x \ << \ (y : z : zs) \\
\wedge & \\
& x \ << \ (z : zs) \ \wedge \ x \leq y \ \wedge \ y > z \ \Rightarrow \ x \ << \ (z : (ins \ y \ zs)) \\
\equiv & \{ 2)(<<) \} \\
& x \ << \ (z : zs) \ \wedge \ x \leq y \ \wedge \ y \leq z \ \Rightarrow \ x \leq y \ \wedge \ x \ << \ (z : zs) \qquad (trivial) \\
\wedge & \\
& x \ << \ (z : zs) \ \wedge \ x \leq y \ \wedge \ y > z \ \Rightarrow \ x \ << \ (z : (ins \ y \ zs)) \\
\equiv & \{ 2)(<<) \} \\
& x \ << \ (z : zs) \ \wedge \ x \leq y \ \wedge \ y > z \ \Rightarrow \ x \leq z \ \wedge \ x \ << \ (ins \ y \ zs) \\
\Leftarrow & \\
& \textit{HI}
\end{array}
$$

Una vez probados $(1)$ y $(2)$, ahora es fácil demostrar por inducción sobre $zs$:

$$
\begin{array}{l}
\forall \, x, zs \ . \ zs \ :: \ [a] \ . \ x \ :: \ a \ . \\
\qquad ord \ zs \qquad \Rightarrow \qquad ord \ (insertar \ x \ zs)
\end{array}
$$

— *Caso Base*: ($zs \equiv [\,]$) (trivial)

— *Paso Inductivo*:

$$ord \ (z : zs) \ \Rightarrow \ ord \ (ins \ x \ (z : zs))$$

$$
\begin{array}{ll}
\equiv & \{ \text{cálculo proposicional} \} \\
& ord \ (z : zs) \ \wedge \ z \leq x \ \Rightarrow \ ord \ (ins \ x \ (z : zs)) \ \wedge \ z \leq x \\
\wedge & \\
& ord \ (z : zs) \ \wedge \ z > x \ \Rightarrow \ ord \ (ins \ x \ (z : zs)) \ \wedge \ z > x \\
\equiv & \{ \text{definición de } ins \} \\
& ord \ (z : zs) \ \wedge \ z \leq x \ \Rightarrow \ ord \ (z : ins \ x \ zs) \ \wedge \ z \leq x \\
\wedge & \\
& ord \ (z : zs) \ \wedge \ z > x \ \Rightarrow \ ord \ (x : z : zs) \ \wedge \ z > x \\
\equiv & \{ (1) \} \\
& ord \ (z : zs) \ \wedge \ z \leq x \ \Rightarrow \ ord \ (ins \ x \ zs) \ \wedge \ z \ << \ ins \ x \ zs \ \wedge \ z \leq x \\
\wedge &
\end{array}
$$

$$
\begin{array}{ll}
 & ord\ (z : zs)\ \wedge\ z\ >\ x\ \Rightarrow\ x \leq z\ \wedge\ ord\ (z : zs)\ \wedge\ z\ >\ x \qquad (trivial) \\
\Leftarrow & \{\ (1) \text{ a la izquierda de la primera, hipótesis de inducción }\} \\
 & ord\ zs\ \wedge\ z\ <<\ zs\ \wedge\ z \leq x\ \Rightarrow\ z\ <<\ ins\ x\ zs\ \wedge\ z \leq x \\
\Leftarrow & \{\ (2)\ \} \\
 & ord\ zs\ \wedge\ z\ <<\ (ins\ x\ zs)\ \wedge\ z \leq x\ \Rightarrow\ z\ <<\ ins\ x\ zs\ \wedge\ z \leq x \\
\equiv & \{ \text{cálculo proposicional} \} \\
 & Cierto
\end{array}
$$

**Solución al Ejercicio 6.33** (pág. 164).– Aunque en principio nos puede tentar la utilización de $(+\!\!+)$ en

$$
\begin{array}{l}
repBin\ 0\ =\ [\text{'0'}] \\
repBin\ 1\ =\ [\text{'1'}] \\
repBin\ n\ =\ repBin\ (n\ `div`\ 2)\ +\!\!+\ repBin\ (n\ `mod`\ 2)
\end{array}
$$

es más eficiente seguir con la misma filosofía que en el Ejercicio 2.14, pero devolviendo una lista de caracteres en vez de una lista de dígitos:

$$
\begin{array}{l}
repBin\ 0\ \ \ \ \ \ =\ [\text{'0'}] \\
repBin\ m\ \ \ \ \ \ =\ repBin'\ m\ [\,] \\
repBin'\ m\ xs\ = \\
\quad \textbf{if}\ m == 0\ \textbf{then} \\
\quad\quad xs \\
\quad \textbf{else} \\
\quad\quad repBin'\ (m\ `div`\ 2)\ (chr\ (ord\ \text{'0'} + (m\ `mod`\ 2))\ :\ xs)
\end{array}
$$

**Solución al Ejercicio 6.34** (pág. 164).–

1. Demostremos

$$
\begin{array}{l}
\forall\ xs\ \centerdot\ xs\ ::\ [a]\ \centerdot\ \forall\ ys\ \centerdot\ ys\ ::\ [a]\ \centerdot \\
\qquad inv2\ xs\ ys\ \ \ =\ \ \ inv2\ xs\ [\,] +\!\!+\ ys
\end{array}
$$

por inducción estructural sobre $xs$.

— *Caso Base*: ($xs \equiv [\,]$)

$$
\begin{array}{ll}
 & inv2\ [\,]\ ys\ =\ inv2\ [\,]\ [\,] +\!\!+\ ys \\
\equiv & \{\ 1) inv2 \text{ dos veces }\} \\
 & ys\ =\ [\,] +\!\!+\ ys \\
\equiv & \{\ 1) +\!\!+\ \} \\
 & Cierto
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & inv2\ (x : xs)\ ys\ =\ inv2\ (x : xs)\ [\,] +\!\!+\ ys \\
\equiv & \{\ 1)inv2 \text{ dos veces}\ \} \\
 & inv2\ xs\ (x : ys)\ =\ inv2\ xs\ [x] +\!\!+\ ys \\
\equiv & \{\ \text{hipótesis de inducción}\ \} \\
 & inv2\ xs\ [\,] +\!\!+\ (x : ys)\ =\ (inv2\ xs\ [\,] +\!\!+\ [x]) +\!\!+\ ys \\
\Leftarrow & \{\ \text{asociativiad de} +\!\!+ \text{ y } (is)\ \} \\
 & x : ys\ =\ [x] +\!\!+\ ys \\
\equiv & \{\ \text{definición de} +\!\!+\ \} \\
 & Cierto
\end{array}
$$

y ahora, con la definición de $inv$ en términos de $inv2$

$$inv\ xs\ =\ inv2\ xs\ [\,]$$

tenemos que:

$$
\begin{array}{ll}
 & inv\ (x : xs) \\
\equiv & \{\ \text{definición de } inv\ \} \\
 & inv2\ (x : xs)\ [\,] \\
\equiv & \{\ 2)inv2\ \} \\
 & inv2\ xs\ [x] \\
\equiv & \{\ (*)\ \} \\
 & inv2\ xs\ [\,] +\!\!+\ [x] \\
\equiv & \{\ \text{hipótesis}\ \} \\
 & inv\ xs +\!\!+\ [x]
\end{array}
$$

con lo cual $inv$ definida en términos de $inv2$ coincide con

$$
\begin{array}{lll}
inv\ [\,] & = & [\,] \\
inv\ (x : xs) & = & inv\ xs +\!\!+\ [x]
\end{array}
$$

2. Ahora se va a demostrar que:

$$inv\ (u +\!\!+\ (x : v)) \qquad = \qquad inv\ v +\!\!+\ (x : inv\ u)$$

por inducción sobre $u$:

— *Caso Base*: ($u \equiv [\,]$)

$$
\begin{array}{ll}
 & inv\ ([\,] +\!\!+\ (x : v)) \\
\equiv & \{\ 1)+\!\!+\ \} \\
 & inv\ (x : v)
\end{array}
$$

$\equiv$ { 2)$inv$ o $(**)$ }
$inv\ v +\!\!+ [x]$
$\equiv$ { definición de $:$ }
$inv\ v +\!\!+ (x : [\,])$
$\equiv$ { 1)$inv$ }
$inv\ v +\!\!+ (x : inv\ [\,])$

— *Paso Inductivo*: Suponemos

$$inv\ (us +\!\!+ (x : v)) \quad = \quad inv\ v +\!\!+ (x : inv\ us)$$

y hay que demostrar

$$inv\ ((u : us) +\!\!+ (x : v)) \quad = \quad inv\ v +\!\!+ (x : inv\ (u : us))$$

con lo cual

$inv\ ((u : us) +\!\!+ (x : v))$
$\equiv$ { 2)$+\!\!+$ }
$inv\ (u : (us +\!\!+ (x : v)))$
$\equiv$ { 2)$inv$ o $(**)$ }
$inv\ (us +\!\!+ (x : v)) +\!\!+ [u]$
$\equiv$ { hipótesis de inducción }
$(inv\ v +\!\!+ (x : inv\ us)) +\!\!+ [u]$
$\equiv$ { asociatividad de $+\!\!+$ }
$inv\ v +\!\!+ ((x : inv\ us) +\!\!+ [u])$
$\equiv$ { 2)$+\!\!+$ }
$inv\ v +\!\!+ (x : (inv\ us +\!\!+ [u]))$
$\equiv$ { 2)$inv$ o $(**)$ }
$inv\ v +\!\!+ (x : inv\ (u : us))$

**Solución al Ejercicio 6.35** (pág. 164).–

$$esCapicua :: String \to Bool$$
$$esCapicua\ xs = (xs == inv2\ xs\ [\,])$$

**Solución al Ejercicio 6.36** (pág. 165).– Sea la función

$$1 \quad `de`\ (x : \_) = x$$
$$(n+1)\ `de`\ (\_ : xs) = n\ `de`\ xs$$

Probaremos que

$$\forall\, m \,.\, m \geq 1 \,.\, \forall\, n \,.\, 1 \leq n \leq m \,.\, \forall\, x_i \,.$$

$$n \text{ `}de\text{` } [x_1,\ \ldots,\ x_m] \quad = \quad x_n$$

Para $m = 1$ es evidente por la ecuación 1)$de$; supuesto cierto para $m$, consideremos $1 \leq n \leq m+1$; si $n = 1$ entonces es trivial; para $1 < n \leq m+1$, tenemos:

$$\begin{array}{ll} & n \text{ `}de\text{` } [x_1,\ \ldots,\ x_{m+1}] \\ \equiv & \{\, 2)de \,\} \\ & (n-1) \text{ `}de\text{` } [x_2,\ \ldots,\ x_{m+1}] \\ \equiv & \{\, \text{hipótesis de inducción } (0 < n-1 \leq m) \,\} \\ & x_n \end{array}$$

Otra forma es probando por inducción sobre $ys$ que

$$\begin{array}{l} \forall\, ys \,\centerdot\, ys :: [a] \,\centerdot\, \forall\, xs, x \,\centerdot\, xs :: [a],\ x :: a \,\centerdot \\ \qquad (\#ys + 1) \text{ `}de\text{` } (ys +\!\!+ (x : xs)) \quad = \quad x \end{array}$$

donde $\#$ se usa para abreviar la función $long$ que devuelve la longitud de la lista argumento:

$$\begin{array}{ll} long\ [\,] & = 0 \\ long\ (\_ : xs) & = 1 + long\ xs \end{array}$$

— *Caso Base*: ($ys \equiv [\,]$)

$$\begin{array}{ll} & (\#[\,] + 1) \text{ `}de\text{` } [\,] +\!\!+ (x : xs) = x \\ \equiv & \{\, 1)\#,\ 1) +\!\!+ \,\} \\ & 1 \text{ `}de\text{` } (x : xs) = x \\ \equiv & \{\, 1)de \,\} \\ & Cierto \end{array}$$

— *Paso Inductivo*:

$$\begin{array}{ll} & (\#(y : ys) + 1) \text{ `}de\text{` } (y : ys) +\!\!+ (x : xs) = x \\ \equiv & \{\, 2)\#, \text{ asoc. y conm. de } +,\ 2) +\!\!+ \,\} \\ & (\#ys + 2) \text{ `}de\text{` } (y : (ys +\!\!+ (x : xs))) = x \\ \equiv & \{\, 2)de \,\} \\ & (\#ys + 1) \text{ `}de\text{` } (ys +\!\!+ (x : xs)) = x \\ \Leftarrow & \\ & HI \end{array}$$

**Solución al Ejercicio 6.37** (pág. 165).– Vamos a probar directamente

$$\forall\, xs :: [Int] \,.\forall\, x :: Int \,.$$

$$par\ x\ \wedge\ todosPares\ xs \quad = \quad todosPares\ (x : xs)$$

$$\begin{array}{ll} & todosPares\ (x : xs) \\ \equiv & \{\ \text{definición de } todosPares\ \} \\ & test\ (\lambda\ y\ \rightarrow\ y\ `mod`\ 2 == 0)\ (x : xs) \end{array}$$

$$\begin{array}{ll} \equiv & \{\ 2)test\ \} \\ & (\lambda\ y\ \rightarrow\ y\ `mod`\ 2 == 0)\ x\ \wedge\ test\ (\lambda\ y\ \rightarrow\ y\ `mod`\ 2 == 0)\ xs \\ \equiv & \{\ \beta\text{–regla, definición de } todosPares\ \} \\ & (x\ `mod`\ 2 == 0)\ \wedge\ todosPares\ xs \end{array}$$

y de aquí, $\forall n \,.\, n \geq 0$

$$\begin{array}{ll} & todosPares\ [x_n,\ x_{n-1},\ \ldots,\ x_0] \\ \equiv & \\ & \forall\ k\ .\ 0 \leq\ k \leq\ n\ .\ par\ x_k \end{array}$$

que se probaría por inducción sobre $n$.

**Solución al Ejercicio 6.38** (pág. 165).– La demostración se realizará por inducción sobre $xs$:

— *Caso Base*:

$$\begin{array}{ll} & takeWhile\ p\ [\,]\ +\!\!+\ dropWhile\ p\ [\,] \\ \equiv & \{\ 1)takeWhile,\ 1)dropWhile\ \} \\ & [\,]\ +\!\!+\ [\,] \\ \equiv & \{\ 1)+\!\!+\ \} \\ & [\,] \end{array}$$

— *Paso Inductivo*:

$$\begin{array}{ll} & takeWhile\ p\ (x : xs)\ +\!\!+\ dropWhile\ p\ (x : xs) \\ \equiv & \{\ 2)takeWhile,\ 2)\ dropWhile\ \} \\ & (x : takeWhile\ p\ xs)\ +\!\!+\ dropWhile\ p\ xs, \text{ si } p\ x \\ & [\,]\ +\!\!+\ (x : xs), \text{ en otro caso} \\ \equiv & \{\ 2)+\!\!+,1)+\!\!+\ \} \\ & x : (takeWhile\ p\ xs\ +\!\!+\ dropWhile\ p\ xs), \text{ si } p\ x \\ & x : xs, \text{ en otro caso} \\ \equiv & \{\ \text{hipótesis de inducción}\ \} \\ & x : xs, \text{ si } p\ x \\ & x : xs, \text{ en otro caso} \\ \equiv & \{\ \text{cálculo de predicados}\ \} \\ & x : xs \end{array}$$

**Solución al Ejercicio 6.39** (pág. 165).– Se considera la función

$$\begin{array}{l} \mathit{último\,YResto}\ [x] \quad = (x,\ [\,]) \\ \mathit{último\,YResto}\ (x : xs) \ = (y,\ x : rs) \\ \quad \mathbf{where}\ (y,\ rs) \ = \mathit{último\,YResto}\ xs \end{array}$$

y vamos a abreviar *último YResto* mediante $uR$. Se demuestra que

$$(*) \qquad \begin{array}{l} \forall\, xs \,\centerdot\, xs :: [a],\ xs \neq [\,] \,\centerdot \\ \qquad xs \ = \ r +\!\!+ [x]\ \mathbf{where}\ (x, r) \ = \ uR\ xs \end{array}$$

— *Caso Base*: ($xs \equiv [y]$) (trivial)

— *Paso Inductivo*: ($xs \equiv y : ys$)

$$\begin{array}{ll} & y : ys \ = \ r +\!\!+ [x]\ \mathbf{where}\ (x, r) \ = \ uR\ (y : ys) \\ \equiv & \\ & y : ys \ = \ r +\!\!+ [x]\ \mathbf{where}\ (x, r) = \ (z, y : v)\ \mathbf{where}\ (z, v) = uR\ ys \\ \equiv & \\ & y : ys \ = \ (y : v) +\!\!+ [x]\ \mathbf{where}\ (x, v) = uR\ ys \\ \Leftarrow & \ \{\ \text{igualdad de listas, asociatividad de } (+\!\!+)\ \} \\ & ys \ = \ v +\!\!+ [x]\ \mathbf{where}\ (x, v) = uR\ ys \\ \equiv & \\ & \mathit{HI} \end{array}$$

Con ello se puede escribir

$$\begin{array}{l} \mathit{circulaD}\ [\,] \ = [\,] \\ \mathit{circulaD}\ xs \ = \ x : r\ \mathbf{where}\ (x, r) \ = \ \mathit{último\,YResto}\ xs \end{array}$$

Para $circulaI$ tenemos:

$$\begin{array}{lll} \mathit{circulaI}\ [\,] & = & [\,] \\ \mathit{circulaI}\ (x : xs) & = & \mathit{ciAux}\ xs\ x \end{array}$$

$$\begin{array}{lll} \mathit{ciAux}\ [\,]\ \ x & = & [x] \\ \mathit{ciAux}\ (y : ys)\ \ x & = & y : \mathit{ciAux}\ ys\ x \end{array}$$

y se prueba de forma muy fácil (por inducción sobre $r$):

$$(**) \qquad \mathit{ciAux}\ r\ x \qquad = \qquad r +\!\!+ [x]$$

con lo cual

$$\mathit{circulaI}\ (x : xs) \qquad = \qquad xs +\!\!+ [x]$$

Si abreviamos con $cI$ y $cD$ los nombres de las funciones, también podemos demostrar (sin utilizar el principio de inducción) que

$$\begin{array}{lll} (a) & cI \,.\, cD & = \quad id \\ (b) & cD \,.\, cI & = \quad id \end{array}$$

En efecto, veamos $(a)$:

$$cI\ (cD\ xs) \qquad = \qquad xs$$

Si $xs \equiv [\,]$ es trivial; si $xs \not\equiv [\,]$, entonces

$$\begin{array}{ll} & cI\ (cD\ xs) \\ \equiv & \{\, 2)cI \,\} \\ & cI\ (y : r)\ \mathbf{where}\ (y, r)\ =\ uR\ xs \\ \equiv & \{\, (**) \,\} \\ & r +\!\!+ [y]\ \mathbf{where}\ (y, r)\ =\ uR\ xs \\ \equiv & \{\, (*) \,\} \\ & xs \end{array}$$

Finalmente, veamos $(b)$:

$$cD\ (cI\ xs) \qquad = \qquad xs$$

Para $xs \equiv [\,]$ es trivial; si $xs \not\equiv [\,]$, entonces

$$\begin{array}{ll} & cD\ (cI\ (x : xs)) \\ \equiv & \{\, \textbf{(**)} \,\} \\ & cD\ (xs +\!\!+ [x]) \\ \equiv & \{\, 2)cD \,\} \\ & x' : r'\ \mathbf{where}\ (x', r')\ =\ uR\ (xs +\!\!+ [x]) \\ \equiv & \{\, (*) \,\} \\ & x : xs \end{array}$$

**Solución al Ejercicio 6.40** (pág. 165).–

$$\begin{array}{lll} [\,] & `esMenorQue` \_ & = \ True \\ \_ & `esMenorQue` [\,] & = \ False \\ (x : xs)\ `esMenorQue` \ (y : ys) & & = \ x < y \ \|\ x == y \ \&\&\ xs\ `esMenorQue`\ ys \end{array}$$

Obsérvese que la tercera ecuación coincide *exactamente* con la definición de orden lexicográfico.

**Solución al Ejercicio 6.41** (pág. 165).–

$$\begin{array}{lll} [\,] & `esMenorQue` \_ & = \ True \\ \_ & `esMenorQue` [\,] & = \ False \\ (x : xs)\ `esMenorQue` \ (y : ys) & & = \ x' < y' \ \|\ x' == y' \ \&\&\ xs\ `esMenorQue`\ ys \\ \qquad \mathbf{where}\ (x', y') & & = \ (toUpper\ x, toUpper\ y) \end{array}$$

**Solución al Ejercicio 6.42** (pág. 165).–

$$\begin{array}{lll} carácter\ \_ & [\,] & = error\ \text{"Fuera de límites"} \\ carácter\ 0 & (x : \_) & = x \\ carácter\ (n+1)\ (x : xs) & & = carácter\ n\ xs \end{array}$$

**Solución al Ejercicio 6.43** (pág. 165).–

$$\begin{array}{lll} pos\ \_\ [\,] & = error\ \text{"No está en la cadena"} \\ pos\ x\ (y : ys) & = \textbf{if}\ x == y\ \textbf{then}\ 0\ \textbf{else}\ 1 + pos\ x\ ys \end{array}$$

**Solución al Ejercicio 6.44** (pág. 166).– Suponiendo que todas las listas son de la misma dimensión:

$$\begin{array}{l} trasponer\ \_\ \_\ [\,] = [\,] \\ trasponer\ xs\ ys\ (z : zs) = \\ \quad caracter\ (pos\ z\ ys)\ xs\ :\ trasponer\ xs\ ys\ zs \end{array}$$

Hay una función (*transpose*) en el módulo de biblioteca *List* para el mismo fin.

**Solución al Ejercicio 6.45** (pág. 166).–

$$\begin{array}{lll} añade\ [\,] & x & = [x] \\ añade\ (y : ys)\ x & & = y : añade\ x\ ys \end{array}$$

**Solución al Ejercicio 6.46** (pág. 166).– Considerando la siguiente definición

$$\textbf{data}\ Empleado = Emp\ [Char]\ Int\ \textbf{deriving}\ Show$$

y suponiendo que la lista de caracteres no es vacía, se tiene:

$$\begin{array}{l} másJoven\ [x] = x \\ másJoven\ (empx@(Emp\ nx\ ex) : empy@(Emp\ ny\ ey) : zs) = \\ \quad másJoven\ ((\textbf{if}\ ex < ey\ \textbf{then}\ empx\ \textbf{else}\ empy) : zs) \end{array}$$

**Solución al Ejercicio 6.47** (pág. 166).– Una primera solución es

$$\begin{array}{l} joven\ empx@(Emp\ nx\ ex)\ empy@(Emp\ ny\ ey) = \\ \quad \textbf{if}\ ex < ey\ \textbf{then}\ empx\ \textbf{else}\ empy \end{array}$$

$$\begin{array}{lll} másJoven'\ [x] & & = x \\ másJoven'\ (x : ys) & & = joven\ x\ (másJoven'\ ys) \end{array}$$

aunque también se pueden definir las instancias:

$$\begin{array}{l}\textbf{instance}\ Eq\ Empleado\ \textbf{where}\\ \quad Emp\ \_\ x == Emp\ \_\ y\ =\ x == y\end{array}$$

$$\begin{array}{l}\textbf{instance}\ Ord\ Empleado\ \textbf{where}\\ \quad Emp\ \_\ x \leq Emp\ \_\ y\ =\ x \leq y\end{array}$$

entonces tenemos

$$\begin{array}{ll} joven & = min \\ másJoven' & = minimum \end{array}$$

siendo $minimum$ la función predefinida que devuelve el mínimo de una lista.

**Solución al Ejercicio 6.48** (pág. 166).– El caso base es fácil:

$$\begin{array}{ll} & p\ <|\ (f\ \#\ [\,])\ =\ f\ \#\ (\ (p.f)\ <|\ [\,]\ ) \\ \equiv & \{\ 1)map,\ 1)filter\ \} \\ & p\ <|\ [\,]\ =\ f\ \#\ [\,] \\ \equiv & \{\ 1)filter,\ 1)map\ \} \\ & Cierto \end{array}$$

y el paso inductivo queda

$$\begin{array}{ll} & p\ <|\ (f\ \#\ (x : xs))\ =\ f\ \#\ ((p.f)\ <|\ (x : xs)) \\ \equiv & \{\ 2)map,\ 2)filter\ \} \\ & p\ <|\ (f\ x\ :\ f\ \#\ xs)\ = \\ & f\ \#\ (\textbf{if}\ \ (p.f)\ x\ \ \textbf{then}\ \ x\ :\ (p.f)\ <|\ xs\ \ \textbf{else}\ \ (p.f)\ <|\ xs) \\ \equiv & \{\ h(\textbf{if}\ \ b\ \ \textbf{then}\ \ u\ \ \textbf{else}\ \ v) = \textbf{if}\ \ b\ \ \textbf{then}\ \ hu\ \ \textbf{else}\ \ hv\ \} \\ & \textbf{if}\ \ p\ (f\ x)\ \ \textbf{then}\ \ f\ x\ :\ p\ <|\ (f\ \#\ xs)\ \ \textbf{else}\ \ p\ <|\ (f\ \#\ xs)\ = \\ & \textbf{if}\ \ (p.f)\ x\ \ \textbf{then}\ \ f\ \#\ (x\ :(p.f)\ <|\ xs)\ \ \textbf{else}\ \ f\ \#\ ((p.f) <|xs) \\ \equiv & \{\ 2)map \text{ al segundo miembro de la igualdad}\ \} \\ & \textbf{if}\ \ p\ (f\ x)\ \ \textbf{then}\ \ f\ x : p\ <|\ (f\ \#\ xs)\ \ \textbf{else}\ \ p\ <|\ (f\ \#\ xs)\ = \\ & \textbf{if}\ \ (p.f)\ x\ \ \textbf{then}\ \ f\ x : f\ \#\ ((p.f) <|xs)\ \ \textbf{else}\ \ f\ \#\ ((p.f) <|xs) \\ \Leftarrow & \\ & HI \end{array}$$

**Solución al Ejercicio 6.49** (pág. 166).– Hay que probar

$$\begin{array}{l} \forall\ xs\ \centerdot\ xs\ ::\ [a]\ \centerdot\ (\ \forall\ f\ \centerdot\ f\ ::\ a \rightarrow\ b\ \centerdot \\ \qquad f\ (head\ xs)\ =\ head\ (f\ \#\ xs)\ ) \end{array}$$

Para el caso base ($xs \equiv [\,]$) hay que suponer que $f$ es estricta (en otro caso la igualdad sólo se daría para listas no vacías). El resto de los casos es:

$$\begin{array}{ll} & f\ (head\ (x : xs))\ =\ head\ (f\ \#\ (x : xs)) \\ \equiv & \{\ 1)head,\ 2)map\ \} \\ & f\ x\ =\ head\ (f\ x\ :\ f\ \#\ xs) \end{array}$$

$\equiv$ { 1)*head* }
*Cierto*

**Solución al Ejercicio 6.50** (pág. 166).– Hay que probar

$$\forall\, xs \,\centerdot\, xs :: [a] \,\centerdot\, \forall f, g \,\centerdot\, f :: b \to c,\ g :: a \to b \centerdot \\ (f\,\#)\,(g\,\#\,xs) \;=\; (f.g)\,\#\,xs$$

— *Caso Base*:

$f\,\#\,(g\,\#\,[\,]) \;=\; (f.g)\,\#\,[\,]$
$\equiv$ { 1)# dos veces }
$f\,\#\,[\,] \;=\; [\,]$
$\equiv$ { 1)# }
*Cierto*

— *Paso Inductivo*:

$f\,\#\,(g\,\#\,(x : xs)) \;=\; (f.g)\,\#\,(x : xs)$
$\equiv$ { 2)# dos veces }
$f\,\#\,(g\;x \;:\; g\,\#\,xs) \;=\; (f.g)\;x \;:\; (f.g)\,\#\,xs$
$\equiv$ { 2)#, definición de composición }
$f\;(g\;x) \;:\; f\,\#\,(g\,\#\,xs) \;=\; f\;(g\;x) \;:\; (f.g)\,\#\,xs$
$\Leftarrow$
*HI*

**Solución al Ejercicio 6.51** (pág. 166).– Para $(a)$, como $f\,\#\,\equiv\; map\,f$, lo primero que hay que probar es

$$map\,f\;(concat\;xs) \quad = \quad concat\;(map\;(map\,f)\;xs)$$

para lo cual procederemos por inducción sobre $xs$:

— *Caso Base*: ($xs \equiv [\,]$)

$map\,f\;(concat\,[\,]) \;=\; concat\;(map\;(map\,f)\,[\,])$
$\equiv$ { 1)*concat*, 1)*map* }
$map\,f\,[\,] \;=\; concat\,[\,]$
$\equiv$ { 1)*map*, 1)*concat* }
*Cierto*

— *Paso Inductivo*: ($xs \equiv x : u$) Hay que probar que

$$map\ f\ (concat\ (x : u)) \quad = \quad concat\ (map\ (map\ f)\ (x : u))$$

sabiendo que

$$map\ f\ (concat\ u) \quad = \quad concat\ (map\ (map\ f)\ u)$$

$$map\ f\ (concat\ (x : u))$$

$\equiv$ { definición de $concat$ }
$map\ f\ (x +\!\!+ concat\ u)$
$\equiv$ { $(*)$ (se demuestra después) }
$(map\ f\ x) +\!\!+ map\ f\ (concat\ u)$
$\equiv$ { hipótesis de inducción }
$(map\ f\ x) +\!\!+ concat\ (map\ (map\ f)\ u)$
$\equiv$ { 2)$concat$ }
$concat\ ((map\ f\ x)\ :\ map\ (map\ f)\ u)$
$\equiv$ { 2)$map$ }
$concat\ (map\ (map\ f)\ (x : u))$

y queda probar, por inducción sobre $u$,

$$(*) \qquad map\ f\ (u +\!\!+ v)\ =\ map\ f\ u +\!\!+ map\ f\ v$$

El caso base ($u \equiv [\,]$) es trivial, y el paso inductivo

$map\ f\ ((x : u) +\!\!+ v)\ =\ map\ f\ (x : u) +\!\!+ map\ f\ v$
$\equiv$ { 2)$+\!\!+$, 2)$map$ }
$map\ f\ (x : (u +\!\!+ v))\ =\ (f\ x\ :\ map\ f\ u) +\!\!+ map\ f\ v$
$\equiv$ { 2)$map$, 2)$+\!\!+$ }
$f\ x\ :\ map\ f\ (u +\!\!+ v)\ =\ f\ x\ :\ (map\ f\ u +\!\!+ map\ f\ v)$
$\Leftarrow$
*HI*

En la demostración de $(b)$:

$$\forall\ xs\ \centerdot\ xs\ ::\ [a],\ xs \neq [\,]\ \centerdot\ inc\#\ (tail\ xs)\ =\ tail\ (inc\ \#\ xs)$$

se exige que la lista $xs$ no sea vacía, dado que $tail$ sólo está definido para listas no vacías; la demostración se hace directamente:

$$
\begin{array}{ll}
 & inc\ \#\ (tail\ (x : xs)) \ =\ tail\ (inc\ \#\ (x : xs)) \\
\equiv & \{\ 1)tail,\ 2)map\ \} \\
 & inc\ \#\ xs \ =\ tail\ (inc\ x \ :\ inc\ \#\ xs) \\
\equiv & \{\ 1)tail\ \} \\
 & Cierto
\end{array}
$$

Obsérvese que en la demostración no se ha utilizado explícitamente ninguna información sobre la función $inc$ (implícitamente sólo que su tipo sea $a \ \rightarrow\ a$).

**Solución al Ejercicio 6.52** (pág. 166).– Siendo $g$ la constante que representa al conjunto de datos, la $conmutativa$ sería:

$$commutativa \ =\ and\ [\ a *:\ b == b *:\ a \mid a \ \leftarrow\ g,\ b \ \leftarrow\ g\ ]$$

Obsérvese que una evaluación perezosa de la igualdad dejará de computar en cuanto se encuentren dos elementos que no conmutan. Para definir $esNeutro$:

$$
\begin{array}{l}
esNeutro\ e \ =\ and\ [\ a *:\ e == a\ \&\&\ e *:\ a == a \mid a \ \leftarrow\ g\ ] \\
\text{-- o también} \\
esNeutro\ e \ =\ [\,] == [\ 1 \mid a \leftarrow\ g,\ (a *:\ e \ \neq\ a \ \|\ e *:\ a \ \neq\ a)\ ]
\end{array}
$$

con lo cual $neutro$ queda:

$$neutro\ xs \ =\ head\ [\ e \mid e \leftarrow\ xs,\ esNeutro\ e\ ]$$

que produce un error si no existe elemento neutro. Finalmente la $asociativa$:

$$
\begin{array}{l}
asociativa \ =\ and\ [\ a *: (b *:\ c) == (a *:\ b) *:\ c \mid \\
\qquad\qquad\qquad a \ \leftarrow\ g,\ b \ \leftarrow\ g,\ c \ \leftarrow\ g\ ]
\end{array}
$$

**Solución al Ejercicio 6.53** (pág. 167).– La función $tops$ calcula la potencia de un conjunto. Una función para obtener todos los pares de subconjuntos disjuntos no vacíos de un conjunto dado puede obtenerse a partir de $tops$ y de la función $\backslash\backslash$ (diferencia de conjuntos):

$$
\begin{array}{l}
lPar\ l \ =\ [\ (sl, sl') \mid sl \ \leftarrow\ tops\ l, \qquad\quad sl \ \neq\ [\,], \\
\qquad\qquad\qquad\qquad\quad sl' \leftarrow\ tops\ (l \backslash\backslash\ sl),\ sl' \neq\ [\,]\ ]
\end{array}
$$

**Solución al Ejercicio 6.54** (pág. 167).– Lo hacemos por inducción sobre la lista argumento $w$ (siendo $length \equiv\ lon$); el caso base es trivial, y el paso inductivo queda:

$$
\begin{array}{ll}
 & lon\ (des\ (x : w)\ a) \ =\ lon\ (x : w) \\
\equiv & \{\ 2)des,\ 2)lon\ \} \\
 & lon\ (((a : x) : w) : (map\ (x :)\ (des\ w\ a))) \ =\ 1 \ +\ lon\ w \\
\equiv & \{\ 2)lon\ \} \\
 & 1 \ +\ lon\ (map\ (x :)\ (des\ w\ a)) \ =\ 1 \ +\ lon\ w \\
\equiv & \{\ (*)\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & 1 \ + \ lon\ (des\ w\ a) \ = \ 1 \ + \ lon\ w \\
\Leftarrow & \\
 & \mathit{HI}
\end{array}
$$

y queda demostrar, por inducción sobre $xs$,

$$(*) \qquad lon\ (\ map\ f\ xs\ )\ =\ lon\ xs$$

El caso base ($xs \equiv [\,]$) es trivial, mientras que el paso inductivo sería:

$$
\begin{array}{ll}
 & lon\ (map\ f\ (x : xs)) \ = \ lon\ (x : xs) \\
\equiv & \{\ 2)map,\ 2)lon\ \} \\
 & lon\ (f\ x\ :\ map\ f\ xs) \ = \ 1 \ + \ lon\ xs \\
\equiv & \{\ 2)lon\ \} \\
 & 1 \ + \ lon\ (map\ f\ xs) \ = \ 1 \ + \ lon\ xs \\
\Leftarrow & \\
 & \mathit{HI}
\end{array}
$$

**Solución al Ejercicio 6.55** (pág. 167).– Probamos por inducción sobre $n$:

$$
\begin{array}{l}
\forall n \centerdot n \ \geq \ 0 \centerdot \\
\qquad \mathit{foldr}\ f\ z\ [x_1,\ \ldots,\ x_n] \ = \ \mathit{foldl}\ (\mathit{flip} f)\ z\ [x_n,\ \ldots,\ x_1]
\end{array}
$$

— *Caso Base*:

$$
\begin{array}{ll}
 & \mathit{foldr}\ f\ z\ [\,] \ = \ \mathit{foldl}\ (\mathit{flip}\ f)\ z\ [\,] \\
\equiv & \{\ 1)\mathit{foldr},\ 1)\mathit{foldl}\ \} \\
 & z \ = \ z
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & \mathit{foldl}\ (\mathit{flip}\ f)\ z\ [x_{n+1},\ \ldots,\ x_1] \\
\equiv & \{\ 2)\mathit{foldl}\ \} \\
 & \mathit{foldl}\ (\mathit{flip}\ f)\ (f\ x_{n+1}\ z)\ [x_n,\ \ldots,\ x_1] \\
\equiv & \{\ \text{hipótesis de inducción}\ \} \\
 & \mathit{foldr}\ f\ (f\ x_{n+1}\ z)\ [x_1,\ \ldots,\ x_n] \\
\equiv & \{\ (*)\ \text{(ver después)}\ \} \\
 & \mathit{foldr}\ f\ z\ [x_1,\ \ldots,\ x_{n+1}]
\end{array}
$$

y queda demostrar:

$$
\begin{array}{ll}
 & \forall\ n \centerdot n \ \geq \ 1 \centerdot \\
(*) & \qquad \mathit{foldr}\ f\ (f\ x_n\ z)[x_1,\ \ldots,\ x_{n-1}] \ = \ \mathit{foldr}\ f\ z\ [x_1,\ \ldots,\ x_n]
\end{array}
$$

— *Caso Base*:

$$
\begin{array}{ll}
& \mathit{foldr}\ f\ (f\ x\ z)\ [\,]\ =\ \mathit{foldr}\ f\ z\ [x] \\
\equiv & \{\ 1)\mathit{foldr},\ 2)\mathit{foldr}\ \} \\
& f\ x\ z\ =\ f\ x\ (\mathit{foldr}\ f\ z\ [\,]) \\
\equiv & \{\ 1)\mathit{foldr}\ \} \\
& \mathit{Cierto}
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
& \mathit{foldr}\ f\ z\ [x_1,\ \ldots,\ x_{n+1}] \\
\equiv & \{\ 2)\mathit{foldr}\ \} \\
& f\ x_1\ (\mathit{foldr}\ f\ z\ [x_2,\ \ldots,\ x_{n+1}]) \\
\equiv & \{\ \text{hipótesis de inducción}\ \} \\
& f\ x_1\ (\mathit{foldr}\ f\ (f\ x_{n+1}\ z)\ [x_2,\ \ldots,\ x_n]) \\
\equiv & \{\ 2)\mathit{foldr}\ \} \\
& f\ (f\ x_{n+1}\ z)\ [x_1,\ \ldots,\ x_n]
\end{array}
$$

**Solución al Ejercicio 6.56** (pág. 167).– Siendo $pld \equiv \mathit{foldr}$, $pli \equiv \mathit{foldl}$; vamos a probar antes que:

$$
(1) \qquad \begin{array}{l} \forall\ ys\ \centerdot\ \forall\ x, z, f\ \centerdot \\ \qquad f\ x\ (pld\ f\ z\ ys)\ =\ pld\ f\ (f\ z\ x)\ ys \end{array}
$$

cuando $f$ sea asociativa y conmutativa, por inducción sobre $ys$:

— *Caso Base*:

$$
\begin{array}{ll}
& f\ x\ (pld\ f\ z\ [\,])\ =\ pld\ f\ (f\ z\ x)\ [\,] \\
\equiv & \\
& f\ x\ z\ =\ f\ z\ x
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
& f\ x\ (pld\ f\ z\ (y : ys))\ =\ pld\ f\ (f\ z\ x)\ (y : ys) \\
\equiv & \{\ \text{definición de } pld\ \} \\
& f\ x\ (f\ y\ (pld\ f\ z\ ys))\ =\ f\ y\ (pld\ f\ (f\ z\ x)\ ys) \\
\equiv & \{\ f \text{ asociativa y conmutativa}\ \} \\
& f\ y\ (f\ x\ (pld\ f\ z\ ys))\ =\ f\ y\ (pld\ f\ (f\ z\ x)\ ys) \\
\Leftarrow & \\
& f\ x\ (pld\ f\ z\ ys)\ =\ pld\ f\ (f\ z\ x)\ ys \\
\equiv & \\
& \mathit{HI}
\end{array}
$$

Veamos ahora

$$pli\ f\ =\ pld\ f$$

o equivalentemente:

$$\begin{array}{l} \forall\, xs \;\centerdot\; xs \;::\; [a] \;\centerdot \\ \qquad (\forall\, f, z \;\centerdot\; pli\ f\ z\ xs \;=\; pld\ f\ z\ xs) \end{array}$$

por inducción sobre $xs$, con lo cual hay que probar:

$$\begin{array}{l} \quad \forall\, f, z \;\centerdot\; pli\ f\ z\ [\,] \;=\; pld\ f\ z\ [\,] \\ \wedge \\ \quad \forall\, x \;\centerdot\; (\forall\, f, z \;\centerdot\; pli\ f\ z\ xs \;=\; pld\ f\ z\ xs) \\ \Rightarrow \\ \quad \forall\, f, z \;\centerdot\; pli\ f\ z\ (x : xs) \;=\; pld\ f\ z\ (x : xs)) \end{array}$$

— *Caso Base*: (trivial)

— *Paso Inductivo*:

$$\begin{array}{ll} & pli\ f\ z\ (x : xs) \;=\; pld\ f\ z\ (x : xs) \\ \equiv & \{\, foldlfoldr \,\} \\ & pli\ f\ (f\ z\ x)\ xs \;=\; f\ x\ (pld\ f\ z\ xs) \\ \equiv & \{\ \text{hipótesis de inducción}\ \} \\ & pld\ f\ (f\ z\ x)\ xs \;=\; f\ x\ (pld\ f\ z\ xs) \\ \Leftarrow & \\ & (1) \end{array}$$

**Solución al Ejercicio 6.57** (pág. 167).–

$$\begin{array}{l} esAnterior\ \_\ [\,] \;=\; True \\ esAnterior\ y\ (x : \_) \;=\; y \leq x \end{array}$$

$$\begin{array}{l} ordenada\ xs \;=\; foldr\ ord\ True\ xs \\ \quad \textbf{where}\ ord \;=\; \lambda\, u\ v \;\rightarrow\; (esAnterior\ u\ (tail\ xs))\ \&\&\ v \end{array}$$

**Solución al Ejercicio 6.58** (pág. 167).– Siendo $wh... \equiv \textbf{where}\, p \;=\; \ldots$, se observa que:

$$\begin{array}{ll} & copia[\,] \\ \equiv & \\ & \lambda y \rightarrow [\,] \end{array}$$

y por tanto:

$$\begin{array}{ll} & copia\ (x : xs) \\ \equiv & \\ & foldr\ p\ (\lambda y \rightarrow [\,])(x : xs) wh... \\ \equiv & \\ & p\ x\ (foldr\ p(\lambda y \rightarrow [\,])\ xs) wh... \\ \equiv & \\ & \lambda y \rightarrow y : copia\ xs \end{array}$$

$$
\begin{array}{l}
copia\ [\,]\ \ \ \ \ \ a\ =\ [\,] \\
copia\ (x : xs)\ a\ =\ a\ :\ copia\ xs\ a
\end{array}
$$

Es decir, la función *copia* duplica la estructura reemplazando todos los términos por el segundo argumento, con lo cual puede escribirse de forma más simple

$$copia\ xs\ =\ \lambda\, y\ \rightarrow\ map\ (\lambda\, \_\ \rightarrow\ y)\ xs$$

**Solución al Ejercicio 6.59** (pág. 167).– La función *alFinal* es un caso particular de la concatenación de listas ($alFinal\ x\ u \equiv\ u +\!\!+\ [x]$); por ello, en vez de usar $(+\!\!+)$ se puede dar directamente en términos de $(:)$

$$
\begin{array}{l}
alFinal\ ::\ a\ \rightarrow\ [a]\ \rightarrow\ [a] \\
alFinal\ x\ [\,]\ \ \ \ \ \ \ =\ [x] \\
alFinal\ x\ (y : ys)\ =\ y\ :\ alFinal\ x\ ys
\end{array}
$$

con lo cual se tiene:

$$
\begin{array}{l}
\ \ \ alFinal\ 7\ [4, 5] \\
\Longrightarrow \\
\ \ \ 4 : alFinal\ 7\ [5] \\
\Longrightarrow \\
\ \ \ 4 : 5 : alFinal\ 7\ [\,] \\
\Longrightarrow \\
\ \ \ 4 : 5 : [7]
\end{array}
$$

y ahora es fácil escribir la función de inversión:

$$
\begin{array}{l}
inv\ ::\ [a]\ \rightarrow\ [a] \\
inv\ =\ foldr\ (\lambda\, x\ u\ \rightarrow\ alFinal\ x\ u)\ [\,]
\end{array}
$$

**Solución al Ejercicio 6.60** (pág. 168).–

```
-- Apartado 0
longSec :: Secuencia a → Int
longSec (Uno _)     = 1
longSec (_ :=> s) = 1 + longSec s

-- Apartado 1
catSec :: Secuencia a → Secuencia a → Secuencia a
catSec (Uno x)    s = x :=> s
catSec (x :=> s) t = x :=> (catSec s t)

-- Apartado 2
invSec :: Secuencia a → Secuencia a
invSec (Uno x)    = Uno x
invSec (x :=> s) = catSec (invSec s) (Uno x)
```

```
-- Apartado 3
secALista :: Secuencia a → [a]
secALista (Uno x)     = [x]
secALista (x :=> s) = x : secALista s

-- Apartado 4
listaASec :: [a] → Secuencia a
listaASec [x]       = Uno x
listaASec (x : xs) = x :=> (listaASec xs)
```

Obsérvese que $listaASec\,[\,]$ no está definido: se dice que la función $listaASec$ no es exhaustiva (i.e., está parcialmente definida); esto se debe a que una secuencia siempre contiene información (¡una lista vacía no!).

**Solución al Ejercicio 6.61** (pág. 168).–

```
mapSec                :: (a → b) → Secuencia a → Secuencia b
mapSec f (Uno x)      = Uno (f x)
mapSec f (x :=> s) = (f x) :=> (mapSec f s)
```

**Solución al Ejercicio 6.62** (pág. 168).– La función $pliegaSec$ es:

```
pliegaSec :: (a → b → b) → b → Secuencia a → b
pliegaSec f z (Uno x)     = f x z
pliegaSec f z (x :=> s) = f x (pliegaSec f z s)
```

mientras que $secALista$ utilizará $pliegaSec$ con $b \equiv [a]$

```
secALista :: Secuencia a → [a]
secALista = pliegaSec (:) [ ]
```

## 21.5. EVALUACIÓN PEREZOSA. REDES DE PROCESOS

**Solución al Ejercicio 8.1** (pág. 188).– En primer lugar se determina una expresión que genere dicha lista infinita; sabemos que con la función $iterate$ de PRELUDE:

```
iterate f x = x : iterate f (f x)
```

se genera, mediante $iterate\ f\ a$, la lista:

$$[a,\ f\ a, f^2\ a,\ f^3\ a,\ f^4\ a,\ f^5\ a,\ \ldots\,]$$

Nuestra función $iterate'$ puede ser:

```
iterate' f x = y : x : iterate' f (f y) where y = f x
```

y la expresión pedida usará la función $take$ de PRELUDE:

$$take\ n\ (iterate'\ f\ a)$$

**Solución al Ejercicio 8.2** (pág. 188).– Siendo $sel \equiv selec$, $ord \equiv ordena$ se tiene la siguiente secuencia de reducciones:

$$\begin{array}{l} \quad sel\ 4\ (ord[1..]) \\ \Longrightarrow \\ \quad sel\ 4\ ((ord\ [y \mid y \leftarrow\ [2..],\ y < 1]) +\!\!+\ \ldots) \end{array}$$

y como $ord$ necesita la cabeza de $[y \mid y \leftarrow\ [2..],\ y < 1]$ habrá que evaluar dicha expresión:

$$\begin{array}{l} \quad [y \mid y \leftarrow [2..],\ y < 1] \\ \Longrightarrow \\ \quad filter\ (< 1)\ [2..] \\ \Longrightarrow \\ \quad filter\ (< 1)\ [3..] \\ \Longrightarrow \\ \quad \ldots \end{array}$$

Es decir, un cómputo perezoso no acabaría de evaluar la expresión en cuestión; la demostración de esto requiere inducción estructural sobre listas infinitas.

**Solución al Ejercicio 8.4** (pág. 191).– Demostremos el primer predicado por inducción sobre $n$:

— *Caso Base*: ($n \equiv 1$)

$$\begin{array}{ll} & aprox\ 1\ (iterate\ f\ x) \\ \equiv & \{\text{ definición de } iterate\ \} \\ & aprox\ 1\ (x\ :\ iterate\ f\ (f\ x)) \\ \equiv & \{\ 2)aprox\ \} \\ & x\ :\ aprox\ 0\ (iterate\ f\ (f\ x)) \\ \equiv & \{\text{ definición de } aprox\ \} \\ & x : \bot \end{array}$$

— *Paso Inductivo*:

$$\begin{array}{ll} & aprox\ (n+1)\ (iterate\ f\ x) \\ \equiv & \{\text{ definición de } iterate\ \} \\ & aprox\ (n+1)\ (x\ :\ iterate\ f\ (f\ x)) \\ \equiv & \{\ 2)aprox\ \} \\ & x\ :\ aprox\ n\ (iterate\ f\ (f\ x)) \end{array}$$

$\equiv$ { hipótesis de inducción }
$x : f\ x : f^2\ x : \ldots : f^{n-1}\ (f\ x) : \bot$

El segundo predicado se prueba de forma similar:

— *Caso Base*: ($n \equiv 1$)

$aprox\ 1\ u$ **where** $u = x : map\ f\ u$
$\equiv$ { cualificador **where** }
$aprox\ 1\ (x : map\ f\ u)$ **where** $u = x : map\ f\ u$
$\equiv$ { 2)$aprox$ }
$x : aprox\ 0\ (map\ f\ u)$ **where** $u = x : map\ f\ u$
$\equiv$ { definición de $aprox$ }
$x : \bot$

— *Paso Inductivo*:

$aprox\ (n+1)\ u$ **where** $u = x : map\ f\ u$
$\equiv$ { cualificador **where** }

$aprox\ (n+1)\ (x : map\ f\ u)$ **where** $u = x : map\ f\ u$
$\equiv$ { 2)$aprox$ }
$x : aprox\ n\ (map\ f\ u)$ **where** $u = x : map\ f\ u$
$\equiv$ { $(*)$ (ver después) }
$x : map\ f\ (aprox\ n\ u)$ **where** $u = x : map\ f\ u$
$\equiv$ { hipótesis de inducción }
$x : map\ f\ (x : f\ x : f^2\ x : \ldots : f^{n-1}\ x : \bot)$
$\equiv$ { prop. $map$ }
$x : f\ x : f^2\ x : \ldots : f^{n-1}\ (f\ x) : \bot$
$\equiv$
$x : f\ x : f^2\ x : \ldots : f^n\ x : \bot$

y queda probar por inducción sobre $n$:

$$(*) \qquad \forall\ n \,.\, n > 0\ .\ \forall\ u\ .\ u :: [a] \,.\, aprox\ n\ (map\ f\ u) = map\ f\ (aprox\ n\ u)$$

donde el caso base ($n \equiv 1$) es trivial, y para el paso inductivo hay que razonar según sea la lista $u$: si $u \equiv [\,]$ es trivial, mientras que para $x : u$:

$aprox\ (n+1)\ (map\ f\ (x : u)) = map\ f\ (aprox\ (n+1)\ (x : u))$
$\equiv$ { 2)$map$, 2)$aprox$ }

$$
\begin{array}{ll}
 & aprox\ (n+1)\ (f\ x\ :\ map\ f\ u)\ =\ map\ f\ (x\ :\ aprox\ n\ u) \\
\equiv & \{\ 2)aprox,\ 2)map\ \} \\
 & f\ x\ :\ aprox\ n\ (map\ f\ u)\ =\ f\ x\ :\ map\ f\ (aprox\ n\ u) \\
\Leftarrow & \\
 & HI
\end{array}
$$

Finalmente, (por la transitividad de $=$) aplicando el lema de la Sección 8.2.2:

$$xs\ =\ ys \qquad \equiv \qquad \forall\ n\ \centerdot\ n > 0\ \centerdot\ aprox\ n\ xs\ =\ aprox\ n\ ys$$

concluimos:

$$\underbrace{iterate\ f\ x}_{xs}\ =\ \underbrace{u\ \mathbf{where}\ u\ =\ x\ :\ map\ f\ u}_{ys}$$

**Solución al Ejercicio 8.5** (pág. 191).– Como $(1)$ es la composición de tres funciones, el tipo de dicha función será $Int\ \rightarrow\ [Int]$ (pues la primera que actúa es $iterate\ (div\ 10)$ y la última es $map\ (mod\ 10)$). En primer lugar analizamos qué devuelve $iterate\ (div\ 10)$ aplicada por ejemplo al natural $12345$:

```
PRELUDE> iterate (div 10)12345
[12345, 1234, 123, 12, 1, 0, 0, 0, ...] :: [Int]
```

Dicha lista infinita es la entrada a $takeWhile\ (\neq\ 0)$, función que devuelve la sublista formada por elementos de la cabeza de las lista argumento hasta que se cumpla la condición; por consiguiente:

```
PRELUDE> takeWhile (≠ 0) (iterate (div10) 12345)
[12345, 1234, 123, 12, 1] :: [Int]
```

Finalmente, al aplicar $map\ (mod\ 10)$ a la lista obtenida se tiene:

```
PRELUDE> map (mod 10) (takeWhile (≠ 0) (iterate (div 10) 12345))
[5, 4, 3, 2, 1] :: [Int]
```

con lo cual $(1)$ descompone un número en la lista de sus cifras en orden inverso. Suponiendo que $n$ es una variable unificada por cierta expresión externa (p.e., $n \equiv\ 2$), el tipo de $(2)$ será $[a]\ \rightarrow\ [[a]]$ (pues la primera que actúa es $iterate\ (drop\ n)$ y la última es $map\ (take\ n)$). En primer lugar analizamos qué devuelve $iterate\ (drop\ 2)$ aplicada por ejemplo a la lista $[1,2,3,4,5]$:

```
PRELUDE> iterate (drop 1) [1,2,3,4,5]
[[1,2,3,4,5], [3,4,5], [5], [], [], [], [], ...] :: [[Int]]
```

es decir, se obtiene una lista infinita de listas (en este caso, finitas). Tras pasar dicha lista de listas por $takeWhile\ (\neq\ [\,])$

```
PRELUDE> takeWhile (≠ []) (iterate (drop 1)[1,2,3,4,5])
[[1,2,3,4,5], [3,4,5], [5]] :: [[Int]]
```

y finalmente aplicarle $map\ (take\ 2)$ se obtiene:

$$\begin{array}{l}\text{PRELUDE}> \ map(take\ 2)\ (takeWhile\ (\neq\ [\,])\ (iterate\ (drop\ 1)\ [1,2,3,4,5])) \\ [[1,2],\ [3,4],\ [5]]\ ::\ [[Int]]\end{array}$$

con lo cual (2) devuelve una lista formada por sublistas de como máximo $n$ elementos de la lista original, respetando su orden.

**Solución al Ejercicio 8.6** (pág. 191).– Si se permitieran repeticiones sería fácil usando:

$$[pot\ [1..n] \mid n \leftarrow [1..]]$$

pero no vamos a permitir repeticiones. Considerando

$$\begin{array}{lll} potencias\ xs & = [[\,]] +\!\!+ pot\ xs\ [[\,]] \\ \\ pot\ (x:xs)\ res & = ac +\!\!+ pot\ xs\ (res +\!\!+ ac) \\ & \quad \textbf{where}\ ac\ =\ map\ (+\!\!+ [x])\ res \end{array}$$

la solución se irá construyendo de la siguiente forma:

$$\begin{array}{l} \quad potencias\ [1..] \\ \Longrightarrow \\ \quad [\,[\,],[1]\ \ldots\,] \\ \Longrightarrow \\ \quad [\,[\,],[1],[2],[1,2],\ \ldots\,] \\ \Longrightarrow \\ \quad [\,[\,],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3],\ \ldots\,] \\ \Longrightarrow \\ \quad \ldots \end{array}$$

**Solución al Ejercicio 8.7** (pág. 191).– Se tiene la siguiente secuencia de reducciones:

$$\begin{array}{l} \quad perfectos\ !!\ 1 \\ \Longrightarrow \\ \quad (filter\ perfecto\ [1..])\ !!\ 1 \\ \Longrightarrow\ \{\ (!!)\ \text{necesita la cabeza de}\ filter\ perfecto\ [1..]\ \} \\ \quad (filter\ perfecto\ [2..])\ !!\ 1 \\ \Longrightarrow \\ \quad \ldots \\ \Longrightarrow \\ \quad (filter\ perfecto\ [6..])\ !!\ 1 \\ \Longrightarrow \\ \quad (6 : filter\ perfecto\ [7..])\ !!\ 1 \\ \Longrightarrow \\ \quad (filter\ perfecto\ [7..])\ !!\ 0 \\ \Longrightarrow \end{array}$$

$$
\begin{array}{ll}
 & \ldots \\
\Longrightarrow & \\
 & (28 : \mathit{filter\ perfecto}\ [29..])\ !!\ 0 \\
\Longrightarrow & \\
 & 28
\end{array}
$$

Es decir, en este caso un cómputo perezoso sí terminaría de evaluar la expresión en cuestión; la demostración de este hecho requiere inducción estructural sobre listas infinitas.

**Solución al Ejercicio 8.8** (pág. 194).– Se tiene la siguiente secuencia de reducciones:

$$
\begin{array}{ll}
 & \mathit{suma}\ 3\ \mathit{primos} \\
\Longrightarrow & \{\ \mathit{primos}\ \} \\
 & \mathit{suma}\ 3\ (\mathit{map\ head}\ (\mathit{iterate\ criba}[2..])) \\
\Longrightarrow & \{\ \mathit{iterate}, \text{ pues } \mathit{suma} \text{ y } \mathit{head} \text{ necesitan la cabeza}\ \} \\
 & \mathit{suma}\ 3\ (\mathit{map\ head}\ ([2..] : \mathit{iterate\ criba}\ (\mathit{criba}\ [2..]))) \\
\Longrightarrow & \\
 & \mathit{suma}\ 3\ (\mathit{map\ head}\ ([2..] : [3..] : \mathit{iterate\ criba}\ [5, 7, 11, \ldots])) \\
\Longrightarrow & \\
 & \mathit{suma}\ 3\ (2 : \mathit{map\ head}\ ([3..] : \mathit{iterate\ criba}\ [5, 7, 11, \ldots])) \\
\Longrightarrow & \\
 & 2\ +\ \mathit{suma}\ 2\ (3 : \mathit{map\ head}\ (\mathit{iterate\ criba}\ [5, 7, 11, \ldots])) \\
\Longrightarrow & \\
 & 2\ +\ 3\ +\ \mathit{suma}\ 1\ (5 : \mathit{map\ head}\ (\mathit{iterate\ criba}\ [7, 11, \ldots])) \\
\Longrightarrow & \\
 & 5\ +\ 5\ +\ \mathit{suma}\ 0\ (7 : \mathit{map\ head}\ (\mathit{iterate\ criba}\ [11, \ldots])) \\
\Longrightarrow & \\
 & 10\ +\ 0 \\
\Longrightarrow & \\
 & 10
\end{array}
$$

**Solución al Ejercicio 8.9** (pág. 197).– Si generamos una lista infinita de pares en los que la primera componente sea el factorial y la segunda la lista (infinita) de los factoriales que quedan por generar, podemos después aplicar *first* a todos los elementos de la lista de pares (siendo *first* la función que extrae la primera componente de un par) para obtener la lista (infinita) de los factoriales:

$$
\begin{array}{l}
\mathit{facts}\ = \mathit{map\ first}\ (\mathit{iterate}\ f\ (1,\ [1..])) \\
\qquad\quad \textbf{where}\ f\ (p, x : xs)\ =\ (p * x, xs)
\end{array}
$$

**Solución al Ejercicio 8.12** (pág. 204).– Lo que se pretende es escribir unas funciones $\mathit{autom}$ y $\mathit{sigm}$ para que en vez de devolver una lista de unos y doses devuelva una lista de ternas en la que la primera componente sea el uno o el dos que corresponda, y la segunda (tercera) sea el número de unos (doses) que han aparecido hasta ese momento, con lo cual:

```
MAIN> auto
[1,2,2,1,1,2,...] :: [Int]


MAIN> autom
[(1,1,0),(2,1,1),(2,1,2),(1,2,2),(1,3,2),(2,3,3),...] :: [Int]
```

Para ello bastará con llevar un par de acumuladores para ir formando las ternas; aunque en principio podría pensarse en tomar la información acumulada de la terna recién mostrada, es mucho más fácil pasarle a *sigm* dos argumentos con el estado actual de los contadores, ya que cada llamada a dicha función genera una nueva terna para *autom*:

```
autom = [(1,1,0),(2,1,1)] ++ (sigm [2] 2 1 1)
sigm (x : xs) u c1 c2 = (x, c1', c2') : sigm (xs ++ nuevos) n_u c1' c2'
  where
    n_u    = if u== 1 then 2 else 1
    c1'    = if x== 1 then c1 + 1 else c1
    c2'    = if x== 2 then c2 + 1 else c2
    nuevos = if x== 2 then [n_u, n_u] else [n_u]
```

Obsérvese que la información suministrada por *autom* podría utilizarse para detectar ciclos.

**Solución al Ejercicio 8.13** (pág. 205).– Se utilizará una función auxiliar *contad* con dos acumuladores (uno para llevar el valor *act* actual que se está contando y otro para llevar el número actual *rep* de repeticiones del valor actual):

```
contadora :: [Int] → [Int]
contadora (x : xs) = contad xs x 1

contad :: [Int] → Int → Int → [Int]
contad []       _   rep = [rep]
contad (x : xs) act rep
  | x== act     = contad xs act (rep + 1)
  | otherwise   = rep : contad xs x 1
```

**Solución al Ejercicio 8.14** (pág. 206).– Aparte de todas las sucesiones de la forma $k, k, k, k, \ldots$, se pueden construir sucesiones solución de dicha ecuación generando a la vez la sucesión $s$ (de dos en dos elementos) y la expandida (con el valor "dictado" por lo que en ese momento tenga $s$); a modo de ejemplo veamos cómo se construye una solución que empiece por $[1, 1]$:

| $s$ | $expande\ s$ |
|---|---|
| $\Longrightarrow$ | |
| $[1,1,...]$ | $[1,...]$ |
| $\Longrightarrow$ | |
| $[1,1,1,1,...]$ | $[1,1,...]$ |
| $\Longrightarrow$ | |
| $[1,1,1,1,2,1,...]$ | $[1,1,1,1,...]$ |
| $\Longrightarrow$ | |
| $[1,1,1,1,2,1,1,2,...]$ | $[1,1,1,1,2,...]$ |
| $\Longrightarrow$ | |
| $[1,1,1,1,2,1,1,2,2,1,...]$ | $[1,1,1,1,2,1,1,...]$ |
| $\Longrightarrow$ | |
| $[1,1,1,1,2,1,1,2,2,1,2,2,...]$ | $[1,1,1,1,2,1,1,2,2,...]$ |
| $\Longrightarrow$ | |
| $...$ | $...$ |

Obsérvese que $comprime\ (expande\ s)\ \equiv\ [4,1,...]$ que no coincide con $s$.

**Solución al Ejercicio 8.10** (pág. 199).– Antes de nada dejaremos claro que el programa que vamos a diseñar servirá para comprobar el teorema de Lucas, no para demostrarlo (una demostración del teorema de Lucas puede verse en [Knuth, 1968]); nuestro programa va a hacer uso de dos listas infinitas: la lista *fibs* de los números de Fibonacci (ver Sección 8.3.5)

$$\begin{array}{l}\text{MAIN}>\ fibs\\ [\,1,\ 1,\ 2,\ 3,\ 5,8,\ \ldots\,]\ ::\ [Int]\end{array}$$

y una lista $pares$ de los pares de naturales sin el cero (ver Ejercicio 8.19):

$$[(1,1),(2,1),(1,2),(3,1),(2,2),(1,3),(4,1),(3,2),(2,3),(1,4),\ldots]$$

de la cual sólo generaremos los que verifiquen que la primera componente es mayor que la segunda

$$\text{MAIN}>\ pares\ [(2,1),(3,1),(4,1),(3,2),\ \ldots\,]\ ::\ [(Int,Int)]$$

debido a la simetría del problema y a que trivialmente el teorema es válido para índices iguales:

$$pares\ =\ [\,(x-n+1,\ n)\mid x\ \leftarrow\ [2..],\ n\ \leftarrow\ [1..(x\ `div`\ 2)]\,]$$

o bien generándolos en otro orden:

$$\text{MAIN}>\ pares\ [(2,1),(3,1),(3,2),(4,1),(4,2),(4,3),\ \ldots\,]\ ::\ [(Int,Int)]$$

con lo cual

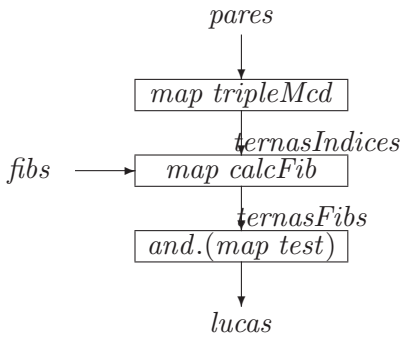


**Figura 21.1**: Comprobación del teorema de Lucas..

$$pares \;=\; [\,(x,y)\mid x \;\leftarrow\; [2..],\; y \;\leftarrow\; [1..(x-1)]\,]$$

En la Figura 21.1 se muestra el proceso a seguir: a partir de *pares* primero generamos una lista *ternasIndices* infinita de ternas añadiendo a cada par de la lista de entrada una tercera componente con el máximo común divisor de ambas:

$$\begin{aligned} ternasIndices \;&=\; map\ tripleMcd\ pares \\ &\quad\ \mathbf{where}\ tripleMcd \;=\; \lambda\,(x,y) \;\rightarrow\; (x,\; y,\; gcd\ x\ y) \end{aligned}$$

A partir de *ternasIndices* y con la ayuda de *fibs* podemos usar las componentes de cada terna como índice de un número de Fibonacci para obtener una lista *ternasFibs* de ternas de la forma $(f_n,\ f_m,\ f_{mcd(n,m)})$:

$$\begin{aligned} &ternasFibs \;= \\ &\quad map\ calcFib\ ternasIndices\ \mathbf{where} \\ &\qquad calcFib \;=\; \lambda\,(n,m,mcdnm) \;\rightarrow \\ &\qquad\quad (selec\ n\ fs,\ selec\ m\ fs,\ selec\ mcdnm\ fs) \\ &\qquad\qquad \mathbf{where}\ fs \;=\; fibs \end{aligned}$$

Finalmente, hay que realizar un *test* a cada una de las ternas y comprobar si todos los resultados son *True*:

$$\begin{aligned} lucas \;&=\; and\ (map\ test\ ternasFibs) \\ &\qquad \mathbf{where}\ test \;=\; \lambda\,(fn,fm,fmcdnm) \;\rightarrow\; gcd\ fn\ fm == fmcdnm \end{aligned}$$

con lo cual la certeza del teorema de Lucas se daría con la no terminación de la llamada *lucas*.

**Solución al Ejercicio 8.11** (pág. 201).– En este caso, si escribimos

$$\begin{array}{cccccccccc} 0 & 1 & 2 & 3 & 4 & \ldots & & & & \\ 1 & 1 & 3 & 6 & 21 & \ldots & 1 & 3 & 6 & 21 & \ldots \end{array}$$

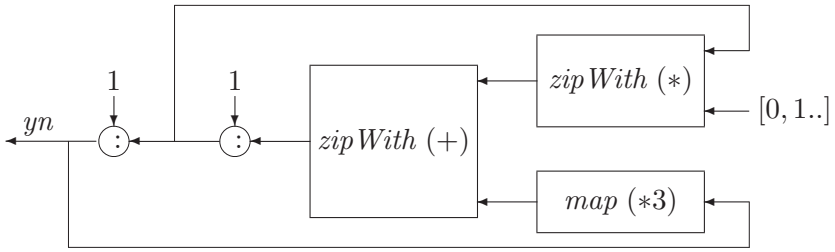


**Figura 21.2**: Red de procesos para la sucesión $yn$..

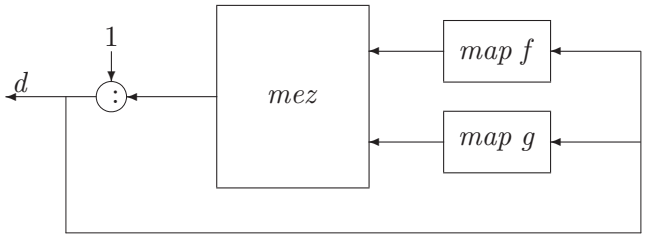


**Figura 21.3**: Primera versión del conjunto de Dijkstra..

donde la primera fila es la sucesión de naturales comenzando en 0, la segunda es la sucesión buscada y la tercera es la cola de dicha sucesión. Si multiplicamos la primera fila por la tercera y sumamos tres veces la segunda obtenemos:

$$3 \quad 6 \quad 21 \quad 81 \quad \ldots$$

que resulta ser la sucesión buscada salvo los dos primeros elementos. Por tanto, la solución es

$$yn \;=\; 1 : 1 : zipWith\ (+)\ (zipWith\ (*)\ (tail\ yn)\ [0, 1..]\ )\ (map\ (*3)\ yn))$$

y cuya red puede verse en la Figura 21.2.

**Solución al Ejercicio 8.15** (pág. 207).–

1. Se considera la red de la Figura 21.3 y las correspondientes funciones:

$$\begin{array}{l} dij\ f\ g\ = d\ \mathbf{where} \\ \qquad d\ =\ 1\ :\ mez\ u\ v\ \mathbf{where}\ u\ =\ map\ f\ d \\ \qquad\qquad\qquad\qquad\qquad\quad v\ =\ map\ g\ d \end{array}$$

   La justificación es la siguiente: $\mathcal{D}' = dij\ f\ g$ es una lista ordenada que verifica los axiomas $Ax1$ y $Ax2$, por lo que $\mathcal{D} \subseteq \mathcal{D}'$; para demostrar que $\mathcal{D}' = \mathcal{D}$ hay que

probar la otra inclusión (i.e., $\mathcal{D}' \subseteq \mathcal{D}$), lo cual puede hacerse por inducción ya que $\mathcal{D}'$ es bien construido (para el orden de $\mathbb{N}$):

$$
\begin{array}{l}
\quad \forall\, x \,\boldsymbol{.}\; x \in \mathcal{D}' \;\boldsymbol{.}\; x \in \mathcal{D} \\
\equiv \\
\quad 1 \in \mathcal{D}' \\
\wedge \\
\quad \forall\, x \,\boldsymbol{.}\; x \in \mathcal{D}' \,\boldsymbol{.}\; \\
\qquad (\forall\, y \,\boldsymbol{.}\; y < x\,,\; y \in \mathcal{D}' \;\boldsymbol{.}\; y \in \mathcal{D}) \quad \Rightarrow \quad x \in \mathcal{D}
\end{array}
$$

y ahora basta utilizar la siguiente propiedad común a $\mathcal{D}$ y $\mathcal{D}'$:

$$x \in \mathcal{D} \quad \equiv \quad \exists\, y \;\boldsymbol{.}\; y < x\,,\; y \in \mathcal{D} \,\boldsymbol{.}\; (x = f(y)) \vee (x = g(y))$$

3. Primero se construye un predicado *perfecto*

$$
\begin{array}{l}
perfecto\ x\ = \\
\quad x == foldr\ (+)\ 0\ [d \mid d \leftarrow [1..x-1],\ x\ `rem`\ d == 0]
\end{array}
$$

y el menor perfecto se obtendría mediante:

$$menorperf\ f\ g\ =\ head\ [x \mid x \leftarrow dij\ f\ g,\ perfecto\ x]$$

**Solución al Ejercicio 8.16** (pág. 207).– Se puede demostrar (véase [Ruiz, 2003]):

$$
\begin{array}{lcll}
a_0 & = & 1 & \\
a_{n+1} & = & min\, \mathcal{D}_n - \{\, a_0, a_1, \ldots, a_n \,\} & (n \geq 0)
\end{array}
$$

siendo

$$
\begin{array}{lcll}
\mathcal{D}_0 & = & \{\, a_0 \,\} & \\
\mathcal{D}_n & = & \{\, h(a_i, a_j) \mid 0 \leq i + j \leq n \,\} \cup \{\, a_0 \,\} & (n \geq 1)
\end{array}
$$

y teniendo en cuenta que:

$$\mathcal{D}_{n+1} \;=\; \mathcal{D}_n \cup \{\, h(a_i, a_j) \mid i + j = n + 1 \,\} \qquad (n \geq 0)$$

podemos simplificar el cálculo y utilizar la recurrencia:

$$
\begin{array}{lcll}
a_1 & = & h(a_0, a_0) & \\
\mathcal{U}_1 & = & \{\, a_1 \,\} & \\
a_{n+1} & = & min\, \mathcal{U}_n & \\
\mathcal{U}_{n+1} & = & (\mathcal{U}_n - \{\, a_{n+1} \,\}) \cup \{\, h(a_i, a_j) \mid i + j = n + 1 \,\} & (n \geq 1)
\end{array}
$$

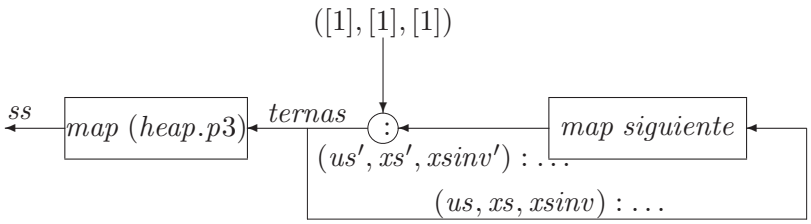


**Figura 21.4**: Segunda versión del conjunto de Dijkstra..

Necesitamos dos acumuladores para las listas

$$[\, a_0, a_1, \ldots, a_n \,] \qquad\qquad [\, a_n, a_{n-1}, \ldots, a_0 \,]$$

con objeto de calcular la diagonal $\{\ h(a_i, a_j)\ \mid\ i + j = n\ \}$; ésta se calcula

$$\begin{array}{l} diag\ [\,]\ \ \ [\,] \ = \ [\,] \\ diag\ (x : xs)\ (y : ys) \ = \ h\ x\ y\ :\ diag\ xs\ ys \end{array}$$

mientras que para el cómputo de

$$\begin{array}{lcl} a_{n+1} & = & min\, \mathcal{U}_n \\ \mathcal{U}_{n+1} & = & (\mathcal{U}_n - \{\ a_{n+1}\ \}) \ \cup\ \{\ h(a_i, a_j)\ \mid\ i + j = n + 1\ \} \qquad (n \geq 1) \end{array}$$

necesitamos una función que calcule el mínimo de una lista y el resto de elementos:

$$\begin{array}{l} m\_y\_resto\ [x] \qquad = \ (x,\ [\,]) \\ m\_y\_resto\ (x : xs) \\ \quad \mid x\ <\ m \qquad = \ (x,\ m : r) \\ \quad \mid otherwise \qquad = \ (m,\ x : r) \\ \quad \mathbf{where}\ (m, r) \quad = \ m\_y\_resto\ xs \end{array}$$

Finalmente, el cómputo del siguiente $\mathcal{U}_n$ se realiza a través de la función:

$$\begin{array}{l} siguiente\ (us,\ xs,\ xsinv) \ = \ (r +\!\!+\ diag\ xs'\ xsinv',\ xs',\ xsinv') \\ \quad \mathbf{where}\ (m, r) \ = \ m\_y\_resto\ us \\ \qquad\qquad xs' \quad = \ xs +\!\!+\ [m] \\ \qquad\qquad xsinv' \ = \ m\ :\ xsinv \end{array}$$

y ahora basta iterar la función *siguiente* en la forma

$$ternas \ = \ iterate\ siguiente\ ([1],\ [1],\ [1])$$

con lo cual los números del conjunto $\mathcal{S}$ se toman desde la cabeza de la tercera componente:

$$ss \ = \ map\ (head\ .\ p3)\ ternas\ \mathbf{where}\ p3\ (\_,\ \_,\ xs) \ = \ xs$$

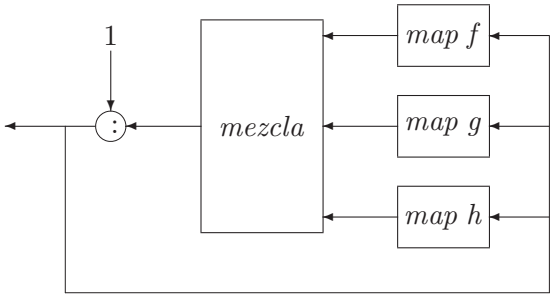


**Figura 21.5**: Tercera versión del conjunto de Dijkstra..

cuya red de procesos se muestra en la Figura 21.4, donde

$$us' \equiv r +\!\!+ diag\ xs'\ xsinv'$$

**Solución al Ejercicio 8.17** (pág. 207).– Lo más simple, aunque no se utilicen las desigualdades $f(x) > g(x) > h(x)$, es realizar el cómputo cuya red de procesos se muestra en la Figura 21.5; para aprovechar la información suministrada por dichas desigualdades habría que hacer un estudio parecido al del Ejercicio 8.16.

**Solución al Ejercicio 8.18** (pág. 207).– Para la primera expresión se tiene la siguiente secuencia de reducciones:

$$
\begin{array}{ll}
 & loca\ [3,5,2] \\
\Longrightarrow & \{\text{ definición de } loca \} \\
 & r\ \mathbf{where}\ (r,m) = foldr\ f\ ([\,],0)\ [3,5,2] \\
\Longrightarrow & \{\text{ plegados sucesivos} \} \\
 & r\ \mathbf{where}\ (r,m) = f\ 3\ (f\ 5\ (f\ 2\ ([\,],0))) \\
\Longrightarrow & \{\text{ aplicación perezosa de } f \} \\
 & r\ \mathbf{where}\ (r,m) = f\ 3\ (f\ 5\ (m : [\,], max\ 2\ 0)) \\
\Longrightarrow & \{\text{ aplicación perezosa de } f \} \\
 & r\ \mathbf{where}\ (r,m) = f\ 3\ (m : m : [\,], max\ 5\ (max\ 2\ 0)) \\
\Longrightarrow & \{\text{ aplicación perezosa de } f \} \\
 & r\ \mathbf{where}\ (r,m) = (m : m : m : [\,], max\ 3\ (max\ 5\ (max\ 2\ 0))) \\
\Longrightarrow & \{\text{ se necesita } r\text{, la cual necesita } m \} \\
 & r\ \mathbf{where}\ (r,m) = (m : m : m : [\,], max\ 3\ (max\ 5\ 2)) \\
\Longrightarrow & \{ max \text{ es estricta en sus dos argumentos} \} \\
 & r\ \mathbf{where}\ (r,m) = (m : m : m : [\,], max\ 3\ 5) \\
\Longrightarrow & \{ max \text{ es estricta en sus dos argumentos} \} \\
 & r\ \mathbf{where}\ (r,m) = (m : m : m : [\,], 5) \\
\Longrightarrow & \{\text{ cualificador } where \} \\
 & [5,5,5]
\end{array}
$$

donde observamos que *loca* reemplaza todos los elementos por el mayor pero dando una única pasada a la lista argumento; por su parte, la segunda expresión simplemente devuelve el máximo de dicha lista.

**Solución al Ejercicio 8.19** (pág. 208).–

1. La función $menorPar$ se puede definir mediante

$$\begin{array}{l} menorPar :: Par \rightarrow Par \rightarrow Bool \\ menorPar\ (x,y)\ (u,v) \mid x+y < u+v \ = \ True \\ \qquad\qquad\qquad\quad \mid x+y == u+v \ = \ v > y \\ \qquad\qquad\qquad\quad \mid otherwise \quad = \ False \end{array}$$

2. Para definir $mezcla$ en función de un operador infijo:

$$mezcla\ ps\ ps' \ = \ ps \ <> \ ps'$$

se puede hacer utilizando la relación de orden del apartado anterior:

$$\begin{array}{l} infix\ 4\ <> \\ (<>) :: [Par] \rightarrow [Par] \rightarrow [Par] \\ ps \quad <> [\,] \quad = ps \\ [\,] \quad <> ps \quad = ps \\ (x,y) : ps \ <> \ (u,v) : ps' \\ \quad \mid menorPar\ (u,v)\ (x,y) \ = (u,v) : (\ (x,y) : ps \ <> \ ps'\ ) \\ \quad \mid (x,y) == (u,v) \quad = (x,y) : (\ ps \ <> \ ps'\ ) \\ \quad \mid otherwise \quad = (x,y) : (\ ps \ <> \ (u,v) : ps'\ ) \end{array}$$

El resto de las funciones que aparecen en la red queda:

$$\begin{array}{l} inx\ ((x,y) : rs) \ = \ (x+1,y) \ : \ inx\ rs \\ iny\ ((x,y) : rs) \ = \ (x,y+1) \ : \ iny\ rs \\ pos \ = \ (0,0) : (\ (inx\ pos) \ <> \ (iny\ pos)\ ) \end{array}$$

La justificación es la siguiente:

- ✓ si $u$ y $v$ son listas ordenadas, $mezcla\ u\ v$ es una lista ordenada
- ✓ si $(x,y) \leftarrow pos$, entonces $siguiente\ (x,y) \ \leftarrow \ pos$, donde

$$\begin{array}{l} siguiente :: Par \rightarrow Par \\ siguiente\ (x,y) \\ \quad \mid x == 0 \ = (y+1, 0) \\ \quad \mid x \ > \ 0 \ = (x-1, \ y+1) \end{array}$$

En efecto: Es

$$\begin{array}{ll} & x > 0 \wedge (x, y) \leftarrow pos \\ \Rightarrow & \{\ inx\ \} \\ & (x-1, y) \leftarrow pos \\ \Rightarrow & \{\ iny\ \} \\ & (x-1, y+1) \leftarrow pos \end{array} \qquad \begin{array}{ll} & (0, y) \leftarrow pos \\ \equiv\equiv & \{\,\text{inducción sobre } y\,\} \\ & (y+1, 0) \leftarrow pos \end{array}$$

más, obsérvese que

$$pos \;=\; iterate\ siguiente\ (0,0)$$

o también (según el lema de la Sección 8.2.2)

$$pos \;=\; u\ \mathbf{where}\ u \;=\; (0,0) : map\ siguiente\ u$$

3. Con una ampliación de la forma

$$pos \longrightarrow \boxed{\text{ampli}} \longrightarrow pares$$

$$\begin{array}{l} ampli\ (\ (x,y) : ps\ ) \\ \quad |\ y{==}0 \quad\ = (x,y) : (-x,y) : ampli\ ps \\ \quad |\ x{==}0 \quad\ = (x,y) : (x,-y) : ampli\ ps \\ \quad |\ otherwise\ = (x,y) : (x,-y) : (-x,y) : (-x,-y) : ampli\ ps \end{array}$$

$$pares \;=\; (0,0) \;:\; ampli\ (tail\ pos)$$

4. Los flujos $inx\ ps$ y $pos$ quedan:

$$\begin{array}{ll} inx\ ps & = [\ (x+1, y)\ |\ (x,y) \leftarrow\ ps\ ] \\ pos & = [\ (n-y, y)\ |\ n \leftarrow\ [0..],\ y \leftarrow\ [0..n]\ ] \end{array}$$

mientras que para $pares$ se puede usar una función $sin$ auxiliar:

$$\begin{array}{l} sin\ x\ y \\ \quad |\ y == 0 \quad\ = [\ (x,y),\ (-x,y)\ ] \\ \quad |\ x == 0 \quad\ = [\ (x,y),\ (x,-y)\ ] \\ \quad |\ otherwise\ = [\ (x,y),\ (x,-y),\ (-x,y),\ (-x,-y)\ ] \end{array}$$

$$pares \;=\; (0,0) : [\ p\ |\ n \leftarrow\ [1..],\ y \leftarrow\ [0..n],\ x \;=\; n \;-\; y,\ p \leftarrow\ sin\ x\ y\ ]$$

5. La función *pares* se puede utilizar para simplemente encontrar los ceros de una función de dos variables, por ejemplo:

$$sol \;=\; [\,(x,y) \mid (x,y) \leftarrow \; pares,\; x^3 + y^3 \; - 6 * x * y == 0\,]$$

o para encontrar ceros de ecuaciones diofánticas en un recinto:

$$\begin{array}{l} sol \;=\; head\,[\,(x,y,n) \mid n \leftarrow\; [1..], \\ \qquad\qquad\qquad\qquad\quad (x,y) \leftarrow\; tail\,(menores\; pares\; n), \\ \qquad\qquad\qquad\qquad\quad 16 * x^3 - n * y^2 - 64 * x * y == 0\,] \\ \qquad \textbf{where}\; menores\;((x,y) : ps)\; n \\ \qquad\qquad\quad \mid abs(x) + abs(y) \leq n \quad = (x,y) : menores\; ps\; n \\ \qquad\qquad\quad \mid otherwise \qquad\qquad\;\; = [\,] \end{array}$$

**Solución al Ejercicio 8.20** (pág. 208).–

1. Consideremos las estructuras de datos:

$$\begin{array}{l} \textbf{type}\; Per \;=\; [Int] \\ \textbf{data}\; Sim \;=\; A \mid B \mid C \mid D \mid E \mid F\; \textbf{deriving}\; Show \end{array}$$

y ahora cómo actúa una permutación sobre un conjunto de símbolos:

$$\begin{array}{l} infixr\; 5\; >->\; \\ (>->) \;::\; Per \;\rightarrow\; [a] \;\rightarrow\; [a] \\ p\; >->\; cs \;=\; [\,s \mid (x,y) \leftarrow\; zip\,[1..]\,p, \\ \qquad\qquad\qquad\quad (y',s) \leftarrow\; zip\,[1..]\,cs, \\ \qquad\qquad\qquad\quad y' == y\,] \end{array}$$

con lo cual tendríamos que

$$\begin{array}{l} \text{MAIN}{>}\; [2,3,1,4]\; >->\; [A,B,C,D] \\ [C,A,B,D] \;::\; [Sim] \end{array}$$

Otra forma de hacerlo es a través de funciones, considerando una permutación como una función

$$\begin{array}{l} g \;::\; [a] \;\rightarrow\; Int \;\rightarrow\; a \\ g\; p \;=\; \lambda\, x \;\rightarrow\; p\; !!\; (x-1) \end{array}$$

$$v\; >->\; u \;=\; map\;(\; g\; u\; .\; g\; v\;)\;[1..length\; v]$$

Utilizando $>->$ puede también definirse el producto (composición) de dos permutaciones

$$(*:) :: Per \rightarrow Per \rightarrow Per$$
$$u *: p = p >-> u$$

con lo cual se tendría

$$\text{MAIN}> [3,2,1] *: [2,3,1]$$
$$[1,3,2] :: Per$$

2. Utilizando el operador \\ de diferencia de listas:

$$per :: (Eq\ a) \Rightarrow [a] \rightarrow [[a]]$$
$$per\ [\,] = [[\,]]$$
$$per\ xs = [\, x : u \mid x \leftarrow xs,\ u \leftarrow per\ (xs \backslash\backslash [x])\,]$$

3. La búsqueda de una solución a la ecuación en cuestión se obtiene con:

$$sol = head\ [p \mid p \leftarrow per\ [1..4],\ p >-> [4,3,1,2] == [1,2,4,3]]$$

También podrían calcularse las permutaciones idempotentes de orden 3

$$idem = [\, p \mid p \leftarrow per\ [1..3],\ p >-> p == p\,]$$

mientras que el cálculo de las inversibles se haría con:

$$inver = [\, (p,q) \mid ps = per\ [1..3],\ p \leftarrow ps,\ q \leftarrow ps,\ p *: q == [1..3]\,]$$

4. Para obtener una trasposición de una permutación de orden $n$:

$$(x <> y)\ n = map\ g\ [1..n]\ \textbf{where}\ g\ u \mid u == x = y \mid u == y = x \mid True = u$$

5. Una forma poco elegante de abordar el problema del cálculo del signo sin utilizar las funciones definidas en los apartados anteriores es calculando el número de transposiciones necesarias mediante una función $n\_s$:

```
n_s :: Per → Int
n_s p = n_sa (zip [1..] p) 0

n_sa [ ]          n = n
n_sa ( (x, y) : rs )n
  | x== y  = n_sa rs n
  | True   = n_sa (cambia x y rs) (n + 1) where
             cambia x y ( (u, v) : rs )
               | x== v = (u, y) : rs
               | True  = (u, v) : cambia x y rs
```

para después simplemente comprobar su paridad:

```
signo :: Per → Int
signo p = if ((n_s p) `mod` 2 == 0) then 1 else − 1
```

Afortunadamente se puede hacer de forma más compacta:

```
signo p = if p == [1..n] then 1 else − signo ((x <> y) n > − > p)
  where n      = length p
        (x, y) = head [ (u, v) | (u, v) ← zip [1..] p, u ≠ v ]
```

donde si $p$ no es la permutación identidad (==$[1..n]$) entonces $(x, u)$ indica la posición $x$ del primer $y$ que no está en su sitio; de esta forma, los que están en su sitio ¡¡no se tocan!! y se garantiza el mínimo número de transposiciones. También se puede generar el signo conjuntamente con la permutación, lo cual será más interesante si se necesitarán todas las permutaciones (como es el caso del apartado siguiente):

```
umu = 1 : (−1) : umu

pers :: (Eq a) ⇒ [a] → [(Int, [a])]
pers [x] = [(1, [x])]
pers xs  = [ (−n ∗ s, x : u) | (s, x) ← zip umu xs,
                               (n, u) ← pers (xs\\ [x]) ]

signo p = s where
  (s, _) = head [ (s′, p′) | n = length p,
                             (s′, p′) ← pers [1..n],
                             p′== p ]
```

6. Para el cálculo de un determinante definiremos una función *produc* que nos calcule los productos:

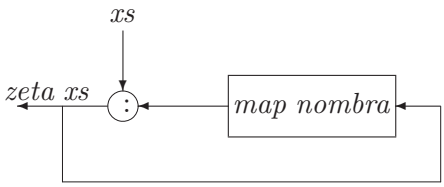


**Figura 21.6**: Red de procesos para computar `zeta`..

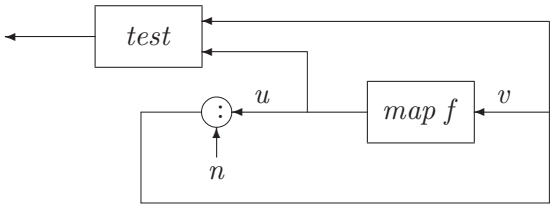


**Figura 21.7**: Red de procesos para computar `ap3`..

$$produc\ p\ fs\ =\ prod\ [\ f!!(i-1)\mid (f,i)\leftarrow\ zip\ fs\ p\ ]$$

con lo cual

$$deter\ fs\ =\ sum\ [\ (signo\ p)*(produc\ p\ fs)\mid p\leftarrow\ per\ [1..length\ fs]\ ]$$

o basándose en obtener las permutaciones y sus signos directamente:

$$deter\ fs\ =\ sum\ [\ s*(produc\ p\ fs)\mid (s,p)\leftarrow\ pers\ [1..length\ fs]\ ]$$

**Solución al Ejercicio 8.21** (pág. 209).–

1. Para definir $nombra$ utilizaremos una función auxiliar $nombra'$ con dos acumuladores (uno para los enteros y otro para las listas) que lleve por separado la cabeza y el resto del argumento que se le pasa a $nombra$:

$$\begin{array}{l} nombra\ (x:xs)\ =\ nombra'\ 1\ x\ xs\ [\,] \\ nombra'\ n\ x\ [\,]\qquad ys\ =\ ys +\!\!+ [n,x] \\ nombra'\ n\ x\ (x':xs)ys \\ \quad \mid x == x'\quad =\ nombra'\ (n+1)\ x\ xs\ ys \\ \quad \mid otherwise\ =\ nombra'\ 1\ x'\ xs\ (ys +\!\!+ [n,x]) \end{array}$$

2. La red de procesos para computar la lista $zeta$ se muestra en la Figura 21.6, cuya codificación es:

$$zeta\ xs\ =\ q\ \mathbf{where}\ q\ =\ xs\ :\ (map\ nombra\ q)$$

3. Siendo $ap3$ la función que computa la secuencia de elementos que verifican la condición dada, su red de procesos (ver Figura 21.7) es:

$$\begin{array}{l} test\ (u:us)\ (v:vs) \\ \quad \mid length\ u == length\ v\ =\ u\ :\ test\ us\ vs \\ \quad \mid otherwise\ \ \ \ \ \ \ \ \ \ \ \ \ \ =\ test\ us\ vs \end{array}$$

$$\begin{array}{l} ap3\ xs\ =\ test\ p\ q\ \mathbf{where}\ q\ =\ xs\ :\ p \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad p\ =\ map\ nombra\ q \end{array}$$

4. El estudio de las distintas conjeturas responde al mismo esquema del apartado 3 definiendo convenientemente $test$, $q$ y $p$; por ejemplo, para la conjetura $(b)$ habrá que utilizar:

$$testb\ p\ q\ =\ and\ [\ length\ y \leq length\ x\ \mid (x,y)\ \leftarrow\ zip\ q\ p\ ]$$

$$\begin{array}{l} cjb\ xs\ =\ testb\ p\ q\ \mathbf{where}\ q\ =\ zeta\ xs \\ \qquad\qquad\qquad\qquad\qquad\quad p\ =\ tail\ q \end{array}$$

y las demás conjeturas se codifican igual; para la $(c)$ se obtiene que en efecto todas terminan con el mismo final de la lista inicial (p.e., todas las sublistas de $zeta\ [3]$ terminan en 3).

**Solución al Ejercicio 8.22** (pág. 210).– Se observa que:

$$\begin{array}{ll} & fix\ f\ a\ \mathbf{where}\ f\ x\ q\ =\ map\ (+x)\ (x:q(x+1)) \\ \equiv & \\ & f\ a\ (fix\ f) \\ \equiv & \\ & 2a\ :\ map\ (+a)\ (fix\ f\ (a+1)) \end{array}$$

pero en general

$$\begin{array}{ll} & map\ (+u)\ (fix\ f\ v) \\ \equiv & \{\ \text{definición de } fix\ \} \\ & map\ (+u)\ (\ map\ (+v)\ (v:fix\ f\ (v+1))\ ) \\ \equiv & \{\ map\ f.map\ g = map(f.g)\ \} \\ & map\ (+(u+v))\ (v:fix\ f\ (v+1)) \end{array}$$

$$\equiv$$
$$u + 2v \;:\; map\ (+(u+v))\ (v : fix\ f\ (v+1))$$

Así, si los términos de la sucesión $\{\ y_n\ \}$ son generados a través de:

$$y_0 : y_1 : \ \ldots\ : y_n : \ map\ (+\ x_n)\ (fix\ f\ (a+n))$$

se puede observar que

$$\begin{aligned} y_{n+1} &= x_n \ + \ 2a \ + \ 2n \\ x_{n+1} &= x_n \ + \ a \ + \ n \end{aligned}$$

de donde

$$y_{n+1} \ = \ y_n \ + \ a \ + n + 1$$

con lo cual la lista solución es:

$$[\ 2a,\ 3a+2,\ 4a+5, 5a+9,\ \ldots\ ]$$

**Solución al Ejercicio 8.23** (pág. 210).–

$$\begin{array}{lcll} m5 & = & iterate\ (+5)\ 5 & --\text{Ap 1} \\ p2 & = & iterate\ (*2)\ 2 & --\text{Ap 2} \\ tf & = & iterate\ not\ True & --\text{Ap 3} \\ es & = & iterate\ (\text{'*'}:)\ \text{"*"} & --\text{Ap 4} \end{array}$$

**Solución al Ejercicio 8.24** (pág. 210).–

$$\begin{array}{lcl} lfact1\ (x : xs) & = & x : (lfact1\ (zipWith\ (*)\ xs\ [1..])) \\ lfact & = & lfact1\ [1..] \end{array}$$

**Solución al Ejercicio 8.25** (pág. 210).– Obsérvese que la lista a calcular es:

$$[[1^2, 2^2, 3^2, \ldots],\ [1^3, 2^3, 3^3, \ldots], [1^4, 2^4, 3^4, \ldots],\ \ldots]$$

de donde

$$[1^{n+1}, 2^{n+1}, 3^{n+1}, \ldots]$$
$$\equiv$$
$$zipWith\ (*)\ [1..]\ [1^n, 2^n, 3^n, \ldots]$$
$$\equiv$$
$$f\ [1^n, 2^n, 3^n, \ldots]\ \mathbf{where}\ f\ =\ \lambda\, x \rightarrow\ zipWith\ (*)\ x\ [1..]$$

Entonces, lo que se tiene es:

$$[[1, 2, 3, \ldots],\ [1^2, 2^2, 3^2, \ldots],\ [1^3, 2^3, 3^3, \ldots],\ \ldots]$$

$$\equiv$$
$$iterate\ f\ [1..]$$

y finalmente

$$lpot\ =\ tail\ (iterate\ (zipWith\ (*)\ [1..])\ [1..])$$

**Solución al Ejercicio 8.26** (pág. 210).– En el Ejercicio 8.19 ya se vio una forma de resolver este problema; aquí daremos una alternativa:

$$\begin{array}{lcl} lista & = & lista1\ [1..] \\ lista1\ (x:xs) & = & pares\ x\ 1\ :\ (lista1\ xs) \\ pares\ 1\ y & = & [(1,y)] \\ pares\ x\ y & = & (x,y) : (pares\ (x-1)\ (y+1)) \end{array}$$

## 21.6. PROGRAMACIÓN CON ÁRBOLES Y GRAFOS

**Solución al Ejercicio 9.16** (pág. 237).– La función *reduce* queda así:

$$\begin{array}{l} reduce\ ::\ ([b]\ \to\ a\ \to\ b)\ \to\ Arb\ a\ \to\ b\ \to\ b \\ reduce\ f\ V\ \ \ \ \ \ \ z\ =\ z \\ reduce\ f\ (N\ c\ ds)\ z\ =\ f\ [\ reduce\ f\ d\ z\ |\ d \leftarrow\ ds\ ]\ c \end{array}$$

Las demás funciones se pueden definir usando *reduce* o sin usarlo; por ejemplo, *aplica* y *visita* sin usarlo pueden ser:

$$\begin{array}{l} aplica\ ::\ (a\ \to\ b)\ \to\ Arb\ a\ \to\ Arb\ b \\ aplica\ f\ V\ \ \ \ \ \ \ =\ V \\ aplica\ f\ (N\ c\ as) =\ N\ (f\ c)\ (map\ (aplica\ f)\ as) \end{array}$$

$$\begin{array}{l} visita\ V\ \ \ \ \ \ \ =\ [\,] \\ visita\ (N\ c\ as)\ =\ c :\ (concat\ [visita\ a | a \leftarrow\ as]) \end{array}$$

y en función de *reduce*

$$\begin{array}{rcl} visita\ a & = & reduce\ f\ a\ [\,]\ \mathbf{where}\ f\ as\ c\ =\ c\ :\ concat\ as \\ \text{-- o bien} & = & reduce\ (\lambda\ as\ c\ \to\ c\ :\ concat\ as)\ a\ [\,] \\ aplica\ f\ a & = & reduce\ g\ a\ V\ \mathbf{where}\ g\ as\ x\ =\ N\ (f\ x)\ as \\ \text{-- o bien} & = & reduce\ (\lambda\ as\ x\ \to\ N\ (f\ x)\ as)\ a\ V \\ prof\ a & = & reduce\ h\ a\ 0\ \mathbf{where}\ h\ bs\ \_\ =\ 1\ +\ (foldr\ max\ 0\ bs) \\ \text{-- o bien} & = & reduce\ (\lambda\ bs\ \_\ \to\ 1\ +\ (foldr\ max\ 0\ bs))\ a\ 0 \end{array}$$

La función *pals* del último apartado se puede implementar mediante:

```
pals V              = [[]]
pals (N '.' ds)     = concat (map pals ds)
pals (N inicial ds) = map (inicial :) ( concat (map pals ds) )
```

o con una expresión más compacta:

```
pals a = [ rs | _ : rs ← reduce (λ ps c → map (c :) (concat ps))
                           a
                           [[]] ]
```

Por su parte, *aMayus* tiene una definición concisa:

```
aMayus = aplica toUpper
```

el predicado *estáEn* es fácil:

```
p `estáEn` (N '.' ds) = p `estáEnLista` ds

_          `estáEnLista` []          = False
[]         `estáEnLista` (V : _)     = True
p@(_ : _)  `estáEnLista` (V : ds)    = p `estáEnLista` ds
p@(c : cs) `estáEnLista` ((N c' ds') : ds)
  | c'== c  = cs `estáEnLista` ds'
  | c' < c  = p `estáEnLista` ds
  | c' > c  = False
```

y finalmente el operador de inserción ordenada se define:

```
(N '.' as) <: p = N '.' (inserl as p)

inserl []          []       = [V]
inserl as          []       = as
inserl []          (c : cs) = [N c (inserl [] cs)]
inserl (V : as)    (c : cs) = V : (inserl as (c : cs))
inserl ((N c' as') : as) (c : cs)
  | c'== c  = (N c' (inserl as' cs)) : as
  | c' < c  = (N c' as') : (inserl as (c : cs))
  | c' > c  = (N c (inserl [] cs)) : (N c' as') : as
```

Para hacer pruebas se puede utilizar:

```
dic = N '.' [ N 'a' [ V, N 'm' [ N 'a' [V],
                                 N 'o' [V] ],
                      N 'b' [V] ] ]

p = pals (dic <: "amor")
```

**Solución al Ejercicio 9.17** (pág. 239).– Se considera la siguiente definición:

$$\mathbf{data}\ Huf\ a\ =\ Hoja\ (a,\ Int)\ |\ Nodo\ (Huf\ a)\ Int\ (Huf\ a)\ \mathbf{deriving}\ Show$$

1. Se define una función *frec* para calcular la frecuencia acumulada en una $Hoja$ o en un $Nodo$:

$$\begin{array}{l} frec\ ::\ Huf\ a\ \rightarrow\ Int \\ frec\ (Hoja\ (\_, x))\ =\ x \\ frec\ (Nodo\ \_\ x\ \_)\ =\ x \end{array}$$

   Ahora se puede construir una función $arbIns$ para insertar un nodo de Huffman en una lista de árboles de Huffman:

$$\begin{array}{lll} arbIns\ ::\ Huf\ a\ \rightarrow\ [Huf\ a]\ \rightarrow\ [Huf\ a] & & \\ arbIns\ x\ [\,] & = & [x] \\ arbIns\ x\ (y : xs) & = & \mathbf{if}\quad (frec\ x \leq frec\ y) \\ & & \mathbf{then}\ x : (y : xs) \\ & & \mathbf{else}\ \ y : arbIns\ x\ xs \end{array}$$

   y la función $listAArb$, que dada una lista de hojas, construye el árbol de Huffman:

$$\begin{array}{l} listAArb\ ::\ [Huf\ a]\ \rightarrow\ Huf\ a \\ listAArb\ [x]\ =\ x \\ listAArb\ (x : (y : xs))\ = \\ \quad listAArb\ (arbIns\ (Nodo\ x\ (frec\ x + frec\ y)\ y)\ xs) \end{array}$$

   Suponemos que no existen errores en los mensajes.

2. Una primera forma (algo engorrosa) de implementar la función $codif\,1$ puede ser:

```
-- Primera opción
codif 1 c af = head (caminos c af)

caminos c (Hoja (x, _)) = if (x== c) then [[]] else []
caminos c (Nodo i _ d)  = pegar '0' (caminos c i) ++
                          pegar '1' (caminos c d)
pegar x []       = []
pegar x (y : ys) = (x : y) : (pegar x ys)
```

   Una alternativa para $codif\,1$ es:

```
-- Segunda opción
codif1 c (Hoja (_,_)) = []
codif1 c (Nodo i _ d) = if      (está c i)
                        then '0' : codif1 c i
                        else '1' : codif1 c d
está c (Hoja (x,_)) = c == x
está c (Nodo i _ d) = (está c i) || (está c d)
```

y como no hay dos sin tres:

```
-- Tercera opción
pliegae f g (Hoja x _)    = g x
pliegae f g (Nodo i _ d) = f (pliegae f g i) (pliegae f g d)

codif1 c ah = u
  where (u,v) =
    pliegae
      (λ(x,lx) (y,ly) →
        if lx then ('0' : x, True)
              else if ly then ('1' : y, True)
                         else ([], False))
      (λx → if x== c then ([], True)
                     else ([], False))
      ah
```

Sin embargo, hay una forma más elegante:

```
data Código = NoEstá | Está [Char]

codif1 :: a → Huf a → [Char]
codif1 c h = xs
  where
    Está xs = código h
    código (Hoja (x,_)) | x== c      = Está []
                        | otherwise = NoEstá
    código (Nodo i _ d) = (código i) || (código d)
    NoEstá || Está xs = Está (1 : xs)
    Está xs || _      = Está (0 : xs)
    _       || _      = NoEstá
```

3. La función *codif* puede definirse ahora de forma fácil (aunque muy ineficiente por el uso de ++):

$$
\begin{array}{l}
codif\ [\,]\qquad ah\ =\ [\,]\\
codif\ (x : xs)\ ah\ =\ codif1\ x\ ah +\!\!+ codif\ xs\ ah
\end{array}
$$

4. Finalmente, la función *decod* queda:

$$
\begin{array}{l}
decod\ [\,]\ \_\ \ =\ [\,]\\
decod\ xs\ ah\ =\ y : decod\ ys\ ah\ \mathbf{where}\ (y, ys)\ =\ decod1\ xs\ ah\\
\\
decod1\ xs\qquad (Hoja\ (y, \_))\ =\ (y, xs)\\
decod1\ ('0' : xs)\ (Nodo\ i\ \_\ \_)\ \ =\ decod1\ xs\ i\\
decod1\ ('1' : xs)\ (Nodo\ \_\ \_\ d)\ =\ decod1\ xs\ d
\end{array}
$$

Para hacer las pruebas se puede usar:

$$
\begin{array}{l}
lh\ =\ [\ Hoja(\text{'e'}, 4),\ Hoja(\text{'d'}, 15),\ Hoja(\text{'c'}, 21),\ Hoja(\text{'b'}, 25),\\
\qquad Hoja(\text{'a'}, 35)]\\
ah\ =\ listAArb\ lh
\end{array}
$$

**Solución al Ejercicio 9.18** (pág. 240).– Consideraremos un digrafo como una lista de vértices y una función sucesor:

$$\mathbf{data}\ Grafo\ a\ =\ G\ [a]\ (a\ \rightarrow\ [a])$$

1. Vamos a definir en primer lugar las funciones *entranEn* y *salenDe*:

$$
\begin{array}{l}
entranEn,\ salenDe\ ::\ a\ \rightarrow\ Grafo\ a\ \rightarrow\ Int\\
entranEn\ v\ (G\ vs\ suc)\ =\ length\ [\ u\ |\ u \leftarrow\ vs,\ v\ `elem`\ suc\ u\ ]\\
salenDe\ \ \ v\ (G\ \_\ suc)\ \ =\ length\ (suc\ v)
\end{array}
$$

para poder expresar el predicado *bal* en términos de *entranEn* y *salenDe*:

$$
\begin{array}{l}
bal\ ::\ Grafo\ a\ \rightarrow\ Bool\\
bal\ g@(G\ vs\ \_)\ =\ and\ [\ entranEn\ v\ g == salenDe\ v\ g\ |\ v \leftarrow\ vs\ ]
\end{array}
$$

2. La comprobación de si un digrafo no tiene vértices aislados es fácil:

$$
\begin{array}{l}
sinAisl\ ::\ Grafo\ a\ \rightarrow\ Bool\\
sinAisl\ g@(G\ vs\ \_)\ =\\
\quad and\ (map\ (\lambda\ v\ \rightarrow\ (entranEn\ v\ g\ >\ 0) || (salenDe\ v\ g\ >\ 0))\ vs)
\end{array}
$$

3. Si cada arco del digrafo lo representamos por un par (origen, destino):

```
type Arco a = (a, a)
```

el predicado *esCamino* quedaría:

```
esCamino :: [Arco a] → Bool
esCamino [_]                    = True
esCamino ((a, b) : (c, d) : xs) = (b== c) && esCamino ((c, d) : xs)
```

suponiendo que los arcos son válidos; en otro caso habrá que pasarle a *esCamino* el digrafo en cuestión

```
esCamino :: [Arco a] → Grafo a → Bool
esCamino [(a, b)]               (G vs suc) =
  (a `elem` vs) && (b `elem` (suc a))
esCamino ((a, b) : (c, d) : xs) g          =
  (b== c) && esCamino [(a, b)] g && esCamino ((c, d) : xs) g
```

4. La codificación de *esCicEul* se hace transcribiendo la propia definición:

```
esCicEul :: [Arco a] → Grafo a → Bool
esCicEul xs@((a, _) : _) g =
  esCamino xs g && a== d &&
  sinReps xs    && length xs == numArcosDe g
  where
    (_, d)                  = último xs
    último [arc]            = arc
    último (_ : as)         = último as
    sinReps []              = True
    sinReps (x : xs)        = x `notElem` xs && sinReps xs
    numArcosDe (G vs suc)   = foldr (+) 0 [ length (suc v) | v ← xs ]
```

5. En la clase *Grafo* disponemos de una función *camino* que devuelve un camino entre dos vértices:

```
class (Eq a) ⇒ Grafo a where
  ...
  camino :: a → a → [a]
  camino u v = head (caminoDesde u (λx → x== v) [])
```

Si contemplamos el caso de que no encuentre un camino entre ambos vértices:

```
hayCamino :: a → a → Bool
hayCamino u v = noVacía (caminoDesde u (λx → x== v) [])
  where noVacía (_ : _) = True
        noVacía []      = False
```

podremos utilizarla para definir nuestro predicado *conexo*:

```
conexo :: Grafo a → Bool
conexo (G vs _) = and [ hayCamino v w | v ← vs, w ← vs, v ≠ w ]
```

6. Si disponemos de una función *aListArcos* que dada una lista de vértices devuelve la lista de arcos correspondientes, podremos reflejar la propia definición a la hora de codificar el predicado *esEuleriano*:

```
esEuleriano :: Grafo a → Bool
esEuleriano g@(G vs _) =
  or [ esCicEul as g | as ← map aListArcos cs ] where
    cs = concat [ caminoDesde u (λx → x== v) [] |
                  u ← vs, v ← vs, u ≠ v ]
    aListArcos [o, d]   = [(o, d)]
    aListArcos a : b : as = (a, b) : aListArcos (b : as)
```

7. Antes de nada dejaremos claro que el programa que vamos a diseñar servirá para comprobar el teorema de Good, no para demostrarlo (una demostración del teorema de Good puede verse en [Knuth, 1968]); nuestro programa partirá de un generador de (una lista infinita de) digrafos:

```
gengrafs :: [Grafo a]
```

para a cada digrafo pasarle un predicado *good* descrito con el *sii* lógico:

```
sii :: Bool → Bool → Bool
x `sii` y = x `implica` y && y `implica` x
  where a `implica` b = (not a) or b

good :: Grafo a → Bool
good g = ((sinAisl g) && (esEuleriano g)) `sii` ((conexo g) && (bal g))

teorgood = and (map good gengrafs)
```

**Solución al Ejercicio 9.19** (pág. 241).–

1. Vamos a definir en primer lugar la función $gradoDe$ para grafos no dirigidos; si consideramos como ejemplo el grafo de la Figura 21.14:

```
figb17 = G [1..5] suc where
  suc 1 = [2,3]   ; suc 2 = [1,3,4,5] ; suc 3 = [1,2,4,5]
  suc 4 = [2,3,5]; suc 5 = [2,3,4]
```

   se observa que:

   - ✓ Aunque cada arco está representado dos veces, esto facilitará la definición de funciones subsiguientes
   - ✓ El número de arcos entrantes es igual al número de arcos salientes

   Con dicha representación, para definir $gradoDe$ se puede usar la función $entranEn$ o la función $salenDe$ del Ejercicio 9.18; preferimos ésta última por ser la más sencilla respecto a la representación elegida:

```
gradoDe :: a → Grafo a → Int
gradoDe v (G _ suc) = length (suc v)
```

   Ahora podemos expresar el predicado $reg$ en términos de $gradoDe$:

```
reg :: Grafo a → Bool
reg g@(G vs _) = ti [ gradoDe v g | v ← vs ]
  where ti [ ]          = True -- puede no ser conexo
        ti [_]          = True
        ti (x : y : ys) = (x== y) && ti (y : ys)
```

2. Comprobar si un grafo no dirigido no tiene vértices aislados es fácil:

```
sinAisl :: Grafo a → Bool
sinAisl g@(G vs _) = and [ gradoDe v g > 0 | v ← vs ]
```

3. Si cada arco del grafo no dirigido se representa por un par (origen, destino):

```
type Arco a = (a, a)
```

   el predicado $esCamino$ quedaría:

```
esCamino :: [Arco a] → Bool
esCamino [_]                  = True
esCamino ((a,b) : (c,d) : xs) = (b== c) && esCamino ((c,d) : xs) ||
                                (b== d) && esCamino ((d,c) : xs)
```

suponiendo que los arcos son válidos; en otro caso habrá que pasarle a *esCamino* el grafo no dirigido en cuestión

```
esCamino :: [Arco a] → Grafo a → Bool
esCamino [(a, b)]                (G vs suc) =
  (a `elem` vs) && (b `elem` (suc a))
esCamino ((a, b) : (c, d) : xs) g           =
  esCamino [(a, b)] g && ( (b== c) && esCamino ((c, d) : xs) g ||
                           (b== d) && esCamino ((d, c) : xs) g )
```

4. Con respecto al predicado *esCicEul* definido en el apartado 4 del Ejercicio 9.18, sólo hay que cambiar la definición de *sinReps* y *numArcosDe*:

```
esCicEul :: [Arco a] → Grafo a → Bool
esCicEul xs@((a, _) : _) g =
  esCamino xs g && a== d &&
  sinReps xs      && length xs == numArcosDe g
  where
    (_, d)                  =último xs
    último [arc]            = arc
    último (_ : as)         =último as
    sinReps [ ]             = True
    sinReps ((a, b) : xs)   = (a, b) `notElem` xs &&
                              (b, a) `notElem` xs &&
                              sinReps xs
    numArcosDe (G vs suc)   = (foldr (+) 0 [length (suc v)|v ← xs]) `div` 2
```

mientras que *esCamEul* se define igual que *esCicEul* cambiando $a == d$ por $a \neq d$.

5. El predicado *conexo* se define exactamente igual que en el apartado 5 del Ejercicio 9.18; además, si un grafo no dirigido es conexo, entre cualesquiera dos vértices existe un camino simple (i.e., un camino que no pasa más de una vez por los vértices del camino)

6. El predicado *esEuleriano* se define exactamente igual que en el apartado 6 del Ejercicio 9.18, mientras que *esSemieuleriano* se define igual que *esEuleriano* cambiando *esCicEul* por *esCamEul*; por su parte, *esNoEuleriano* se define en términos de los dos anteriores:

```
esNoEuleriano :: Grafo a → Bool
esNoEuleriano g = not (esEuleriano g) && not (esSemieuleriano g)
```

7. Antes de nada dejaremos claro que los programas que vamos a diseñar servirán para comprobar los teoremas en cuestión, no para demostrarlos; los dos primeros

teoremas suelen encontrarse en todos los libros de teoría de grafos (véase por ejemplo, las páginas 87–88 de [Cooke, 1985]), mientras que en la página 11 de [Biggs et al., 1976] se tiene una demostración del tercero (empezada por Euler y completada por C. Hierholzer en 1873). Utilizando la representación de los grafos no dirigidos y las funciones definidas en los apartados anteriores, diseñamos una función *impares* que dado un grafo determine cuántos nodos hay con grado impar:

```
impares :: Grafo a → Int
impares g@(G vs _) =
  length [ v | v ← vs, (gradoDe v g) `mod` 2 == 1 ]
```

Nuestro programa partirá de un generador de (una lista infinita de) grafos no dirigidos; por ejemplo, para generar los $K_n$ (grafo no dirigido con $n$ vértices tal que cada vértice está conectado directamente por medio de un arco con todos los demás, pero no con sí mismo):

```
gengrafs :: [Grafo a]
gengrafs = map ksubn [1..]
  where ksubn n = Grafo [1..n] (λ v → [1..n] \\ [v])
-- xs \\ v quita v de la lista xs
```

Cada uno de los grafos lo pasamos por el predicado de caracterización correspondiente:

```
sii :: Bool → Bool → Bool
x `sii` y = x `implica` y && y `implica` x
  where a `implica` b = (not a) or b

careuler, carsemie, carnoeul :: Grafo a → Bool
careuler  g = ((conexo g) && (esEuleriano g))      `sii` (impares g == 0)
carsemie  g = ((conexo g) && (esSemieuleriano g))`sii` (impares g == 2)
carnoeul  g = ((conexo g) && (esNoEuleriano g))  `sii` (impares g > 2)
```

con lo cual tenemos

```
teuler  = and (map careuler  gengrafs)
tsemie  = and (map carsemie gengrafs)
tnoeul  = and (map carnoeul  gengrafs)
```

Por último, plantear al lector la mejora de la solución propuesta en al menos tres aspectos:

- ✓ permitir que los grafos tengan *lazos* (arcos de un vértice a sí mismo), los cuales cuentan doble a la hora de calcular el grado de un vértice, y
- ✓ adaptar las funciones propuestas a *multigrafos* (grafos en los que el número de arcos que conectan directamente dos nodos puede ser mayor que 1).

✓ estudiar otros tipos de grafos no dirigidos que no sean los $K_n$.

## 21.7. PROGRAMACIÓN CON MÓNADAS

**Solución al Ejercicio 11.1** (pág. 268).– Vamos a demostrar por inducción que la instancia dada para el tipo *ÁrbolH* es realmente un funtor. Para (m1) tenemos un caso base para el constructor $VacíoH$:

$$
\begin{array}{ll}
 & fmap\ id\ VacíoH\ =\ id\ VacíoH \\
\equiv & \{\ \text{def. } fmap, \text{ def. } id\ \} \\
 & VacíoH\ =\ VacíoH
\end{array}
$$

y otro para el constructor $HojaH$:

$$
\begin{array}{ll}
 & fmap\ id\ (HojaH\ x)\ =\ id\ (HojaH\ x) \\
\equiv & \{\ \text{def. } fmap, \text{ def. } id\ \} \\
 & HojaH\ (id\ x)\ =\ HojaH\ x \\
\equiv & \{\ \text{def. } id\ \} \\
 & HojaH\ x\ =\ HojaH\ x
\end{array}
$$

La demostración del paso inductivo es:

$$
\begin{array}{ll}
 & fmap\ id\ (NodoH\ i\ d)\ =\ id\ (NodoH\ i\ d) \\
\equiv & \{\ \text{def. } fmap, \text{ def. } id\ \} \\
 & NodoH\ (fmap\ id\ i)\ (fmap\ id\ d)\ =\ NodoH\ i\ d \\
\equiv & \{\ \text{hipótesis de inducción}\ \} \\
 & NodoH\ (id\ i)\ (id\ d)\ =\ NodoH\ i\ d \\
\equiv & \{\ \text{def. } id\ \} \\
 & NodoH\ i\ d\ =\ NodoH\ i\ d
\end{array}
$$

Para (m2) tenemos nuevamente dos casos base:

$$
\begin{array}{ll}
 & fmap\ (g\ .\ f)\ VacíoH\ =\ (fmap\ g\ .\ fmap\ f)\ VacíoH \\
\equiv & \{\ \text{def. } fmap, \text{ def. } (.)\ \} \\
 & VacíoH\ =\ fmap\ g\ (fmap\ f\ \ VacíoH) \\
\equiv & \{\ \text{def. } fmap\ \} \\
 & VacíoH\ =\ fmap\ g\ \ VacíoH \\
\equiv & \{\ \text{def. } fmap\ \} \\
 & VacíoH\ =\ VacíoH
\end{array}
$$

y

$$
\begin{array}{ll}
 & fmap\ (g\ .\ f)\ (HojaH\ x)\ =\ (fmap\ g\ .\ fmap\ f)\ (HojaH\ x) \\
\equiv & \{\ \text{def. } fmap, \text{ def. } (.)\ \} \\
 & HojaH\ ((g\ .\ f)\ x)\ =\ fmap\ g\ (fmap\ f\ (HojaH\ x)) \\
\equiv & \{\ \text{def. } fmap, \text{ def. } (.)\ \} \\
 & HojaH\ (g\ (f\ x))\ =\ fmap\ g\ (HojaH\ (f\ x))
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \{\text{ def. } fmap \} \\
& HojaH\ (g\ (f\ x)) \ =\ HojaH\ (g\ (f\ x))
\end{array}
$$

Por último el paso inductivo:

$$
\begin{array}{ll}
& fmap\ (g\ .\ f)\ (NodoH\ i\ d) \ =\ (fmap\ g\ .\ fmap\ f)\ (NodoH\ i\ d) \\
\equiv & \{\text{ def. } fmap, \text{ def. } (.) \} \\
& NodoH\ (fmap\ (g\ .\ f)\ i)\ (fmap\ (g\ .\ f)\ d) \ =\ fmap\ g\ (fmap\ f\ (NodoH\ i\ d)) \\
\equiv & \{\text{ def. } fmap \} \\
& NodoH\ (fmap\ (g\ .\ f)\ i)\ (fmap\ (g\ .\ f)\ d) \\
& = \\
& fmap\ g\ (NodoH\ (fmap\ f\ i)\ (fmap\ f\ d)) \\
\equiv & \{\text{ def. } fmap \} \\
& NodoH\ (fmap\ (g\ .\ f)\ i)\ (fmap\ (g\ .\ f)\ d) \\
& = \\
& NodoH\ (fmap\ g\ (fmap\ f\ i))\ (fmap\ g\ (fmap\ f\ d)) \\
\equiv & \{\text{ def. } (.) \} \\
& NodoH\ (fmap\ (g\ .\ f)\ i)\ (fmap\ (g\ .\ f)\ d) \\
& = \\
& NodoH\ ((fmap\ g\ .\ fmap\ f)\ i)\ ((fmap\ g\ .\ fmap\ f)\ d) \\
\equiv & \{\text{ hipótesis de inducción} \} \\
& NodoH\ ((fmap\ g\ .\ fmap\ f)\ i)\ ((fmap\ g\ .\ fmap\ f)\ d) \\
& = \\
& NodoH\ ((fmap\ g\ .\ fmap\ f)\ i)\ ((fmap\ g\ .\ fmap\ f)\ d)
\end{array}
$$

luego el tipo *ÁrbolH* con estas definiciones verifica las propiedades de un funtor.

## 21.8. ALGORITMOS NUMÉRICOS PROGRAMADOS FUNCIONALMENTE

**Solución al Ejercicio 12.1** (pág. 305).– Como resulta que

$$
\begin{array}{ll}
& [a,\ f\ a,\ f^2\ a,\ \ldots,\ f^n\ a] \\
\equiv & \\
& a : [f\ a,\ f^2\ a,\ \ldots,\ f^n\ a] \\
\equiv & \\
& a : \langle \text{aplicar } f \text{ a los elementos de } [a,\ f\ a,\ \ldots,\ f^{n-1}\ a] \rangle
\end{array}
$$

tenemos la solución:

$$
\begin{array}{l}
itera\ 0\ \ \ \ \ \ \ \ \_\ a \ =\ [a] \\
itera\ (n+1)\ f\ a \ =\ a\ :\ map\ f\ (itera\ n\ f\ a)
\end{array}
$$

Tal solución es muy ineficiente ya que aplica $f$ demasiadas veces y no se aprovechan los valores ya calculados. Otra solución puede derivarse de la siguiente propiedad:

$$[f^n\ a,\ \ldots,\ f^2\ a,\ f\ a,\ a]$$

$$\equiv$$
$$f\ (f^{n-1}\ a)\ :\ [f^{n-1}\ a,\ \ldots,\ f^2\ a,\ f\ a,\ a]$$

de donde tenemos la ecuación:

$$itera\ (n+1)\ f\ a\ =\ f\ x\ :\ (x:u)\ \mathbf{where}\ x:u\ =\ itera\ n\ f\ a$$

El problema de tal solución es que se obtiene la lista al revés; sin embargo, la solución más eficiente es también la más elegante:

$$itera\ (n+1)\ f\ a\ =\ a\ :\ itera\ n\ f\ (f\ a)$$

con lo cual se tiene:

$$\begin{array}{l} progArit\quad prim\ d\ n\ =\ itera\ (n-1)\ (+d)\ prim \\ progGeom\ prim\ r\ n\ =\ itera\ (n-1)\ (*r)\ \ prim \end{array}$$

**Solución al Ejercicio 12.2** (pág. 311).– Sólo hay que modificar la función $tal$

$$\begin{array}{l} tal(p:ps)\ = \\ \quad (map\ (cocincr\ f\ x0)\ (iterate\ (q*)\ h0))\ :\ map(extrapola\ p)\ (tal\ ps) \end{array}$$

y la llamada correspondiente

$$\text{MAIN}>\ \ testabs\ 1.0e-5\ (map\ head\ (tal\ [1..]))$$

**Solución al Ejercicio 12.3** (pág. 312).– Lo único que hay que hacer es modificar la función $tal$

$$\begin{array}{l} tal\ (p:ps)\ = \\ \quad (sucesion\ sin\ 0\ pi)\ :\ map(extrapola\ p)\ (tal\ ps) \end{array}$$

pues la llamada correspondiente sigue siendo

$$\text{MAIN}>\ \ testabs\ 1.0e-5\ (map\ head\ (tal\ [2,4..]))$$

**Solución al Ejercicio 12.4** (pág. 312).– Una vez que $\mathbf{A} = \mathbf{LR}$, resolver $\mathbf{Ax} = \mathbf{LRx} = \mathbf{b}$ sólo requiere resolver dos sistemas con matriz de coeficientes triangular: uno $\mathbf{Ly} = \mathbf{b}$ con matriz triangular inferior mediante la denominada sustitución progresiva, y otro $\mathbf{Rx} = \mathbf{y}$ con matriz triangular inferior mediante la denominada sustitución regresiva. Nuevamente la perezosidad de los arrays HASKELL nos evita el engorro de determinar el orden en el que se van a obtener las incógnitas al resolver cada uno de los sistemas con matriz triangular. Para una sustitución progresiva tenemos:

$$x_k = \frac{1}{a_{kk}}\left(b_k - \sum_{i=1}^{k-1} a_{ki}x_i\right)$$

mientras que para una sustitución progresiva tenemos:

$$x_k = \frac{1}{a_{kk}} \left( b_k - \sum_{i=k+1}^{n} a_{ki} x_i \right)$$

con lo cual

```
sustProg :: Matriz Integer → MatrizInteger → Matriz Integer
sustProg a b = x where
  x =
    array ((1,1), dimsMatriz b)
      [((k,1), ( b!(k,1) - sum [ a!(k,i) * x!(i,1) | i ← range (1,k-1)]
               ) `div` a!(k,k)
       ) | (k,1) ← range ((1,1), dimsMatriz b) ]
```

y la función *sustRegr* se define de forma similar.

**Solución al Ejercicio 12.5** (pág. 312).– La librería *Complex* de números complejos define un tipo *Complex*:

```
data RealFloat a ⇒ Complex a = !a :+ !a deriving (Eq, Read, Show)
```

mediante un constructor $:+$ infijo (obsérvese que tanto la parte real como la imaginaria son estrictas), así como una función *conjugate*

```
conjugate           :: RealFloat a ⇒ Complex a → Complex a
conjugate(x :+ y) = x :+ (-y)
```

Luego lo único que hay que hacer es modificar ligeramente la función *tras* anterior:

```
trasconj ([] : _) = []
trasconj fs       = [ conjugate x | x : _ ← fs ] : trasconj [ xs | _ : xs ← fs ]
```

**Solución al Ejercicio 12.6** (pág. 316).– Una vez definido el tipo *NumReal*:

```
type NumReal = (Integer, Integer)
```

y sabiendo que la exponenciación $\uparrow$ del PRELUDE admite exponentes negativos, las funciones de normalización y denormalización pueden ser descritas fácilmente si no consideramos limitaciones en el exponente (aunque un sistema de punto flotante "de verdad" debería contemplar este aspecto):

$$
\begin{array}{l}
denormaliza \quad :: \; NumReal \rightarrow Double \\
denormaliza\ (m, e) \;=\; (fromInteger\ m) * (10.0 \uparrow (e - 4))
\end{array}
$$

$$
\begin{array}{l}
normaliza \quad :: \; Double \rightarrow NumReal \\
normaliza\ x \;=\; construye\ x\ 4
\end{array}
$$

$$
\begin{array}{l}
construye :: \; Double \rightarrow Integer \rightarrow NumReal \\
construye\ x\ e \mid abs\ x \;<\; 1000.0 \;=\; construye\ (x * 10.0)\ (e - 1) \\
\qquad\qquad\quad \mid abs\ x \;>\; 9999.0 \;=\; construye\ (x/10.0)\ (e + 1) \\
\qquad\qquad\quad \mid otherwise \qquad\;\; = \; (truncate\ x,\ e)
\end{array}
$$

Una primera forma (poco elegante) de definir las operaciones se basa en utilizar directamente las funciones *normaliza* y *denormaliza*:

$$
\begin{array}{l}
mult,\ divi,\ suma,\ rest :: \; NumReal \rightarrow NumReal \rightarrow NumReal \\
\text{– – Supondremos que los argumentos están normalizados} \\
\text{– – y devolveremos el resultado normalizado} \\
mult\ \ x\ y \;=\; normaliza\ (denormaliza\ x * \ denormaliza\ y) \\
divi\ \ \ x\ y \;=\; normaliza\ (denormaliza\ x\ /\ \ denormaliza\ y) \\
suma\ x\ y \;=\; normaliza\ (denormaliza\ x + denormaliza\ y) \\
rest\ \ \ x\ y \;=\; normaliza\ (denormaliza\ x - denormaliza\ y)
\end{array}
$$

pero aprovechando la representación utilizada se pueden obtener versiones más eficientes si se emplea *construye* y se realizan las operaciones con las mantisas en el conjunto *Double* en vez de en *Integer* para evitar el problema de los dígitos de reserva (aunque un sistema de punto flotante "de verdad"debería contemplar este aspecto):

$$
\begin{array}{l}
mult,\ divi,\ suma,\ rest :: \; NumReal \rightarrow NumReal \rightarrow NumReal \\
\text{– – Supondremos que los argumentos están normalizados} \\
\text{– – y devolveremos el resultado normalizado} \\
mult\ \ (mx, ex)\ (my, ey) \;= \\
\quad construye\ (fromInteger\ mx \ * \ fromInteger\ my)\ (ex + ey - 4) \\
divi\ \ \ (mx, ex)\ (my, ey) \;= \\
\quad construye\ (fromInteger\ mx\ /\ fromInteger\ my)\ (ex - ey + 4) \\
suma\ (mx, ex)\ (my, ey) \;= \\
\quad construye\ (fromInteger\ mx \ + \ denormaliza(my, ey - ex + 4))\ ex \\
\quad \text{– – La expresión } denormaliza(my, ey - ex + 4) \text{ se usa} \\
\quad \text{– – para igualar los exponentes de los dos operandos} \\
rest\ \ x \qquad\quad (my, ey) \;=\; suma\ x\ (-my, ey)
\end{array}
$$

**Solución al Ejercicio 12.7** (pág. 316).– Utilizando $foldr \equiv pliegad$ tenemos:

$$suma\ f\ a\ d\ n \;=\; foldr\ (+)\ 0\ [\ f\ (a + k * d) \mid k \leftarrow [n, n - 1..0]\ ]$$

Veamos de la verificación sólo el paso inductivo:

$$suma\ f\ a\ d\ (n + 1)$$

$$
\begin{array}{ll}
\equiv & \\
 & foldr\ (+)\ 0\ [\ f\ (a + k * d) \mid k \leftarrow\ [n+1, n..0]\ ] \\
\equiv & \\
 & foldr\ (+)\ 0\ (f\ (a + (n+1) * d))\ :\ [\ f\ (a + k * d) \mid k \leftarrow\ [n, n-1..0]\ ] \\
\equiv & \\
 & f\ (a + (n+1) * d)\ +\ foldr\ (+)\ 0\ [\ f\ (a + k * d) \mid k \leftarrow\ [n, n-1..0]\ ] \\
\equiv & \{\ \text{hipótesis de inducción}\ \} \\
 & f\ (a + (n+1) * d)\ +\ \sum_{k=0}^{n} f\ (a\ +\ kd) \\
\equiv & \\
 & \sum_{k=0}^{n+1} f\ (a\ +\ kd)
\end{array}
$$

**Solución al Ejercicio 12.8** (pág. 317).– La demostración requerida es

$$
\begin{array}{ll}
 & A(B) \\
\equiv & \{\ \text{sustituir } x \text{ por } B \text{ en la expansión de } A\ \} \\
 & a + B \times\ A_s(B) \\
\equiv & \{\ \text{expansión de la primera } B\ \} \\
 & a + (b + x \cdot\ B_s) \times\ A_s(B) \\
\equiv & \{\ \text{distributividad de } \times\ \} \\
 & (a + bA_s(B)) + x \cdot\ B_s\ \times\ A_s(B)
\end{array}
$$

que con $b = 0$ queda:

$$componer\ (a : as)\ be@(0 : bs)\ =\ a\ :\ bs * (componer\ as\ be)$$

El código correspondiente a la función generadora $A$ para árboles ordenados queda:

$$
\begin{array}{lll}
arbOrds & = & 0\ :\ bosques \\
bosques & = & componer\ listas\ arbOrds \\
listas & = & 1\ :\ listas
\end{array}
$$

lo cual permite obtener

$$
\begin{array}{l}
\text{MAIN}>\ arbOrds \\
[0,\ 1,\ 1,\ 2,\ 5,\ 14,\ 42,\ 132,\ 429,\ 1430,\ 4862,\ \ldots\ ]
\end{array}
$$

nuevamente los números de Catalán, pues puede demostrarse que $A = x \cdot T$ donde $T$ es la función generadora para enumerar árboles binarios.

**Solución al Ejercicio 12.9** (pág. 317).– Dado que la integral de un determinado término de una serie de potencias depende de su índice, usaremos una función auxiliar para controlarlo:

$$
\begin{array}{l}
integralDe\ as\ =\ 0\ :\ (integrar\ as\ 1)\ \mathbf{where} \\
\quad integrar\ (b : bs)\ n\ =\ b/n\ :\ (integrar\ bs\ (n+1))
\end{array}
$$

De esta manera tenemos:

$$\begin{aligned} expx &= 1 + (integralDe\ expx) \\ senx &= integralDe\ cosx \\ cosx &= 1 - integralDe\ senx \end{aligned}$$

con lo cual, por ejemplo

$$\begin{aligned} &\text{MAIN>}\ expx \\ &[1\,\%1,\ 1\,\%1,\ 1\,\%2,\ 1\,\%6,\ 1\,\%24,\ 1\,\%120,\ 1\,\%720,\ \ldots\,] \end{aligned}$$

**Solución al Ejercicio 12.10** (pág. 317).– Al igual que hicimos en el Ejercicio 12.9, usaremos una función auxiliar para controlar el índice del término a derivar:

$$\begin{aligned} &derivadaDe\ (\_ : as) = derivar\ as\ 1\ \mathbf{where} \\ &\quad derivar\ (b : bs)\ n = n * b\ :\ (derivar\ bs\ (n+1)) \end{aligned}$$

La comprobación requerida puede hacerse mediante:

$$\begin{aligned} &\text{MAIN>}\ (take\ 30\ (senx - sqrt(1 - cosx \uparrow 2))) == (take\ 30\ sp0) \\ &True \end{aligned}$$

**Solución al Ejercicio 12.11** (pág. 317).– La demostración requerida es

$$\begin{aligned} & x \\ \equiv & \quad \{\,\text{inversa funcional}\,\} \\ & A(B(x)) \\ \equiv & \quad \{\,\text{sustituir } x \text{ por } B \text{ en la expansión de } A\,\} \\ & a + B \times A_s(B) \\ \equiv & \quad \{\,\text{expansión de la primera } B\,\} \\ & a + (b + x \cdot B_s) \times A_s(B) \end{aligned}$$

que con $b = 0$ queda, igualando términos

$$x = a + x \cdot B_s \times A_s(B) \qquad \Rightarrow \qquad a = 0 \quad \wedge \quad B_s = 1/A_s(B)$$

y entonces el código correspondiente será

$$invertir\ (0 : as) = bs\ \mathbf{where}\ bs = 0\ :\ 1/(componer\ as\ bs)$$

$$prueba = senx/cosx - invertir(integralDe(1/(1 + x \uparrow 2)))$$

La comprobación requerida puede hacerse mediante:

$$\begin{aligned} &\text{MAIN>}\ (take\ 30\ prueba) == (take\ 30\ sp0) \\ &True \end{aligned}$$

## 21.9. PUZZLES Y SOLITARIOS

**Solución al Ejercicio 13.1** (pág. 322).– Siendo *vasijas* la función que devuelve todas las soluciones, vamos a parametrizar dicha función con respecto a las capacidades $mx$ y $my$ de las vasijas y al número $n$ de litros a aislar; para ello, lo más sencillo es que el estado sea quien vaya arrastrando las capacidades:

```
data Vasijas = V Int -- contenido vasija mayor (x)
                 Int -- contenido vasija menor (y)
                 Int -- capacidad vasija mayor (mx)
                 Int -- capacidad vasija menor (my)
  deriving (Show, Eq)
```

con lo cual las operaciones quedarían:

```
ops = [llX, llY, vaX, vaY, voXY, voYX]
llX   V _ y mx my = V mx y mx my            -- llenar X
llY   V x _ mx my = V x my mx my            -- llenar Y
vaX   V _ y mx my = V 0 y mx my             -- vaciar X
vaY   V x _ mx my = V x 0 mx my             -- vaciar Y
voXY  V x y mx my =                         -- volcar X sobre Y
      if s <= my    then V 0 s mx my        -- cabe todo
                    else V (s - my) my mx my -- sobra algo
                    where s = x + y
voYX  V x y mx my =                         -- volcar Y sobre X
      if s <= mx    then V s 0 mx my        -- cabe todo
                    else V mx (s - mx) mx my -- sobra algo
                    where s = x + y
```

De esta manera, la instancia de la clase *Grafo* para *Vasijas* no hay que modificarla y sólo hay que modificar *vasijas*:

```
vasijas mx my n = caminoDesde (V 0 0 mx my) test []
  where test (V n _ _ _) = True
        test (V _ n _ _) = True
        test _           = False
```

Ahora se puede construir fácilmente un predicado *aislable*:

```
aislable :: Int -> Int -> Int -> Bool
aislable mx my = noVacía . (vasijas mx my)
  where noVacía (x : _) = True
        noVacía _       = False
```

que es la piedra angular para definir un predicado *aislables* tal que dadas las capacidades $mx$ y $my$, determine si son aislables los múltiplos del máximo común divisor de $mx$ y $my$:

```
aislables :: Int → Int → Bool
aislables mx my =
  and [ aislable mx my mul |
                  let mcd = gcd mx my,
                      mul ← [ k * mcd | k ← [1..mx `div` mcd] ] ]
-- mcd ≤ k * mcd ≤ mx implica que 1 ≤ k ≤ mx/mcd
```

donde $mul$ sólo debe tomar valores de la lista (finita) con aquellos múltiplos del máximo común divisor de $mx$ y $my$ que obviamente no superen al mayor de $mx$ y $my$ (en nuestro caso, $mx$). Así, consideraremos la lista $pares$ infinita de los pares de capacidades (naturales sin el cero) que verifiquen que la primera componente es mayor que la segunda, ya que trivialmente la conjetura es válida para vasijas iguales:

```
pares = [ (x, n) | x ← [2..], n ← [1..x − 1] ]
```

con lo cual se tiene

```
MAIN> pares [ (2,1),(3,1),(3,2),(4,1),(4,2),(4,3), ... ] :: [(Int, Int)]
```

y la certeza de la conjetura de las vasijas se daría con la no terminación de la llamada $conjvas$:

```
conjvas = and [ aislables mx my | (mx, my) ← pares ]
```

**Solución al Ejercicio 13.2** (pág. 322).– Con respecto al problema de las vasijas planteado en el Ejercicio 13.1, lo único que hay que tener en cuenta en esta versión es:

- ✓ En lugar de una fuente de vino tenemos una botella (capacidad limitada)
- ✓ La botella de vino puede actuar como vasija
- ✓ No se puede tirar el vino, dado que es un bien mucho más preciado que el agua
- ✓ Se trata de repartir la capacidad de la botella, no de aislar un litro de vino

Por lo tanto, el problema lo podemos plantear como la generalización a tres vasijas del problema de las dos vasijas sin considerar las operaciones de llenado y vaciado de las vasijas (sólo las de volcado) y donde el número de litros a aislar sea la mitad de la capacidad de la vasija mayor:

```
data Vasijas = V Int -- contenido vasija mayor (z)
                 Int -- contenido vasija media (y)
                 Int -- contenido vasija menor (x)
                 Int -- capacidad vasija mayor (mz)
                 Int -- capacidad vasija media (my)
                 Int -- capacidad vasija menor (mx)
  deriving (Show, Eq)
```

```
ops = [voXY, voYX, voXZ, voZX, voYZ, voZY]
voXY (V z y x mz my mx) = -- volcar X sobre Y
      if s ≤ my then  V z s 0 mz my mx          -- cabe todo
                else  V z my (s − my) mz my mx  -- sobra algo
                where s = x + y
voYX (V z y x mz my mx) = -- volcar Y sobre X
      if s ≤ mx then  V z 0 s mz mx my          -- cabe todo
                else  V z (s − mx) mx mz my mx  -- sobra algo
                where s = x + y
...
```

y se definen de forma análoga el resto de las operaciones. De esta manera, sólo hay que modificar la función *vasijas* que se dio en el Ejercicio 13.1:

```
vasijas mz my mx = caminoDesde (V mz 0 0 mz my mx) test []
  where test (V _ n _ _ _ _) = True
        test (V _ _ n _ _ _) = True
        test _               = False
        n = mz `div` 2
```

**Solución al Ejercicio 13.3** (pág. 324).– Con respecto al problema original, sólo hay que evitar el hecho de que dos caníbales viajen en la barca con un misionero, ya que se lo comerían durante el trayecto. Para ello, lo único que hay que modificar del programa es los posibles modos de subir a la barca: aunque en principio sean cuatro los nuevos modos de subirse a la barca,

```
MAIN> pares [ (3,0), (2,0), (2,1), (1,0), (1,1), (0,1), (1,2), (0,2), (0,3) ]
```

hay que descartar $(1, 2)$ para que en la barca no se produzca ninguna desgracia:

```
MAIN> pares [ (3,0), (2,0), (2,1), (1,0), (1,1), (0,1), (0,2), (0,3) ]
```

lo cual puede obtenerse mediante:

```
-- Posibles modos de subir a la barca: (misioneros, caníbales)
pares = [ (x, y) | x ← [0..3], y ← [0..3], x + y ≤ 3, x ≥ y || x == 0 ]
```

**Solución al Ejercicio 13.4** (pág. 326).– Como la conjetura involucra la generación de diversos tableros (que supondremos todos triangulares) y el estudio para cada tablero de la existencia de soluciones según dónde se encuentre el hueco, vamos a parametrizar el problema con respecto a la lista $tab$ de saltos posibles (representación del tablero), al hueco inicial $h$ y al número $n$ de filas del tablero; para ello, arrastraremos cada tablero por todo el grafo considerando cada configuración como un par formado por las casillas ocupadas y por el tablero:

```
type Salto    = (Casilla, Casilla, Casilla)
type Tablero  = [Salto]
data Config   = C [Casilla] Tablero deriving (Show, Eq)
```

Cada salto será una tupla con la casilla $o$ de origen del salto, la casilla $c$ en la que está la ficha "a comer" y la casilla $d$ de destino del salto; de esta manera,

```
instance Grafo Config where
  suc (C os tab) =
    map (λx → C x tab) [ (d : os) \\ [o, c] | (o, c, d) ← tab,
                                              o `elem` os,
                                              c `elem` os,
                                              d `notElem` os ]

abreu (tab, n) h = caminoDesde (C (hueco h) tab) test []
  where hueco x       = [1..(numcas n)] \\ [x]
        test (C [_] _) = True
        test _         = False
```

donde la función *numcas* devuelve el número de casillas que tiene un tablero de $n$ filas:

```
numcas n = (n * (n + 1)) `div` 2
```

Ahora se puede construir fácilmente un predicado *resoluble*:

```
resoluble :: (Tablero, Int) → Casilla → Bool
resoluble (tab, n) = noVacía . (abreu (tab, n))
  where noVacía (x : _) = True
        noVacía _       = False
```

que es la piedra angular para definir un predicado *resolubles*, que dada una terna cuyas componentes son un tablero *tab*, el número de filas $n$ y una lista *hinis* formada por todos los posibles huecos iniciales, determine si el tablero es resoluble para todos y cada uno de los huecos iniciales:

```
resolubles :: (Tablero, Int, [Casilla]) → Bool
resolubles (tab, n, hinis) = and (map (resoluble (tab, n)) hinis)
```

Así, siguiendo el esquema de cómputo de la Figura 21.8, suponemos a nuestra disposición un generador *gentab* de tableros tal que podamos obtener una lista *ternas* infinita de ternas cuya primera componente sea un tablero de $n$ filas, su segunda componente sea $n$ y su tercera componente sea la lista [1]:

```
gentab :: Int → (Tablero, Int, [Casilla])
gentab n = (saltos n, n, [1])

ternas :: [(Tablero, Int, [Casilla])]
ternas = map gentab [3..]
```

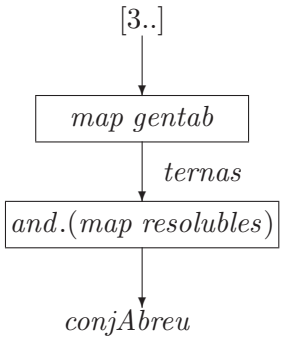


**Figura 21.8**: Comprobación de la conjetura de Abreu..

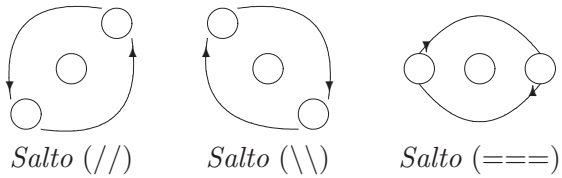


**Figura 21.9**: Tipos de saltos en tableros triangulares de Abreu..

pues dada la simetría del problema no hay por qué comprobar un tablero con los huecos en cada una de sus tres esquinas (basta con la superior, cuyo número es 1), y consideramos tableros de 3 filas en adelante. Así, la certeza de la conjetura de Abreu se daría si devuelve *False* la llamada *conjAbreu* definida mediante:

$$conjAbreu \;=\; and\;(map\;resolubles\;ternas)$$

Sólo nos resta describir la función *saltos*, por ejemplo:

MAIN> *saltos* 4

$$\begin{array}{l}[\,(1,2,4),(1,3,6),\;\;(2,4,7),(2,5,9),(3,5,8),\;\;(3,6,10),\\ \;\;(4,2,1),(4,5,6),\;\;(6,3,1),(6,5,4),(7,8,9),\;\;(7,4,2),\\ \;\;(8,5,3),(8,9,10),(9,5,2),\;(9,8,7),(10,6,3),(10,9,8)]\;::\;\;Tablero\end{array}$$

que a partir del tablero de $n-1$ filas sea capaz de generar el de $n$ filas si sabemos generar los saltos en diagonal hacia la izquierda y hacia la derecha desde las casillas de la fila $n-2$ a las de la $n$ (y viceversa) comiéndose las de la $n-1$, y también los saltos en horizontal en la fila $n$ (ver Figura 21.9):

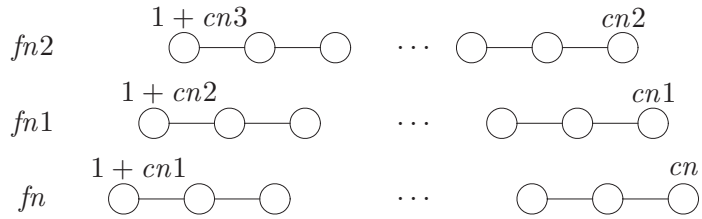


**Figura 21.10**: Generación recursiva de tableros triangulares de Abreu..

```
saltos :: Int → Tablero
saltos n
  | n < 3 = []
  | n ≥ 3 =
    saltos (n − 1)      ++ -- saltos tablero anterior
    ((//) fn2 fn1 fn)   ++ -- hacia SO (y hacia NE)
    ((\\) fn2 fn1 fn)   ++ -- hacia SE (y hacia NO)
    (===) fn               -- hacia E  (y hacia O )
    where cn2 = numcas (n − 2)
          cn1 = numcas (n − 1)
          fn2 = [numcas (n − 3) + 1 .. cn2]      -- fila n-2
          fn1 = [cn2            + 1 .. cn1]      -- fila n-1
          fn  = [cn1            + 1 .. numcas n] -- fila n
```

donde para construir las filas involucradas se aprovecha el hecho de que $numcas\ n$ devuelve (ver Figura 21.10) el número de la casilla inferior derecha de un tablero de $n$ filas usando la numeración de la Figura 21.9; la descripción del resto de las operaciones es fácil:

```
(//) []       _       _       = []
(//) (o : os) (c : cs) (d : ds) = (o, c, d) : (d, c, o) : (//) os cs ds

(\\) os       (_ : cs) (_ : _ : ds) = (//) os cs ds

(===) (o : c : d : os) = (o, c, d) : (d, c, o) : (===) (c : d : os)
(===) _                = []
```

Por último, plantear al lector la mejora de la solución propuesta en al menos tres aspectos:

- ✓ la descripción en forma de red de procesos de la función $gentab$,
- ✓ la descripción de la función $suc$ sin utilizar $tab$ (en función de $os$ y $n$) para evitar arrastrar el tablero entero por todo el grafo,
- ✓ el estudio de otro tipo de tableros que no sean triangulares.

**Solución al Ejercicio 13.5** (pág. 326).– Visto de nuevo el problema como la descripción de un camino en un grafo, el problema quedará resuelto si a partir de una determinada estructura de datos podemos describir la función sucesor del grafo; para ello, consideramos la siguiente codificación de las casillas del tablero

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}$$

y describimos una configuración como una lista *fs* de nueve enteros

```
type Casilla = Int                          --1..9
type Ficha   = Int                          --0..8
data Config  = C [Ficha] deriving (Show, Eq)
```

donde el hueco lo representaremos por 0 y en la casilla número $n$ estará la ficha indicada por el enésimo elemento de la lista (es decir, *fs* !! $(n-1)$); así, las configuraciones inicial y final mostradas en la Figura 13.4 vendrán dadas por

$$[3,1,4,7,8,5,6,0,2] \qquad\qquad [1,2,3,4,0,5,6,7,8]$$

Para definir la función sucesor podemos utilizar una función *inter* para intercambiar las fichas que haya en las casillas $a$ y $b$ de la configuración *fs*:

```
inter :: Casilla → Casilla → Config → Config
inter a b (C fs) =
  C [ s | (x, y) ← [(z, if z== a then b else
                         if z== b then a else z) | z ← [1..]],
        (y', s) ← zip [1..] fs,
        y' == y ]
```

que se ha descrito en términos de la función *zip* del PRELUDE para formar una lista de pares a partir de dos listas. Ahora consideramos una función *movs* que dada una casilla $c$ nos devuelva una lista con las casillas alcanzables desde $c$:

```
movs :: Casilla → [Casilla]
movs c =
  case c of 1 → [2,4]   ; 2 → [1,3,5]   ; 3 → [2,6]
            4 → [1,5,7]; 5 → [2,4,6,8]; 6 → [3,5,9]
            7 → [4,8]   ; 8 → [5,7,9]   ; 9 → [6,8]
```

con lo cual, usando *inter* y *movs* la declaración de *Config* como instancia de *Grafo* es trivial si sabemos localizar con *pos* (ver Ejercicio 6.43) la casilla en la que está el hueco:

```
instance Grafo Config where
  suc (C c) = [ inter x v c | x ← movs v ]
    where v          = pos 0 c
          pos x (y:ys) = if x== y then 1 else 1 + pos x ys
```

donde se observa que no hace falta ni quitar el propio $c$ ni quitar los repetidos. Finalmente, la función $puzzle8$, con argumentos una permutación de $[1..8]$ y el número de la casilla con el hueco inicial, resolverá el problema

```
puzzle8 :: [Ficha] → Casilla → [[Config]]
puzzle8 xs h =
-- xs ← permutaciones [1..8]
-- h ← [1..9]
  caminoDesde (C (hueco xs h)) (== C (hueco [1..8] 5)) []
    where hueco xs      1       = 0 : xs
          hueco (x : xs)(n + 1) = x : hueco xs n
```

donde la igualdad entre configuraciones es la igualdad estructural entre listas de enteros; si en lugar de todas las soluciones al problema se desea obtener una solución óptima (el camino más corto), en vez de llamar directamente a $puzzle8$ podemos usar $masCorta\ (puzzle8\ xs\ h)$.

**Solución al Ejercicio 13.6** (pág. 327).– El problema de los relojes de arena intenta determinar la forma de medir un tiempo de $t$ minutos con dos relojes de arena capaces de medir $tx$ y $ty$ minutos, las cuales definimos por comodidad como funciones constantes (para no ir arrastrándolas como argumento):

```
tx = 7
ty = 11
```

En primer lugar definiremos las estructuras de datos necesarias (ver Figura 21.11), que son los dos relojes (definidos mediante un constructor con dos argumentos (el tiempo arriba y el tiempo abajo) y no como pares para ahorrar paréntesis) y la configuración (en la cual es ***fundamental*** contemplar el tiempo, pues dos configuraciones en las que los relojes sean iguales pueden alcanzarse en distinto instante y deben considerarse configuraciones distintas):

```
data RelojX = X Int Int
data RelojY = Y Int Int
type Config = (RelojX, RelojY, Int)
```

Los dos únicos movimientos que podemos hacer son $vaciar$ y $girar$, que se definen mediante

```
vaciar (X x x', Y y y', t)
  | y≥ x && x > 0 = [ (X 0       tx,      Y (y − x) (y' + x), t + x),
                      (X 0       tx,      Y 0       ty ,      t + y) ]
  | x≥ y && y > 0 = [ (X (x − y)(x' + y), Y 0       ty ,      t + y),
                      (X 0       tx,      Y 0       ty ,      t + x) ]
  | otherwise     = []
```

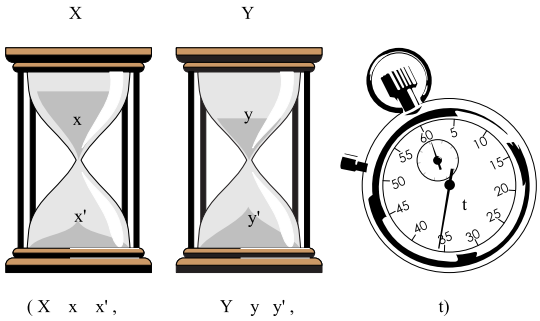


**Figura 21.11**: La configuración para el problema de los relojes de arena..

$$
\begin{array}{lcl}
girar\ (X\ 0\ x',\ Y\ y\ y',\ t) & = & [\ (X\ tx\ 0,\ \ Y\ y'\ y,\ t), \\
 & & \ \ (X\ tx\ 0,\ \ Y\ y\ \ y',t), \\
 & & \ \ (X\ 0\ \ x',\ Y\ y'\ y,\ t)\ ] \\
girar\ (X\ x\ x',\ Y\ 0\ y',\ t) & = & [\ (X\ x'\ x,\ \ Y\ ty\ 0,\ t), \\
 & & \ \ (X\ x\ \ x',\ Y\ ty\ 0,\ t), \\
 & & \ \ (X\ x'\ x,\ \ Y\ 0\ \ y',t)\ ] \\
girar\ \_ & = & [\,] \\
movs & = & [vaciar, girar]
\end{array}
$$

Finalmente, para la búsqueda de las soluciones podríamos pensar en instanciar desde la clase *Grafo*, pero como el grafo que resulta es infinito, la búsqueda puede divergir; interesa mejor instanciar directamente la función, limitándonos a los sucesores de un vértice que no superaron el tiempo permitido:

$$
\begin{array}{l}
suc\ c\ =\ [\ c'\mid m \leftarrow\ movs,\ c' \leftarrow\ m\ c\ ] \\
sol\ t\ \ =\ caminoDesde\ (X\ 0\ tx,\ \ Y\ 0\ ty,\ 0)\ test\ [\,]\ \textbf{where} \\
\quad test\ (\_,\_,t')\ =\ t{==}\ t' \\
\quad caminoDesde\ o\ te\ vis \\
\quad\quad \mid te\ o\ \ \ \ \ \ \ \ =\ [o : vis] \\
\quad\quad \mid otherwise\ =\ concat\ [\ caminoDesde\ o'\ te\ (o : vis)\ \mid \\
\quad\quad\quad\quad\quad\quad\quad\quad o'@(\_,\_,t')\ \leftarrow\ suc\ o,\ t'{\leq}\ t\ ]
\end{array}
$$

y si lo que queremos es buscar la solución mínima:

$$
\begin{array}{l}
minima\ [x]\ \ \ \ \ \ \ \ \ \ =\ x \\
minima\ (x : y : xs)\ =\ minima(m : xs) \\
\quad \textbf{where}\ m\ =\ \textbf{if}\ \ length\ x\ >\ length\ y\ \ \textbf{then}\ \ y\ \ \textbf{else}\ \ x
\end{array}
$$

$$
\begin{array}{l}
busca\ =\ [\ (t,\ minima\ ss)\mid t \leftarrow\ [14..],\ ss = sol\ t,\ noVacía\ ss\ ] \\
\quad \textbf{where}\ noVacía\ (x : \_)\ =\ True \\
\quad\quad\quad\ noVacía\ \_\ \ \ \ \ \ \ =\ False
\end{array}
$$

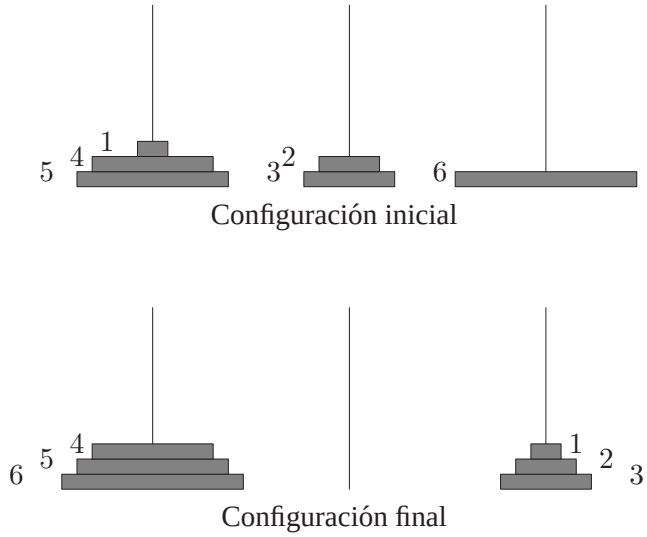


**Figura 21.12**: Las torres de Hanoi..

**Solución al Ejercicio 13.7** (pág. 338).– Para este problema una configuración se describe de la forma siguiente:

$$\mathbf{data}\ Torres\ =\ T\ [Int]\ [Int]\ [Int]\ \mathbf{deriving}\ (Show, Eq)$$

donde cada torre es una lista de enteros (los radios de sus discos), teniendo en cuenta que las listas aparecen en orden ascendente (discos con radios crecientes) y que la igualdad que usaremos es la estructural; por ejemplo, la configuración inicial mostrada en la Figura 21.12 se representaría mediante

$$T[1,4,5]\ [2,3]\ [6]$$

Si definimos un predicado *posible* para comprobar si es posible colocar un disco en una torre

$$\begin{array}{lcl} posible\ x\ [\,] & = & True \\ posible\ x\ (y:ys) & = & x \leq y \\ posible\ \_\ \_ & = & False \end{array}$$

el movimiento $m1$ del disco de la cima de la primera torre puede describirse mediante:

$$\begin{array}{l} m1\ (T\ (x:xs)\ ys\ zs)\ = \\ \quad [\ T\ xs\ (x:ys)\ zs \mid posible\ x\ ys\ ] +\!\!+ [\ T\ xs\ ys\ (x:zs) \mid posible\ x\ zs\ ] \\ m1\ (T\ [\,]\ \ \ \_\ \_)\ =\ [\,] \end{array}$$

donde se comprueba si el disco $x$ (que es el único que puede moverse para la primera torre) es posible colocarlo en las otras dos torres; análogamente se describen $m2$ y $m3$:

```
m2 (T xs (y : ys) zs)        =
  [ T (y : xs) ys zs | posible y xs ] ++ [ T xs ys (y : zs) | posible y zs ]
m2 (T _ [] _)                = []

m3 (T xs ys (z : zs)) =
  [ T xs (z : ys) zs | posible z ys ] ++ [ T (z : xs) ys zs | posible z zs ]
m3 (T _ _ [])                = []
```

Finalmente, los sucesores de una configuración se obtienen "juntando" todas las posibilidades:

```
instance Grafo Torres where
  suc t = [ t' | m ← [m1, m2, m3], t' ← m t ]

sol = caminoDesde (T [1, 4, 5] [2, 3] [6]) te []
  where te (T [4, 5, 6] [] [1, 2, 3]) = True
        te _                          = False
```

**Solución al Ejercicio 13.8** (pág. 339).– El caso base es trivial, mientras que el paso inductivo es

```
    pam x (f : fs)
≡≡  { 2)pam }
    f x : pam x fs
≡≡  { hipótesis de inducción }
    f x : map ($ x) fs
≡≡  { definición de ($) }
    (f $ x) : map ($ x) fs
≡≡  { 2) map }
    map ($ x) (f : fs)
```

**Solución al Ejercicio 13.9** (pág. 339).–

```
    movs (t : ts)
≡≡  { 2)movs }
    saltaDesde t ts ++ saltaHasta t ts ++ map (t :) (movs ts)
≡≡  { definición de concat }
    concat [saltaDesde t ts, saltaHasta t ts, map (t :) (movs ts)]
≡≡
    concat (map ($ ts) [saltaDesde t, saltaHasta t, map (t :).movs])
≡≡
    (concat.map ($ ts)) [saltaDesde t, saltaHasta t, map (t :).movs]
≡≡  { Ejercicio 13.8 }
    (concat.pam ts) [saltaDesde t, saltaHasta t, map (t :).movs]
≡≡  { definición de ($) }
```

$$concat.pam\ ts\ \$\ [saltaDesde\ t,\ saltaHasta\ t,\ map\ (t:).movs]$$

**Solución al Ejercicio 13.10** (pág. 339).– Siendo

$$lista \quad \equiv \quad [saltaDesde\ t, saltaHasta\ t,\ map\ (t:).movs]$$

podríamos expresar $movs$ mediante

$$movs\ (t:ts)\ =\ foldr\ (+\!\!+)\ [\,]\ (map\ (\$\ ts)lista)$$

Como además tenemos

$$map\ (\$\ ts)\ lista \quad \equiv \quad foldr\ (\lambda f\ fs\ \rightarrow\ f\ ts\ :\ fs)\ [\,]\ lista$$

con lo cual

$$movs\ (t:ts)\ =\ foldr\ (+\!\!+)\ [\,]\ (foldr\ (\lambda f\ fs\ \rightarrow f\ ts\ :\ fs)\ [\,]\ lista)$$

o de manera más eficiente con un solo *foldr*

$$movs\ (t:ts)\ =\ foldr\ (\lambda f\ fs\ \rightarrow\ f\ ts +\!\!+ fs)[\,]\ lista$$

**Solución al Ejercicio 13.11** (pág. 339).– Vamos a considerar las siguientes estructuras de datos para representar las fichas de dominó:

$$\begin{array}{l}\mathbf{type}\ Ficha\ =\ (Int, Int)\\ \mathbf{type}\ Fichas\ =\ [Ficha]\end{array}$$

Si consideramos la siguiente codificación de las casillas del cuadrado 4×4:

$$\begin{array}{cccc} 1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{array}$$

podemos describir las casillas que ocupa una ficha mediante un par, con lo cual una posible disposición de las fichas vendrá dada por una lista de posiciones, y una solución será un par formado por dos listas de igual longitud (la que define la disposición de las fichas y la de las propias fichas) tal que la relación entre sus elementos viene dada por la posición en la lista:

$$\begin{array}{l}\mathbf{type}\ Posic\ =\ (Int, Int)\\ \mathbf{type}\ Disp\ =\ [Posic]\\ \mathbf{type}\ Solu\ =\ (Disp, Fichas)\end{array}$$

El programa que vamos a escribir va a devolver una lista con las soluciones que haya para un determinado tamaño del dominó (siendo 6 el tamaño del dominó estándar), para

una cierta disposición de las fichas y para cierto valor de la constante *mágica* (la suma de cada línea del cuadrado mágico):

```
type TamDom = Int
type ConMag = Int

sol = map sol' gendisps where
  sol' (tdom, disp, cmags) = concat (map (solus tdom disp) cmags)

gendisps :: [(TamDom, Disp, [ConMag])]
...
```

donde *gendisps* se encargará de generar (la lista con) todas las ternas posibles; se ha elegido parametrizar el problema con respecto a la disposición y a una lista de constantes mágicas debido a los siguientes resultados teóricos bien conocidos (ver páginas 170–173 de [Gardner, 1983]):

- ✓ Se puede realizar un estudio de las disposiciones de las fichas que evite generar soluciones simétricas.
- ✓ No existen cuadrados mágicos de todos los tamaños posibles (con el dominó estándar sólo son posibles los de 4×4 y los de 6×6)
- ✓ No existen cuadrados mágicos para cualquier valor de las constantes mágicas posibles (con el dominó estándar, la constante mágica para 4×4 ha de estar entre 5 y 19 (ambos inclusive), mientras que para 6×6, entre 13 y 23 (ambos inclusive))

que podremos incorporar a la hora de generar las ternas a probar; por ejemplo, la terna:

```
( 6,
  [(1,2),(3,4),(5,6),(7,8),(9,10),(11,12),(13,14),(15,16)],
  [5..19] )
```

indica buscar con el dominó estándar los posibles cuadrados mágicos 4×4 en los cuales las ocho fichas de dominó que lo conforman estén dispuestas horizontalmente, para cada uno de los valores admisibles de la constante mágica. Ahora veamos cómo podemos describir nuestra función *solus*; la definición utilizará una cláusula **where** gigantesca para no ir arrastrando las constantes *tdom*, *cmag*, *fichas* (con todas las fichas del dominó considerado) y *dims* (con el número de casillas del lado del cuadrado mágico en cuestión):

```
solus :: TamDom → Disp → ConMag → [Solu]
solus tdom disp cmag = dominó [] [] disp
  where
    fichas    = [ (x,y) | x ← [0..tdom], y ← [x..tdom] ]
    dims      = sqrt (2 * length disp)
    dominó    ...
    esMágico  ...
```

donde la función *dominó* tiene como argumentos una lista $os$ de posiciones ocupadas, una lista $us$ de fichas colocadas y una lista $ls$ de posiciones libres:

$$
\begin{array}{l}
dominó\ os\ us\ [\,] \ =\ \textbf{if}\ \ (esMágico\ os\ us)\ \ \textbf{then}\ \ [(os, us)]\ \ \textbf{else}\ \ [\,] \\
dominó\ os\ us\ ls\ = \\
\quad concat\ [\ dominó\ ((c, c') : os)\ ((u, u') : us)\ ls'\ |\ ((c, c'), ls')\ \leftarrow\ pos\ ls, \\
\quad\ \ \textbf{where}\ \ldots \qquad\qquad\qquad\qquad\qquad\qquad\quad (u, u') \qquad \leftarrow\ fic\ us\ ]
\end{array}
$$

y la definición de *pos* y *fic* plasmará el grado de "inteligencia" del buscador de soluciones, desde el más simple:

$$
\begin{array}{l}
pos\ (l : ls)\ =\ (l, ls) \\
fic\ us \qquad =\ [\ (x, y)\ |\ (x, y) \leftarrow\ fichas,\ (x, y)\ `notElem`\ us\ ]
\end{array}
$$

hasta los que contemplen podas cuando por ejemplo alguna línea del cuadrado mágico no verifique que su suma coincida con la constante mágica. Nos falta definir el predicado *esMágico* que determine, cuando todas las posiciones libres estén ocupadas, si el cuadrado representado por $(os, us)$ es mágico, lo cual ocurrirá cuando todas sus líneas (filas, columnas y diagonales) sumen $cmag$:

$$
\begin{array}{l}
esMágico\ os\ us\ = \\
\quad compr\ filas\ \&\&\ compr\ colus\ \&\&\ compr\ diags\ \textbf{where} \\
\quad\ \ compr\ lins\ =\ and\ (map\ (\lambda\ lin \rightarrow\ sum\ lin == cmag)\ lins)
\end{array}
$$

y la generación de las líneas del cuadrado mágico es fácil si previamente lo construimos:

$$
\begin{array}{l}
cuad\ =\ construye\ os\ us\ (take\ (dims * dims)\ menosunos) \\
menosunos\ =\ (-1)\ :\ menosunos \\
construye\ [\,]\ [\,]\ c\ =\ c \\
construye\ ((o1, o2) : os)\ ((u1, u2) : us)\ = \\
\quad ponerEn\ o2\ u2\ .\ ponerEn\ o1\ u1\ \textbf{where} \\
\quad\ \ ponerEn\ 1 \qquad u\ (\_ : cs)\ =\ u : cs \\
\quad\ \ ponerEn\ (n + 1)\ u\ (c : cs)\ =\ c : ponerEn\ n\ u\ cs
\end{array}
$$

creando un cuadrado $dims\ \times\ dims$ en el que todas las casillas estén a un valor no utilizado (por ejemplo, $-1$) para después ir rellenando las dos casillas que determina cada posición $(o1, o2)$ para la ficha $(u1, u2)$, con lo cual la generación de las líneas del cuadrado mágico queda:

$$
\begin{array}{l}
(filas, colus, diags)\ = (filasDe\ cuad, \\
\qquad\qquad\qquad\qquad\ \ colusDe\ cuad, \\
\qquad\qquad\qquad\qquad\ \ diagsDe\ cuad)\ \textbf{where}\ \ldots
\end{array}
$$

donde para el cálculo de las filas aprovecharemos que sabemos que las dimensiones de $cuad$ serán $dims\ \times\ dims$:

$$
\begin{array}{l}
filasDe\ [\,]\ =\ [\,] \\
filasDe\ xs\ =\ f\ :\ filasDe\ rs\ \textbf{where}\ (f, rs)\ =\ splitAt\ dims\ xs
\end{array}
$$

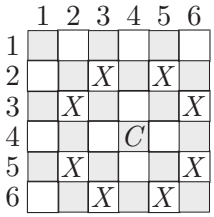


**Figura 21.13**: Salto de un caballo en un tablero $6 \times 6$..

para el de las columnas se mezclan de forma adecuada las filas (en este problema no es necesario obtener las columnas de abajo a arriba y por ello no hay que realizar una inversión previa de la lista con las filas):

$$
\begin{array}{l}
colusDe\ xs\ = \\
\quad colusDe'\ (filasDe\ xs)\ (map\ (\lambda\_ \rightarrow\ [\,])\ [1..dims])\ \mathbf{where} \\
\quad\ colusDe'\ [\,]\qquad cs\ =\ cs \\
\quad\ colusDe'\ (f : fs)\ cs\ =\ colusDe'\ fs\ (zipWith\ (\alpha\ b\ \rightarrow\ b : a)\ cs\ f)
\end{array}
$$

y para la diagonal principal se vuelve a aprovechar el conocimiento de las dimensiones de $cuad$, mientras que la secundaria se puede definir en términos de la principal si previamente invertimos cada una de las filas que aparecen en la lista con las filas:

$$
\begin{array}{l}
diagsDe\ xs\ =\ dpri\ xs\ :\ dsec\ xs\ :\ [\,]\ \mathbf{where} \\
\quad dpri\ [p]\ =\ p \\
\quad dpri\ xs\ =\ p : dpri\ rs\ \mathbf{where}\ (p : \_, rs)\ =\ splitAt\ (dims + 1)\ xs \\
\quad dsec\ xs\ =\ dpri\ (concat\ (map\ inv\ (filasDe\ xs)))
\end{array}
$$

$$
\begin{array}{l}
inv\ xs\ =\ inv2\ xs\ [\,]\ \mathbf{where} \\
\quad inv2\ [\,]\qquad ys\ =\ ys \\
\quad inv2\ (x : xs)\ ys\ =\ inv2\ xs\ (x : ys)
\end{array}
$$

Por último, plantear al lector la mejora de la solución propuesta en al menos tres aspectos:

- ✓ la descripción de la función $gendisps$,
- ✓ la descripción de funciones $pos$ y $fic$ más sofisticadas, para elegir la posición y la ficha a colocar de una forma más inteligente, y
- ✓ el estudio de otro tipo de dominós no estándar (de mayor tamaño) y de cuadrados mágicos de dimensión impar.

**Solución al Ejercicio 13.13** (pág. 342).– Si consideremos el tablero (ver Figura 21.13) como una rejilla rectangular de $nf$ filas y $nc$ columnas numeradas de forma que la esquina superior izquierda sea $(1, 1)$, podemos representar la casilla en la que está el caballo con un par de coordenadas (fila, columna)

$$\textbf{type}\ Casilla\ =\ (Int, Int)$$

con lo cual el caballo $C$ mostrado en la Figura 21.13 está en la casilla $(4, 4)$ y puede saltar a las casillas marcadas con una $x$; así, para definir la lista de sucesores de un nodo consideramos una función $desps$ constante con los posibles desplazamientos de un caballo:

$$\begin{array}{l}
\textbf{type}\ Desplaz\ =\ (Int, Int) \\
\\
desps\ ::\ [Desplaz] \\
desps\ =\ [\,(-2,-1),(-2,\ 1),(\ 2,-1),(\ 2,\ 1),(\ 1,-2),(\ 1,\ 2),(-1,-2),(-1,\ 2)\,] \\
\\
suc\,(x,y)\ =\ [\,(x',y')\mid (dx,dy)\ \leftarrow\ desps, \\
\qquad\qquad\qquad\qquad x'\ =\ x+dx,\ 1\leq x',\ x'\leq nf, \\
\qquad\qquad\qquad\qquad y'\ =\ y+dy,\ 1\leq y',\ y'\leq nc\,]
\end{array}$$

donde obsérvese que no instanciamos desde la clase *Grafo* para *Casilla* ya que el test de encontrado es desconocido, ya que sabemos que la solución será una lista de casillas de longitud $nf * nc$ pero no sabemos en qué nodo va a terminar. Por ello, la búsqueda de soluciones se puede hacer escribiendo directamente la función de búsqueda, y como la condición de final depende de la longitud de los vértices ya recorridos, habrá que pasarle a *caminoDesde* un contador:

$$\begin{array}{l}
caballo\ ::\ Casilla\ \rightarrow\ [Casilla] \\
caballo\,(x,y)\ =\ head\,(caminoDesde\,(x,y)\,[\,]\,0) \\
\\
caminoDesde\ ::\ a\ \rightarrow\ [a]\ \rightarrow\ Int \\
caminoDesde\ o\ vis\ con \\
\quad \mid con == nf * nc - 1\ =\ [o : vis] \\
\quad \mid otherwise\ \ \ \ \ \ \ \ \ \ =\ concat\,[\,caminoDesde\ o'\,(o : vis)\,(con+1)\mid \\
\qquad\qquad\qquad\qquad\qquad\qquad o'\ \leftarrow\ suc\ o,\ o'\ `notElem`\ vis\,]
\end{array}$$

donde hemos supuesto en todo momento que las dimensiones del tablero vienen dadas por funciones constantes,

$$\begin{array}{l}
nf\ =\ 6 \text{ -- número de filas} \\
nc\ =\ 6 \text{ -- número de columnas}
\end{array}$$

aunque es fácil parametrizar el problema con respecto al par $(nf, nc)$:

```
type Casilla  = (Int, Int)
type Config   = (Int, Int)
type Desplaz  = (Int, Int)
type Dimens   = (Int, Int)

desps :: [Desplaz]
desps = [ (-2,-1),(-2, 1),( 2,-1),( 2, 1),( 1,-2),
          ( 1, 2),(-1,-2),(-1, 2) ]


caballo :: Dimens -> Casilla -> [Casilla]
caballo (nf, nc) (x, y) = head (caminoDesde (x, y) [] 0) where
  caminoDesde o vis con
    | con== nf * nc - 1 = [o : vis]
    | otherwise         = concat [ caminoDesde o' (o : vis) (con + 1) |
                                   o' <- suc o, o' `notElem` vis ]
  suc (x, y) = [ (x', y', nf, nc) | (dx, dy) <- desps,
                                    x' = x + dx, 1 <= x', x' <= nf,
                                    y' = y + dy, 1 <= y', y' <= nc ]
```

**Solución al Ejercicio 13.14** (pág. 342).– Considerando la siguiente codificación de las casillas del tablero

$$\begin{array}{ccc} 0 & 3 & 6 \\ 5 & & 1 \\ 2 & 7 & 4 \end{array}$$

y describiendo las posiciones de los caballos con una lista de enteros

$$T\,[b,\ b',\ n,\ n']$$

donde $b$ y $b'$ (resp. $n$ y $n'$) representan las posiciones de los caballos blancos (resp. negros), tendremos la siguiente igualdad entre configuraciones

```
data Tablero = T [Int] deriving Show
instance Eq Tablero where
  T [b, b', n, n'] == T [b1, b1', n1, n1'] =
    ([b, b']== [b1, b1']  || [b, b']== [b1', b1]) &&
    ([n, n']== [n1, n1'] || [n, n']== [n1', n1])
```

Un salto desde $b$ viene dado por

$$salto\ b\ =\ [(b+1)\ `mod`\ 8,\ (b+7)\ `mod`\ 8]$$

y los movimientos vienen dados por

```
mueve 1 (T [b, b', n, n']) =
  [ T [b'', b', n, n'] | b'' ← salto b, b'' `notElem` [b', n, n'] ]

mueve 2 ...
```

Si recordamos la interfaz de la clase *Grafo*

```
class Eq a ⇒ Grafo a where
  suc          :: a → [a]
  caminoDesde :: a → (a → Bool) → [a] → [[a]]
  ...
```

basta instanciar antes la clase *Tablero*

```
instance Grafo Tablero where
  suc tabl = concat [ mueve i tabl | i ← [1..4] ]
```

y la búsqueda se realiza con

```
sol = caminoDesde (T [1,3,7,5]) (== T [7,5,1,3]) []
```

**Solución al Ejercicio 13.15** (pág. 342).– Para resolver el problema de los trenes con tres vías las estructuras de datos que usaremos (con igualdad estructural) son:

```
data Pieza  = L | A | B deriving (Enum, Eq, Show)
type Tren   = [Pieza]
data Config = C Tren Tren Tren deriving (Eq, Show)
-- Representa (IzdaPuente,DchaPuente,VíaEntrada), donde
-- IzdaPuente → en sentido contrario a las agujas del reloj
-- DchaPuente → en sentido contrario a las agujas del reloj
-- VíaEntrada → de izquierda a derecha
```

donde las componentes de *Config* denotan el tren (con sus piezas ordenadas en sentido antihorario) que en cierto instante está en la vía muerta a la izquierda del puente, el tren (con sus piezas también ordenadas en sentido antihorario) que está en la vía muerta a la derecha del puente y el tren (con sus piezas ordenadas de izquierda a derecha) que está en la vía de entrada a la vía muerta. Los posibles movimientos son tres parejas de operaciones "duales" que vamos a describir mediante seis funciones que toman una configuración y devuelven una lista de configuraciones; dichas funciones son:

- ✓ *puenteDcha*, para ir desde la vía muerta a la izquierda del puente hacia la vía muerta a la derecha del puente, atravesando el puente
- ✓ *puenteIzda*, para ir desde la vía muerta a la derecha del puente hacia la vía muerta a la izquierda del puente, atravesando el puente
- ✓ *salirVía*, para ir desde la vía muerta a la izquierda del puente hacia la vía de entrada a la vía muerta

- ✓ $entrarVía$, para ir desde la vía de entrada a la vía muerta hacia la vía muerta a la izquierda del puente

- ✓ $girarDcha$, para ir desde la vía muerta a la izquierda del puente hacia la vía muerta a la derecha del puente, sin atravesar el puente

- ✓ $girarIzda$, para ir desde la vía muerta a la derecha del puente hacia la vía muerta a la izquierda del puente, sin atravesar el puente

En las dos primeras operaciones hay que tener en cuenta que sólo puede pasar el puente la locomotora; la operación más simple es $puenteDcha$:

$$\begin{array}{lcl} puenteDcha\ (C\ (L:x)\ y\ z) & = & [C\ x\ (y +\!\!+\ [L])\ z] \\ puenteDcha\ \_ & = & [\,] \end{array}$$

y aunque $puenteIzda$ podría describirse mediante:

$$\begin{array}{l} puenteIzda\ (C\ \_\ [\,]\ \_) = [\,] \\ puenteIzda\ (C\ x\ y\ z) = \\ \quad [\ C\ (L:x)\ y'\ z \mid (y', v) = splitAt\ ((length\ y) - 1)\ y,\ v == [L]\ ] \end{array}$$

es más eficiente hacerlo en términos de una función *último*:

$$\begin{array}{lcl} puenteIzda\ (C\ x\ y\ z) & = & [\ C\ (L:x)\ r\ z \mid r \leftarrow último\ y\ ] \\ último\ [\,] & = & [\,] \text{ -- si no hay nada, no } puenteIzda \\ último\ [L] & = & [[\,]] \text{ -- si sólo hay } L\text{, queda vacío } DchaPuente \\ último\ [\_] & = & [\,] \text{ -- si hay algo } \neq L\text{, no } puenteIzda \\ último\ (p:t) & = & [\ p:r \mid r \leftarrow último\ t\ ] \end{array}$$

Las dos últimas parejas de funciones se definen en base a una operación $partir$, que devuelve la lista de pares de listas en que puede partirse cierta lista argumento (o sea, si $(u, v) \leftarrow partir\ t$, entonces $u +\!\!+ v == t$):

$$\begin{array}{lcl} partir\ [\,] & = & [([\,], [\,])] \\ partir\ (p:t) & = & ([\,], p:t) : [\ (p:u, v) \mid (u, v) \leftarrow partir\ t\ ] \end{array}$$

con lo cual para describir una operación que vaya de izquierda a derecha (de derecha a izquierda) habrá que partir el tren de la izquierda (derecha) de forma que la locomotora aparezca en el lado derecho (izquierdo):

$$\begin{array}{lcl} salirVía\ \ (C\ x\ y\ z) & = & [\ C\ u\ y\ (v +\!\!+\ z) \mid (u, v) \leftarrow partir\ x,\ L\ `elem`\ v\ ] \\ entrarVía\ (C\ x\ y\ z) & = & [\ C\ (x +\!\!+\ u)\ y\ v \mid (u, v) \leftarrow partir\ z,\ L\ `elem`\ u\ ] \end{array}$$

$$\begin{array}{lcl} girarDcha\ (x, y, z) & = & [\ C\ u\ (v +\!\!+\ y)\ z \mid (u, v) \leftarrow partir\ x,\ L\ `elem`\ v\ ] \\ girarIzda\ \ (x, y, z) & = & [\ C\ (x +\!\!+\ u)\ v\ z \mid (u, v) \leftarrow partir\ y,\ L\ `elem`\ u\ ] \end{array}$$

Finalmente, la búsqueda de las soluciones puede hacerse instanciando directamente desde la clase *Grafo* para búsqueda en profundidad:

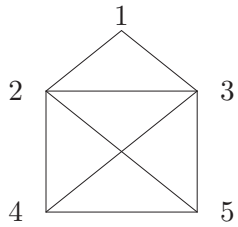


**Figura 21.14**: Un polígono..

```
movs = [ puenteDcha, puenteIzda, girarDcha, girarIzda,
         salirVía, entrarVía]

instance Grafo Config where
  suc c = [ c' | m ← movs, c' ← m c ]

can = caminoDesde (C [A] [B] [L]) test [ ]
  where test (C [B] [A] [L]) = True
        test _               = False
```

Además, se puede imponer un límite de movimientos a la solución mediante

```
corta = head [ c | c ← can, length c ≤ 21 ]
```

y de hecho se obtiene una solución en 20 pasos (i.e., con 21 estados).

**Solución al Ejercicio 13.16** (pág. 343).– Una vez numerados los vértices como se muestra en la Figura 21.14, la dificultad de este problema estriba en que hay que determinar desde qué vértice hay que empezar a pintar. Cada una de las configuraciones del espacio de configuraciones va a ser un par de enteros denotando dónde está el lápiz y hacia qué vértice va; además, en este caso la igualdad entre configuraciones no debe ser la estructural, pues hay que contemplar el hecho de que no se pueda pasar dos veces por el mismo arco:

```
data Config = C Int Int deriving Show
instance Eq Config where
  (C x y) == (C x' y') = x== x' && y== y' || x== y' && y== x'

vertices = [1,2,3,4,5]
arcos    = [(1,2),(1,3),(2,4),(2,3),(2,5),(3,4),(3,5),(4,5)]

na  = length arcos
na1 = na - 1
```

Obsérvese que no hay que definir arcos en los dos sentidos, y que se han definido funciones constantes por comodidad. La búsqueda de soluciones se debe hacer definiendo directamente la función de búsqueda, ya que la condición de final depende de la longitud

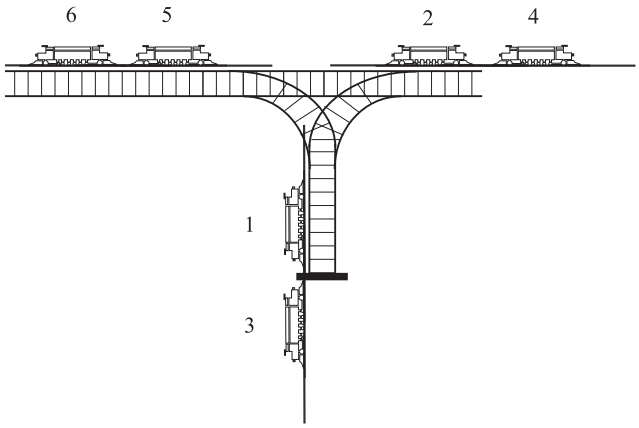


**Figura 21.15**: El estado $T[5..n][1,3][2,4]$..

de la lista de arcos ya recorridos (si no aparecen arcos repetidos en tal lista), con lo cual podemos pasarle a *caminoDesde* un contador para no estar calculando la longitud de la lista *vis* continuamente:

$$
\begin{array}{l}
suc\ (C\ x\ y)\ = \\
\quad [\ C\ y\ z\ |\ (a,b)\ \leftarrow\ arcos, \\
\qquad\quad z\ \ \leftarrow\ [\ b\ |\ a == y, x \neq\ b\ ] +\!\!+\ [\ a\ |\ b == y, a \neq\ x\ ]\ ] \\
dibujo\ =\ head\ [\ s\ |\ (x,y)\ \leftarrow\ arcos,\ s\ \leftarrow\ caminoDesde\ (C\ x\ y)\ [\,]\ 0\ ]
\end{array}
$$

$$
\begin{array}{l}
caminoDesde\ o\ vis\ n \\
\quad |\ n == na1\ \ =\ [o : vis] \\
\quad |\ otherwise\ \ =\ concat\ [\ caminoDesde\ o'\ (o : vis)\ (n+1)\ | \\
\qquad\qquad\qquad\qquad o'\ \leftarrow\ suc\ o,\ o'\ `notElem`\ vis\ ]
\end{array}
$$

Obsérvese que tomaremos en la función *dibujo* la primera solución encontrada, porque todas son "igual de buenas".

**Solución al Ejercicio 13.17** (pág. 343).– Tres listas permiten describir cada configuración:

$$\mathbf{data}\ Trenes\ =\ T\ [Int]\ [Int]\ [Int]\ \mathbf{deriving}\ (Eq, Show)$$

Así, por ejemplo

$$T\ [5..n]\ [1,3]\ [2,4]$$

representa el estado que se muestra en la Figura 21.15 (la primera lista aparece invertida). Así, partiendo de la configuración inicial $T[1..n]\ [\,]\ [\,]$, representaremos el grafo de configuraciones considerando tres posibles movimientos:

$$
\begin{array}{ll}
direc & \equiv \langle \text{trasladar la cabeza de la primera lista a la tercera} \rangle \\
aPila & \equiv \langle \text{idem de la primera a la segunda} \rangle
\end{array}
$$

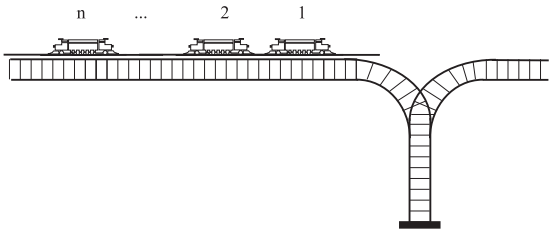


**Figura 21.16**: La vía eliminando `direc`..

$$dPila \equiv \langle \text{idem de la segunda a la tercera} \rangle$$

Por ejemplo,

$$\begin{array}{lcl} direc\ (\ T\ [5..n]\ [1,3]\ [2,4]\ ) & = & T\ [6..n]\ [1,3]\ [5,2,4] \\ aPila\ (\ T\ [5..n]\ [1,3]\ [2,4]\ ) & = & T\ [6..n]\ [5,1,3]\ [2,4] \\ dPila\ (\ T\ [5..n]\ [1,3]\ [2,4]\ ) & = & T\ [5..n]\ [3]\ [1,2,4] \end{array}$$

Tales movimientos describen un grafo dirigido sin ciclos; el problema que nos planteamos es contar las configuraciones finales de la forma $T\ [\,]\ [\,]\ s$ que son alcanzables desde el nodo inicial $T\ [1..n]\ [\,]\ [\,]$. Podemos escribir un programa en HASKELL directamente si podemos describir la función sucesor de la clase *Grafo*; veamos en primer lugar las funciones *direc*, *aPila* y *dPila*

$$\begin{array}{l} direc,\ aPila,\ dPila\ ::\ Trenes \rightarrow Trenes \\ direc\ (T\ (a:as)\ bs\ cs) = T\ as\ bs\ (a:cs) \\ aPila\ (T\ (a:as)\ bs\ cs) = T\ as\ (a:bs)\ cs \\ dPila\ (T\ as\ (b:bs)\ cs) = T\ as\ bs\ (b:cs) \end{array}$$

donde se observa que *direc* es la composición $dPila\ .aPila$, con lo cual puede ser eliminada y la figura queda como se muestra en la Figura 21.16 (que es el planteamiento inicial propuesto en la página 236 de [Knuth, 1968]). Ahora podemos dar una instancia de la clase *Grafo* para *Trenes*:

$$\begin{array}{l} \textbf{instance}\ Grafo\ Trenes\ \textbf{where} \\ \quad suc\ c = [\ c' \mid cs \leftarrow (aPila\ c +\!\!+ dPila\ c),\ c' \leftarrow cs\ ] \\ \quad\ \textbf{where}\ aPila\ (T\ [\,]\ \_\ \_) = [\,] \\ \qquad\qquad aPila\ (T\ (a:as)\ bs\ cs) = [T\ as\ (a:bs)\ cs] \\ \qquad\qquad dPila\ (T\ \_\ [\,]\ \_) = [\,] \\ \qquad\qquad dPila\ (T\ as\ (b:bs)\ cs) = [T\ as\ bs\ (b:cs)] \end{array}$$

en la cual se ha modificado el tipo del valor devuelto por *aPila* y *dPila* para simplificar la notación; de esta forma, la solución vendría dada por:

$$\begin{array}{l} sol\ n = caminoDesde\ (T\ [1..n]\ [\,]\ [\,])\ test\ [\,] \\ \qquad \textbf{where}\ test\ (T\ [\,]\ [\,]\ \_) = True \\ \qquad\qquad test\ \_ = False \end{array}$$

y la siguiente función resuelve el problema de contar el número de movimientos para los distintos valores de $n$

$$pr \;=\; [\; length\;(sol\;n) \mid n \;\leftarrow\; [1..]\;]\;]$$

resultando los valores

$$[\,1,\;2,\;5,\;14,\;42,\;132,\;429, 1430,\;\ldots\,]$$

que conforman la denominada *sucesión de Catalán*, muy frecuente en problemas combinatorios.

### LOS NÚMEROS DE CATALÁN GENERALIZADOS

El siguiente apartado es un resumen de la solución dada en [Muñoz y Ruiz, 1995]. Sea $a_{n,k}$ el número de nodos finales (formas posibles de extraer las locomotoras) desde un nodo con $n$ elementos en la primera lista y $k$ elementos en la segunda; de aquí es fácil obtener la recurrencia

$$\begin{array}{lll} (B1) & a_{n,k} \;= a_{n-1,k+1} + a_{n,k-1}, & n,k \;\geq\; 1 \\ (B2) & a_{0,i} \;= 1, & i \geq\; 1 \end{array}$$

El primer sumando del segundo miembro de $(B1)$ es el número de nodos resultantes después de realizar la operación $aPila$ mientras que el segundo corresponde a después de realizar $dPila$. Es fácil obtener de aquí la expresión

$$(B3) \qquad a_{n,k} \;= \textstyle\sum_{1\leq\, i\leq\, k+1} a_{n-1,i}$$

que junto a los valores iniciales, permite resolver la recurrencia $(B1)$–$(B2)$. En efecto, calculemos sucesivamente los valores $a_{1,k}, a_{2,k}, \ldots$; para $n = 1$ encontramos (ya que $a_{0,i} = 1$)

$$a_{1,k} = k + 1$$

y de aquí,

$$\begin{array}{l} a_{2,k} \\ \equiv\!\equiv\; \{\,\text{ya que } \textstyle\sum_{1\leq i\leq n} i = n(n+1)/2\,\} \\ (k+1)(k+4)/2 \end{array}$$

Además

$$\begin{array}{l} a_{3,k} \\ \equiv\!\equiv\; \{\,\text{ya que } \textstyle\sum_{1\leq i\leq n} i^2 = n(n+1)(2n+1)/6\,\} \\ (k+1)(k+5)(k+6)/3! \end{array}$$

Veamos ahora los valores para $n = 4$

$$\begin{array}{l} a_{4,k} \\ \equiv\!\equiv\; \{\,\text{ya que } \textstyle\sum_{1\leq i\leq n} i^3 = n^2(n+1)^2/4\,\} \\ (k+1)(k+6)(k+7)(k+8)/4! \end{array}$$

En resumen, hemos obtenido para los primeros valores

$$\begin{array}{rcl} a_{1,k} & = & k+1 \\ a_{2,k} & = & (k+1)(k+4)/2 \\ a_{3,k} & = & (k+1)(k+5)(k+6)/3! \\ a_{4,k} & = & (k+1)(k+6)(k+7)(k+8)/4! \end{array}$$

y podemos conjeturar la forma de la solución

$$(B4) \qquad a_{n,k} = \frac{k+1}{n}\left(\begin{array}{c} 2n+k \\ n-1 \end{array}\right)$$

Es fácil ver que la solución de la recurrencia $(B1)$–$(B2)$ viene dada por $(B4)$ por inducción sobre $n$; para $n=1$ es trivial, mientras el paso inductivo es

$$\begin{array}{l} a_{n+1,k} \\ \equiv \quad \{\, (B3) \,\} \\ \sum_{1\le i\le k+1} a_{n,i} \\ \equiv \quad \{\, \text{hipótesis de inducción} \,\} \\ \displaystyle\sum_{1\le i\le k+1} \frac{i+1}{n}\left(\begin{array}{c} 2n+i \\ n-1 \end{array}\right) \\ \equiv \quad \{\, \text{ver siguiente identidad} \,\} \\ \displaystyle\frac{k+1}{n+1}\left(\begin{array}{c} 2n+k+2 \\ n \end{array}\right) \end{array}$$

y queda probar la identidad

$$\sum_{1\le i\le k+1} \frac{i+1}{n}\left(\begin{array}{c} 2n+i \\ n-1 \end{array}\right) = \frac{k+1}{n+1}\left(\begin{array}{c} 2n+k+2 \\ n \end{array}\right)$$

que de nuevo se prueba fácilmente por inducción, pero esta vez sobre $k$. En particular, para $k=0$ obtenemos

$$a_{n,0} = \frac{1}{n}\left(\begin{array}{c} 2n \\ n-1 \end{array}\right) = \frac{1}{n+1}\left(\begin{array}{c} 2n \\ n \end{array}\right)$$

conocidos como *números de Catalán*, que corresponde a la solución del problema propuesto; utilizando la fórmula de Stirling se obtiene la siguiente aproximación, para valores grandes de $n$

$$a_{n,0} \simeq \frac{4^n}{\pi n\sqrt{n}}$$

Para valores de $k>0$ obtenemos los *números de Catalán generalizados* dados por $(B4)$, que conforman la solución al problema generalizado.

## 21.10. ANALIZADORES

**Solución al Ejercicio 14.1** (pág. 355).–

$$\begin{array}{ll} rString\ [\,] & = \acute{e}xito\ "" \\ rString(c : cs) & = rChar\ c >\!\!>\!\!> \lambda\_ \rightarrow \\ & \quad rString\ cs >\!\!>\!\!> \lambda\ rs \rightarrow \\ & \quad \acute{e}xito\ (c : rs) \end{array}$$

**Solución al Ejercicio 14.2** (pág. 357).– Este analizador no contempla el tratamiento de los espacios en blanco.

**Solución al Ejercicio 14.4** (pág. 366).– En primer lugar se observa que:

$$\begin{array}{ll} & aplica\ (iter\ m)\ y \\ \equiv & \{\text{ definición de } iter\ \} \\ & aplica\ (\mathbf{do}\ a \leftarrow\ m;\ x \leftarrow\ iter\ m;\ return\ (a : x)\ !+\ return\ [\,])\ y \\ \equiv & \{\text{ definición de } (!+) \text{ y de } aplica\ \} \\ & aplica\ (\mathbf{do}\ a \leftarrow\ m;\ x \leftarrow\ iter\ m;\ return\ (a : x))\ y +\!\!+ aplica\ (return\ [\,])\ y \\ \equiv & \\ & aplica\ (\mathbf{do}\ a \leftarrow\ m;\ x \leftarrow\ iter\ m;\ return\ (a : x))\ y +\!\!+ [([\,], y)] \end{array}$$

con lo cual faltaría probar:

$$\begin{array}{ll} & aplica\ (\mathbf{do}\ a \leftarrow\ m;\ x \leftarrow\ iter\ m;\ return\ (a : x))\ y \\ \equiv & \{\text{ notación } \mathbf{do}\text{, definición de } aplica\ \} \\ & aplica\ (m\ >\!\!>\!\!=\ \\ & \lambda\ a\ \rightarrow\ iter\ m\ >\!\!>\!\!=\ \\ & \lambda\ x\ \rightarrow\ return\ (a : x))\ y \\ \equiv & \{\text{ def. } >\!\!>\!\!=, k \equiv \lambda a \rightarrow iter\ m >\!\!>\!\!= \lambda x \rightarrow return(a : x)\ \} \\ & \mathbf{do}\ (a, y') \leftarrow\ aplica\ m\ y;\ (b, z) \leftarrow\ k\ a\ y';\ return\ (b, z) \\ \equiv & \{\ (*)\ \text{(ver después)}\ \} \\ & \mathbf{do}\ (a, y') \leftarrow\ aplica\ m\ y; \\ & \quad (b, z) \leftarrow\ \mathbf{do}\{(a', y'') \leftarrow\ aplica\ (iter\ m)\ y';\ return\ (a : a', y'')\}; \\ & \quad return\ (b, z) \\ \equiv & \\ & \mathbf{do}\ (a, y') \leftarrow\ aplica\ m\ y;\ (a', y'') \leftarrow\ aplica\ (iter\ m)\ y';\ return\ (a : a', y'') \end{array}$$

donde $(*)$ también es fácil:

$$\begin{array}{ll} & k\ a\ y' \\ \equiv & \\ & (\ \lambda\ a\ \rightarrow\ iter\ m\ >\!\!>\!\!=\ \lambda\ x\ \rightarrow\ return\ (a : x)\ )\ a\ y' \\ \equiv & \{\ \beta\text{–regla}\ \} \\ & (\ iter\ m\ >\!\!>\!\!=\ \lambda\ x\ \rightarrow\ return\ (a : x)\ )\ y' \\ \equiv & \{\text{ definición de } >\!\!>\!\!=\ \} \end{array}$$

```
    do {(a', y'') ← aplica (iter m) y'; (b', z') ← (λ x → return (a : x) a') y'';
        return (b', z') }
≡≡  { β–regla }
    do {(a', y'') ← aplica (iter m) y'; (b', z') ← return (a : a') y'';
        return (b', z') }
≡≡  { definición de return }
    do { (a', y'') ← aplica (iter m) y'; (b', z') ← return (a : a', y'');
        return (b', z') }
≡≡
    do (a', y'') ← aplica (iter m) y'; return (a : a', y'')
```

**Solución al Ejercicio 14.5** (pág. 366).–

```
type Idv  = String
type Idc  = String
data Ter v = Var v | Con Idc deriving Show

anaIden :: Analiz String
anaIden = AN (return . (span (λu → isAlphaNum u || u=='_')))

anaVar, anaCon, anaTer :: Analiz (Ter Idv)
anaVar = do c ← elemento !> isUpper; r ← anaIden; return (Var (c : r))
anaCon = do c ← elemento !> isLower; r ← anaIden; return (Con (c : r))
anaTer = anaVar !+ anaCon

data Obj = O String [Ter Idv] deriving Show

anaObj :: Analiz Obj
anaObj = do
           Con p ← anaCon
           _ ← literal '('
           a ← anaTer
           as ← anaArg
           return (O p (a : as))

anaArg :: Analiz[Ter Idv]
anaArg =   do _ ← literal')'; return []
       !+ do _ ← literal ','; t ← anaTer; r ← anaArg; return (t : r)

data Regla = Obj :- [Obj] deriving Show

anaReg :: Analiz Regla
anaReg =   do oc ← anaObj; _ ← literal '.'; return (oc :- [])
       !+ do oc ← anaObj; _ ← literal ':'; _ ← literal '-';
             o ← anaObj; os ← anaObs; return (oc :- (o : os))

anaObs :: Analiz [Obj]
anaObs =   do literal '.'; return []
       !+ do literal ','; o ← anaObj; os ← anaObs; return (o : os)
```

```
vacía :: Analiz ()
vacía = AN (λ ent → if (null ent) then return ((), []) else [])

anaRegs :: Analiz [Regla]
anaRegs =  do () ← vacía; return []
        !+ do r ← anaReg; rs ← anaRegs; return (r : rs)

analiz :: String → [Regla]
analiz = fst . head .aplica anaRegs . filter (not . isSpace)

rs = analiz rss
```

**Solución al Ejercicio 14.6** (pág. 367).– Consideremos el tipo

```
data Term = Var String | Lam String Term | App Term Term
          deriving Show
```

y supongamos que un identificador tiene el primer carácter en minúsculas y el resto de caracteres alfanuméricos:

```
identif :: Analiz String
identif = do
            x ← elemento !> isLower
            xs ← iter (elemento !> isAlphaNum)
            return (x : xs)
```

donde tanto *elemento* como *iter* se definieron en el texto. Entonces se tiene

```
term  = atom `chainl1` (return App)
atom  = var !* lam !* entre
var   = do { v ← identif; return (Var v) }
lam   = do { literal 'λ'; x ← identif; literal '.'; t ← term; return (Lam x t) }
entre = paren (literal '(') term (literal ')')
```

siendo ‘chainl1‘ un analizador de secuencias que permite resolver la recursión por la izquierda en la definición de los $\lambda$-términos:

```
chainl1 :: Analiz a → Analiz (a → a → a) → Analiz a
p `chainl1` op = do
                   x ← p
                   s ← iter (do{f ← op; y ← p; return (f, y)})
                   return (foldl(λ a (f, y) → f a y) x s)
```

**Solución al Ejercicio 14.7** (pág. 367).– Con el tipo *Term* del Ejercicio 14.6 y el transformador de estados definido en el Capítulo 11, el renombrador de $\lambda$-expresiones queda

```
rename e = runTE (renamer e) 0
```

```
renamer (Var x) = return (Var x)
renamer (Lam x t)   = do
                        nx ← newName
                        r ← renamer t
                        return (Lam nx (subst x nx r))
renamer (App e1 e2) = do
                        r ← renamer e1
                        s ← renamer e2
                        return (App r s)
```

donde la función *newName* se encarga de "inventar" (con el transformador de estados) un nombre nuevo a partir del último inventado:

```
newName = do
            n ← newVar
            return (mkName n)
newVar    = T (λ e → (e + 1, e))
mkName n = "x"++ show n
```

Finalmente, la función *subst* es la que realiza el reemplazamiento de la variable antigua por la nueva variable:

```
subst x nx (Var y)
  | x== y           = Var nx
  | otherwise       = Var y
subst x nx (Lam y t)
  | x== y           = Lam nx z
  | otherwise       = Lam y z
                      where z = subst x nx t
subst x nx (App e1 e2) = App (subst x nx e1) (subst x nx e2)
```

## 21.11. SIMULACIÓN

**Solución al Ejercicio 15.1** (pág. 372).–

```
buscaProyección (a', b') (a, b) = pintasecciones(secciona vs)
  where vs = [ (t, transformaEntreIntervalos (a', b') (a, b) t) | t ← [a'..b']]
        secciona [] = []
        secciona [x] = [x]
        secciona ((u', u) : (v', v) : xs)
          | u== v     = secciona ((u', v) : xs)
          | otherwise = (u', u) : secciona ((v', v) : xs)
```

$$
\begin{aligned}
&pintasecciones\ (\ (t, v) : (t', v') : xs)\ = \\
&\quad putStrLn\ (\text{"En "} +\!\!+\ show\ [t, t'-1] +\!\!+ \text{" vale "} +\!\!+\ show\ v\ +\!\!+ \\
&\qquad\qquad \text{". Total: "} +\!\!+\ show\ (t'-t) +\!\!+ \text{" valores."})\ >> \\
&\quad pintasecciones\ ((t', v') : xs) \\
&pintasecciones\ [(t, v)]\ = \\
&\quad putStrLn\ (\text{"En "} +\!\!+\ show\ [t, b'] +\!\!+ \text{" vale "} +\!\!+\ show\ v +\!\!+ \\
&\qquad\qquad \text{". Total: "} +\!\!+\ show\ (b'-t+1) +\!\!+ \text{" valores."})
\end{aligned}
$$

**Solución al Ejercicio 15.2** (pág. 386).– Se podría pensar en una estructura como

$$
\begin{aligned}
&\mathbf{data}\ Complem\ =\ C\{sinComp, conComp\ ::\ Int\} \\
&\mathbf{data}\ Resultado\ =\ T\{deCero, \ldots, deCuatro, deCinco, deSeis\ ::\ Complem\}
\end{aligned}
$$

de forma que tanto *ResultadoSimulaciónPrimitiva* como *ResultadoEscrutinio* fueran de tipo *Resultado*

$$
\begin{aligned}
&\mathbf{type}\ ResultadoSimulaciónPrimitiva\ =\ Resultado \\
&\mathbf{type}\ ResultadoEscrutinio\ =\ Resultado
\end{aligned}
$$

con la idea de que el resultado de un escrutinio fuera una estructura del tipo anterior con todos los campos nulos salvo uno de ellos que estaría a uno. Así, una vez sobrecargado el operador suma se tendría

$$actualizaTotal\ t\ c\ =\ t\ +\ c$$

**Solución al Ejercicio 15.3** (pág. 388).– Dado que el reintegro es un número entre 0 y 9 asignado a una apuesta de forma independiente del resto de las bolas pronosticadas podemos considerar

$$
\begin{aligned}
&\mathbf{type}\ Reintegro\ =\ Int \\
&\mathbf{type}\ Apuesta\ =\ ([Bola], Reintegro) \\
&\mathbf{type}\ Sorteo\ =\ (Primitiva, Complementario, Reintegro)
\end{aligned}
$$

pero no podemos contemplar el acierto del reintegro como un valor lógico adicional en el resultado del escrutinio:

$$
\begin{aligned}
\mathbf{data}\ ResultadoEscrutinio\ =\ &SinCompSinRein\ Int \mid SinCompConRein\ Int \\
&ConCompSinRein\ Int \mid ConCompConRein\ Int
\end{aligned}
$$

ya que el reintegro va a depender de la longitud de la apuesta jugada, con lo cual es mejor un entero adicional con dicha longitud y 0 indicando sin reintegro

$$
\begin{aligned}
\mathbf{data}\ ResultadoEscrutinio\ =\ &SinComp\ Int\ Int \mid ConComp\ Int\ Int \\
&\mathbf{deriving}\ Show
\end{aligned}
$$

con las pertinentes modificaciones en *escrutinio* supuesta definida la función no li-

neal *reintegra* que indica cuántas unidades de reintegro (a una apuesta de 6 números le corresponde una unidad de reintegro) le corresponden a una apuesta dada:

```
escrutinio (qs, qr)(ps, c, r) =
  if c `elem` qs then ConComp ac dc else SinComp ac dc
  where ac = length (comunes ps qs)
        dc = if r== qr then reintegra (length qs) else 0
```

Para simular la generación del reintegro en cada sorteo usamos una lista infinita de reintegros generada mediante

```
reins :: Semilla → [Reintegro]
reins = listaAzar (0, 9)
```

y emparejamos cada reintegro con su sorteo correspondiente

```
type NúmeroDeSorteos = Int
generaSorteos :: Semilla → NúmeroDeSorteos → [Sorteo]
generaSorteos sem n = generaAux n (bolas sem) (reins sem)
  where generaAux 0       bs rs = []
        generaAux (n + 1) bs rs = (ps, c, rs !! n) : generaAux n bs'
              where (c : ps, bs') = tomaSinRepetir 7 [] bs
```

Aunque se podría intentar hacer el recuento a través de una lista de contadores con la novena posición para aquellas apuestas con reintegro (p.e., iterando la función *actualiza cont* 9 tantas veces como unidades de reintegro le correspondan), es más fácil el recuento con campos etiquetados:

```
data ResultadoSimulaciónPrimitiva =
  T{deTres, deCuatro, deCinco, deCincoYcomp, deSeis, reintegros :: Int}
  deriving Show
```

de forma que, bajo la hipótesis de que el reintegro se cobra de manera adicional a cualquier otra categoría:

$$
\begin{array}{rcl}
actualizaTotal\ t\ (SinComp\ \ 3\ y) & = & t\{deTres\ =\ deTres\ t\ +1; \\
 & & \quad reintegros\ =\ reintegros\ t\ +y\} \\
actualizaTotal\ t\ (ConComp\ 3\ y) & = & t\{deTres\ =\ deTres\ t\ +1; \\
 & & \quad reintegros\ =\ reintegros\ t\ +y\} \\
actualizaTotal\ t\ (SinComp\ \ 4\ y) & = & t\{deCuatro\ =\ deCuatro\ t\ +1; \\
 & & \quad reintegros\ =reintegros\ t\ +y\} \\
actualizaTotal\ t\ (ConComp\ 4\ y) & = & t\{deCuatro\ =\ deCuatro\ t\ +1; \\
 & & \quad reintegros\ =\ reintegros\ t\ +y\} \\
actualizaTotal\ t\ (SinComp\ \ 6\ y) & = & t\{deSeis\ =\ deSeis\ t\ +1; \\
 & & \quad reintegros\ =\ reintegros\ t\ +y\} \\
actualizaTotal\ t\ (ConComp\ 6\ y) & = & t\{deSeis\ =\ deSeis\ t\ +1; \\
 & & \quad reintegros\ =reintegros\ t\ +y\} \\
actualizaTotal\ t\ (ConComp\ 5\ y) & = & t\{deCincoYcomp\ =\ deCincoYcomp\ t\ +1; \\
 & & \quad reintegros\ =\ reintegrost\ +y\}
\end{array}
$$

$$
\begin{array}{rcl}
actualizaTotal\ t\ (SinComp\ \ 5\ y) & = & t\{deCinco\ =deCinco\ t\ +1; \\
 & & \quad reintegros\ =\ reintegros\ t\ +y\} \\
actualizaTotal\ t\ (SinComp\ \ \_\ y) & = & t\{reintegros\ =reintegros\ t\ +y\} \\
actualizaTotal\ t\ (ConComp\ \_\ y) & = & t\{\ reintegros\ =\ reintegros\ t\ +y\}
\end{array}
$$

**Solución al Ejercicio 15.4** (pág. 391).– Se sigue inmediatamente de la mayor prioridad de la aplicación frente a la composición y del hecho de que $id\ g\ =g$.

## 21.12. TÉCNICAS DE PROGRAMACIÓN Y TRANSFORMACIONES DE PROGRAMAS

**Solución al Ejercicio 16.4** (pág. 404).– Siendo $I\ \equiv\ Suc\ O$, se puede demostrar que

$$
\begin{array}{ll}
(y*0=0) & y\ *\ O\ =O \\
(y*1=y) & y\ *I\ \ =y \\
(Suc(y)=y+1) & Suc\ y\ \ =y\ +\ I
\end{array}
$$

En efecto, para $(y*0=0)$ se procede por inducción sobre $y$:

— *Caso Base*:

$$
\begin{array}{ll}
 & O\ *\ O \\
\equiv & \{\,(1)(*)\,\} \\
 & O
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & (Suc\ y) * O \\
\equiv & \{\,(2)(*)\,\} \\
 & y * O\ +\ O \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & O\ +\ O \\
\equiv & \{\,(1)(+)\,\} \\
 & O
\end{array}
$$

y análogamente para $(y * 1 = y)$:

— *Caso Base*:

$$
\begin{array}{ll}
 & O\ *\ I \\
\equiv & \{\,(1)(*)\,\} \\
 & O
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & (Suc\ y) * I \\
\equiv & \{\,(2)(*)\,\} \\
 & y * I\ +\ I \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & y\ +\ I \\
\equiv & \{\,\text{conmutatividad de } (+)\,\} \\
 & Suc\ O\ +\ y \\
\equiv & \{\,(2)(+)\,\} \\
 & Suc\ (O\ +\ y) \\
\equiv & \{\,(1)(+)\,\} \\
 & Suc\ y
\end{array}
$$

mientras que $(Suc(y) = y + 1)$ se demuestra directamente:

$$
\begin{array}{ll}
 & Suc\ x \\
\equiv & \{\,(1)(+)\,\} \\
 & Suc\ (O + x) \\
\equiv & \{\,(2)(+)\,\} \\
 & Suc\ O\ +\ x \\
\equiv & \{\,\text{definición de } I\,\} \\
 & I\ +\ x \\
\equiv & \{\,(m + 1 = 1 + m) \text{ del Ejemplo 16.1}\,\} \\
 & x\ +\ I
\end{array}
$$

Para demostrar la propiedad asociativa probaremos las siguientes implicaciones

**(a)** $(*)$ distributiva a derecha $\Rightarrow$ $(*)$ conmutativa

**(b)** $(*)$ distrib. a derecha $\wedge$ $(*)$ conmut. $\Rightarrow$ $(*)$ distrib. a izquierda

**(c)** $(*)$ distributiva a izquierda $\Rightarrow$ $(*)$ asociativa

con lo cual sólo sería necesario demostrar la propiedad distributiva a la derecha.

**(a)** $(*)$ distributiva a derecha $\Rightarrow$ $(*)$ conmutativa. Suponiendo que $(*)$ es distributiva a la derecha hay que demostrar que $x * y = y * x$, y lo hacemos por inducción estructural sobre $x$:

— *Caso Base*:

$$
\begin{array}{ll}
 & O * y \\
\equiv & \{\, (1)(*) \,\} \\
 & O \\
\equiv & \{\, (y * 0 = 0) \,\} \\
 & y * O
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & (Suc\ x) * y = y * (Suc\ x) \\
\equiv & \{\, (2)(*),\ (Suc(x) = x + 1) \,\} \\
 & x * y + y = y * (x + I) \\
\equiv & \{\, \text{hipótesis de inducción, distr. derecha} \,\} \\
 & y * x + y = y * x + y * I \\
\equiv & \{\, (y * 1 = y) \,\} \\
 & Cierto
\end{array}
$$

**(b)** $(*)$ distrib. a derecha $\wedge$ $(*)$ conmut. $\Rightarrow$ $(*)$ distrib. a izquierda. Suponiendo que $(*)$ es distributiva a la derecha y conmutativa hay que demostrar:

$$(x + y) * z = x * z + y * z$$

La demostración se hace directamente:

$$
\begin{array}{ll}
 & (x + y) * z \\
\equiv & \{\, \text{conmutatividad de } (*) \,\} \\
 & z * (x + y) \\
\equiv & \{\, \text{distributividad a derecha de } (*) \,\} \\
 & z * x + z * y \\
\equiv & \{\, \text{conmutatividad de } (*) \,\} \\
 & x * z + y * z
\end{array}
$$

**(c)** $(*)$ distributiva a izquierda $\Rightarrow$ $(*)$ asociativa. Suponiendo que $(*)$ es distributiva a la izquierda hay que demostrar que $x * (y * z) = (x * y) * z$ de nuevo por inducción estructural sobre $x$:

— *Caso Base*:

$$
\begin{array}{ll}
 & O * (y * z) = (O * y) * z \\
\equiv & \{\,(1)(*) \text{ dos veces}\,\} \\
 & O = O * z \\
\equiv & \{\,(1)(*)\,\} \\
 & Cierto
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & Suc\ x * (y * z) = (Suc\ x * y) * z \\
\equiv & \{\,(2)(*) \text{ dos veces}\,\} \\
 & x * (y * z) + y * z = (x * y + y) * z \\
\equiv & \{\text{ hipótesis de inducción}\,\} \\
 & (x * y) * z + y * z = (x * y + y) * z \\
\equiv & \{\,\text{distr. izquierda}\,\} \\
 & Cierto
\end{array}
$$

Finalmente veamos la distributividad a la derecha $x * (y + z) = x * y + x * z$ por inducción sobre $x$:

— *Caso Base*:

$$
\begin{array}{ll}
 & O * (y + z) = O * y + O * z \\
\equiv & \{\,(1)(*) \text{ tres veces}\,\} \\
 & O = O + O \\
\equiv & \{\,(1)(+)\,\} \\
 & Cierto
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & (Suc\ x) * (y + z) = (Suc\ x) * y + (Suc\ x) * z \\
\equiv & \{\,(2)(*) \text{ tres veces}\,\} \\
 & x * (y + z) + (y + z) = (x * y + y) + (x * z + z) \\
\equiv & \{\text{ hipótesis de inducción}\,\} \\
 & x * y + x * z + (y + z) = (x * y + y) + (x * z + z) \\
\equiv & \{\,\text{conm. y asoc. de } (+) \text{ (Ejemplo 16.1)}\,\} \\
 & Cierto
\end{array}
$$

**Solución al Ejercicio 16.5** (pág. 405).– Definiendo

$$\begin{array}{ll} (def1) & I \;\; = Suc\ O \\ (def-) & -\ x\ = O\ -\ x \end{array}$$

es fácil demostrar que

$$\begin{array}{ll} (Suc(b) = 1 + b) & Suc\ b\ = I + b \\ (Pre(b) = b - 1) & Pre\ b\ = b - I \\ (Pre(0) = -1) & -\ I\ \;\; = Pre\ O \end{array}$$

Además:

$$\begin{array}{l} \quad Pre\ (-x) \\ \equiv\ \{\, (def-)\,\} \\ \quad Pre\ (O - x) \\ \equiv\ \{\, 2)-\,\} \\ \quad O - Suc\ x \\ \equiv\ \{\, (def-)\,\} \\ \quad -\ Suc\ x \end{array} \qquad \begin{array}{l} \quad -\ Pre\ x \\ \equiv\ \{\, (def-)\,\} \\ \quad O - Pre\ x \\ \equiv\ \{\, 3)-\,\} \\ \quad Suc\ (O - x) \\ \equiv\ \{\, (def-)\,\} \\ \quad Suc\ (-x) \end{array}$$

Es decir

$$\begin{array}{ll} (cs1) & Pre\ (-x)\ =\ -\ Suc\ x \\ (cs2) & -\ \ Pre\ x\ = Suc\ (-x) \end{array}$$

Por otra parte, también puede demostrarse que

$$\begin{array}{ll} (iden0) & -\ O\ \;\; = O \\ (idenx) & -\ (-x)\ = x \end{array}$$

donde la demostración para $(iden0)$ es

$$\begin{array}{l} \quad -\ O \\ \equiv\ \{\, (def-)\,\} \\ \quad O - O \\ \equiv\ \{\, 1)-\,\} \\ \quad O \end{array}$$

mientras que $(idenx)$ la demostraremos por inducción estructural sobre $x$:

— *Caso Base*:

$$\begin{array}{l} \quad -\ (-O) \\ \equiv\ \{\, (iden0)\,\} \\ \quad -\ O \\ \equiv\ \{\, (iden0)\,\} \\ \quad O \end{array}$$

— *Paso Inductivo*: tiene dos casos, como se refleja en:

$$
\begin{array}{ll}
 & -\,(-\,Suc\ x) \\
\equiv & \{\,(cs1)\,\} \\
 & -\,(Pre\ (-x)) \\
\equiv & \{\,(cs2)\,\} \\
 & Suc\ (-(-x)) \\
\equiv & \{\text{ hipótesis de inducción }\} \\
 & Suc\ x
\end{array}
\qquad
\begin{array}{ll}
 & -\,(-\,Pre\ x) \\
\equiv & \{\,(cs2)\,\} \\
 & -\,(Suc\ (-x)) \\
\equiv & \{\,(cs1)\,\} \\
 & Pre\ (-(-x)) \\
\equiv & \{\text{ hipótesis de inducción }\} \\
 & Pre\ x
\end{array}
$$

También se tiene

$$-\,x\ =\ O \qquad \Rightarrow \qquad x\ =\ O$$

ya que

$$
\begin{array}{ll}
 & -\,x\ =\ O \\
\Rightarrow & \{\ (is)\,\} \\
 & -\,(-x)\ =\ -O \\
\Rightarrow & \{\ (idenx)\,\} \\
 & x\ =\ -O \\
\Rightarrow & \{\ (iden0)\,\} \\
 & x\ =\ O
\end{array}
$$

Veremos a continuación que para obtener ciertas propiedades de las nuevas suma y diferencia ($(+)$ y $(-)$ para $Ent$) es necesario considerar algunos axiomas. Parece ser que es suficiente considerar el axioma

$$(Ax1) \qquad Pre\ I\ =\ O$$

del cual se deducen fácilmente

$$
\begin{array}{ll}
(i1) & Pre\ (Suc\ x)\ =\ x \\
(i2) & Suc\ (Pre\ x)\ =\ x \\
(c1) & Pre\ a\ +\ b\ =\ a\ +\ Pre\ b \\
(c2) & Suc\ a\ +\ b\ =\ a\ +\ Suc\ b
\end{array}
$$

En efecto, pues tanto $(i1)$ como $(i2)$ se deducen directamente:

$$
\begin{array}{ll}
 & Pre\ (Suc\ x) \\
\equiv & \{\,(Suc(x) = 1 + x)\,\} \\
 & Pre\ (I + x) \\
\equiv & \{\,3)+\,\} \\
 & Pre\ I + x \\
\equiv & \{\,(Ax1)\,\} \\
 & O + x \\
\equiv & \{\,1)+\,\} \\
 & x
\end{array}
\qquad
\begin{array}{ll}
 & Suc\ (Pre\ x) \\
\equiv & \{\,(idenx)\,\} \\
 & -\,(-\ Suc\ (Pre\ x)) \\
\equiv & \{\,(cs1)\,\} \\
 & -\,(Pre\ (-\ Pre\ x)) \\
\equiv & \{\,(cs2)\,\} \\
 & -\,(Pre\ (Suc\ (-x))) \\
\equiv & \{\,(i1)\,\} \\
 & -\,(-x) \\
\equiv & \{\,(idenx)\,\} \\
 & x
\end{array}
$$

mientras que $(c1)$ lo demostramos por inducción sobre $a$

— *Caso Base*: ($a \equiv O$)

$$
\begin{array}{ll}
 & Pre\ O\ +\ b \\
\equiv & \{\,3)+\,\} \\
 & Pre\ (O + b) \\
\equiv & \{\,1)+\,\} \\
 & Pre\ b \\
\equiv & \{\,1)+\,\} \\
 & O\ +\ Pre\ b
\end{array}
$$

— *Paso Inductivo*: se deduce de la siguiente prueba para los dos casos:

$$
\begin{array}{ll}
 & Pre\ (Suc\ a) + b \\
\equiv & \{\,(i1)\,\} \\
 & a + b \\
\equiv & \{\,(i2)\,\} \\
 & Suc\ (Pre\ (a + b)) \\
\equiv & \{\,3)+\,\} \\
 & Suc\ (Pre\ a + b) \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & Suc\ (a + Pre\ b) \\
\equiv & \{\,2)+\,\} \\
 & Suc\ a + Pre\ b
\end{array}
\qquad
\begin{array}{ll}
 & Pre\ (Pre\ a) + b \\
\equiv & \{\,3)+\,\} \\
 & Pre\ (Pre\ a + b) \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & Pre\ (a + Pre\ b) \\
\equiv & \{\,3)+\,\} \\
 & Pre\ a + Pre\ b
\end{array}
$$

y finalmente $(c2)$ lo probamos directamente:

$$
\begin{array}{ll}
 & Suc\ a\ +\ b \\
\equiv & \{\,(i1)\,\}
\end{array}
$$

$$
\begin{array}{ll}
 & Suc\ a\ +\ Pre\ (Suc\ b) \\
\equiv & \{\,(c1)\,\} \\
 & Pre\ (Suc\ a)\ +\ Suc\ b \\
\equiv & \{\,(i1)\,\} \\
 & a\ +\ Suc\ b
\end{array}
$$

También se tiene (por inducción sobre $y$) que:

$$(dif) \qquad x\ -\ y \qquad = \qquad x\ +\ (-y)$$

— *Caso Base*:

$$
\begin{array}{ll}
 & x\ -\ O \\
\equiv & \{\,1)-\,\} \\
 & x \\
\equiv & \{\,1)+\,\} \\
 & O\ +\ x \\
\equiv & \{\,\text{conmutatividad de } + \text{ (aún no demostrada)}\,\} \\
 & x\ +\ O \\
\equiv & \{\,(iden0)\,\} \\
 & x\ +\ (-O)
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & x - Suc\ y \\
\equiv & \{\,2)-\,\} \\
 & Pre\ (x - y) \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & Pre\ (x + (-y)) \\
\equiv & \{\,3)+\,\} \\
 & Pre\ x + (-y) \\
\equiv & \{\,(c1)\,\} \\
 & x + Pre\ (-y) \\
\equiv & \{\,(cs1)\,\} \\
 & x + (-\ Suc\ y)
\end{array}
\qquad
\begin{array}{ll}
 & x - Pre\ y \\
\equiv & \{\,3)-\,\} \\
 & Suc\ (x - y) \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & Suc\ (x + (-y)) \\
\equiv & \{\,2)+\,\} \\
 & Suc\ x + (-y) \\
\equiv & \{\,(c2)\,\} \\
 & x + Suc\ (-y) \\
\equiv & \{\,(cs2)\,\} \\
 & x + (-\ Pre\ y)
\end{array}
$$

de donde se pueden obtener:

$$
\begin{array}{ll}
(r1) & x\ -\ Suc\ y\ = Pre\ x\ -\ y \\
(r2) & x\ -\ Pre\ y\ = Suc\ x\ -\ y
\end{array}
$$

como queda demostrado mediante

$$
\begin{array}{ll}
 & x - Suc\ y \\
\equiv & \{\,(dif)\,\} \\
 & x + (-\ Suc\ y) \\
\equiv & \{\,(cs1)\,\} \\
 & x + Pre\ (-y) \\
\equiv & \{\,(c1)\,\} \\
 & Pre\ x + (-y) \\
\equiv & \{\,(dif)\,\} \\
 & Pre\ x - y
\end{array}
\qquad
\begin{array}{ll}
 & x - Pre\ y \\
\equiv & \{\,(dif)\,\} \\
 & x + (-\ Pre\ y) \\
\equiv & \{\,(cs2)\,\} \\
 & x + Suc\ (-y) \\
\equiv & \{\,(c2)\,\} \\
 & Suc\ x + (-y) \\
\equiv & \{\,(dif)\,\} \\
 & Suc\ x - y
\end{array}
$$

La demostración de la propiedad asociativa es muy fácil: se puede hacer de la misma forma que con el tipo $Nat$ (ver Ejemplo 16.1 y Ejercicio 16.4). Veamos ahora (por inducción sobre $y$) la propiedad

$$x\ -\ (y\ +\ z) \qquad = \qquad (x\ -\ y)\ -\ z$$

— *Caso Base*:

$$
\begin{array}{ll}
 & x\ -\ (O\ +\ z)\ =\ (x\ -\ O)\ -\ z \\
\equiv & \{\,1)+,2)-\,\} \\
 & Cierto
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & x - (Pre\ y + z) \\
\equiv & \{\,(c1)\,\} \\
 & x - (y + Pre\ z) \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & (x - y) - Pre\ z \\
\equiv & \{\,(r2)\,\} \\
 & Suc\ (x - y) - z \\
\equiv & \{\,(3)-\,\} \\
 & (x - Pre\ y) - z
\end{array}
\qquad
\begin{array}{ll}
 & x - (Suc\ y + z) \\
\equiv & \{\,(c2)\,\} \\
 & x - (y + Suc\ z) \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & (x - y) - Suc\ z \\
\equiv & \{\,(r1)\,\} \\
 & Pre\ (x - y) - z \\
\equiv & \{\,(2)-\,\} \\
 & (x - Suc\ y) - z
\end{array}
$$

y también se tiene:

$$
\begin{array}{rcl}
-\ (x + y) & = & -\ x - y \\
x - (y - z) & = & (x - y) + z \\
y\ -\ y & = & O \\
-\ y\ +\ y & = & O
\end{array}
$$

y de los cuales sólo demostraremos el último por inducción sobre $y$:

— *Caso Base*:

$$
\begin{array}{ll}
 & -\ O\ +\ O \\
\equiv & \{\,(iden0)\,\} \\
 & O\ +\ O \\
\equiv & \{\,(1)+\,\} \\
 & O
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
 & Suc\ y - Suc\ y \\
\equiv & \{\,(r1)\,\} \\
 & Pre\ (Suc\ y) - y \\
\equiv & \{\,(i1)\,\} \\
 & y - y \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & O
\end{array}
\qquad\qquad
\begin{array}{ll}
 & Pre\ y - Pre\ y \\
\equiv & \{\,(r2)\,\} \\
 & Suc\ (Pre\ y) - y \\
\equiv & \{\,(i2)\,\} \\
 & y - y \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & O
\end{array}
$$

Finalmente, ahora ya es fácil demostrar la propiedad commutativa de $(+)$ para $Ent$ a partir de la propiedad

$$x\ -\ y\ =\ O \qquad \Rightarrow \qquad x\ =\ y$$

probando, por ejemplo, que

$$(a+b)\ -\ (b+a)\ =\ O$$

**Solución al Ejercicio 16.8** (pág. 410).– Para $n = 0$ y para $n = 1$ es trivial; supongamos que $(FI)$ es cierta para los valores $k \leq n$; entonces, para $n \geq 2$

$$
\begin{array}{ll}
 & \mathit{fib}'\ (n+1)\ p\ q \\
\equiv & \{\ \text{definición de } \mathit{fib}'\,\} \\
 & \mathit{fib}'\ n\ q\ (p+q) \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
 & \mathit{fib}'\ (n-2)\ q\ (p+q)\ +\ \mathit{fib}'\ (n-1)\ q\ (p+q) \\
\equiv & \{\ \text{definición de } \mathit{fib}'\,\} \\
 & \mathit{fib}'\ (n-1)\ p\ q\ +\ \mathit{fib}'\ n\ p\ q
\end{array}
$$

**Solución al Ejercicio 16.9** (pág. 410).– Si definimos

$$
\begin{array}{l}
pro\ O\ \ \ \ \ \ \ \ \_\ p\ =\ p \\
pro\ (Suc\ x)\ y\ p\ =\ pro\ x\ y\ (y+p)
\end{array}
$$

podemos considerar un nuevo operador $(*:)$ para la multiplicación de $Nat$ definido mediante:

$$x *: y = pro\ x\ y\ O$$

Para demostrar que el nuevo $(*:)$ y el antiguo $(*)$ llevan a cabo la misma operación (i.e., que $(*:) = (*)$) probaremos que

$$\forall\, x, y \,.\, x, y :: Nat \,.\, pro\ x\ y\ p = x * y + p$$

con lo cual, si aislamos la $x$, lo anterior equivale a

$$\forall\, x \,.\, x :: Nat \,.\, (\forall\, y \,.\, y :: Nat \,.\, pro\ x\ y\ p = x * y + p)$$

y entonces razonaremos por inducción estructural sobre $x$:

$$\begin{array}{ll} & pro\ O\ y\ p = O * y + p \\ \equiv & \{\,(1)pro, (1)(*)\,\} \\ & p = O + p \\ \equiv & \{\,(1)(+)\,\} \\ & Cierto \end{array}$$

$$\begin{array}{ll} & pro\ (Suc\ x)\ y\ p = (Suc\ x) * y + p \\ \equiv & \{\,(2)pro, (2)(*)\,\} \\ & pro\ x\ y\ (y + p) = (x * y + y) + p \\ \equiv & \{\ \text{hipótesis de inducción}\,\} \\ & x * y + (y + p) = (x * y + y) + p \\ \equiv & \{\,\text{asoc. de } (+) \text{ (Ejemplo 16.1)}\,\} \\ & Cierto \end{array}$$

En definitiva, podemos cambiar la instancia de $Num$ para $Nat$ por la instancia equivalente siguiente:

```
instance Num Nat where
  O       + m = m
  (Suc n) + m = Suc (n + m)

  x * y = pro x y O
   where
     pro :: Nat -> Nat -> Nat
     pro O       _ p = p
     pro (Suc x) y p = pro x y (y + p)
```

**Solución al Ejercicio 16.14** (pág. 418).– Procederemos por inducción estructural sobre $xs$:

— *Caso Base*: $(xs \equiv [\,])$

$$\begin{array}{ll} & g\,[\,] = 2 * suma[\,] \\ \equiv & \{\,\text{definición de } g, (1)suma\,\} \\ & suma\,(doble\,[\,]) = 2 * 0 \\ \equiv & \{\,(1)doble, (*)\,\} \\ & suma\,[\,] = 0 \end{array}$$

$\equiv$ { (1)$suma$ }
$Cierto$

— *Paso Inductivo*:

$g\ (x : xs)\ =\ 2\ *\ suma\ (x : xs)$
$\equiv$ { definición de $g$, (2)$suma$ }
$suma\ (doble\ (x : xs))\ =\ 2 * (x\ +\ suma\ xs)$
$\equiv$ { (2)$doble$, distributividad de $(+)$ respecto a $(*)$ }
$suma\ (2 * x\ :\ doble\ xs)\ =\ 2 * x\ +\ 2 * suma\ xs$
$\Leftarrow$ { (2)$suma$ }
$suma\ (doble\ xs)\ =\ 2 * suma\ xs$
$\Leftarrow$ { definición de $g$ }
$HI$

**Solución al Ejercicio 16.18** (pág. 425).– A partir de $reemplaza$ y $ultl$ es fácil escribir una función $lult$ que cumpla los requisitos pero que dé dos pasadas a la lista:

$$\begin{array}{ll} lult\ [x] & = [x] \\ lult\ u@(\_ : \_ : \_) & = reemplaza\ (ultl\ u)\ u \end{array}$$

Para obtener una versión de $lult$ que sólo dé una pasada pueden usarse la técnica D/P o bien intentar eliminar directamente las funciones auxiliares. Utilizando la técnica D/P:

Antes de nada veamos una propiedad de la función $ultl$:

$$ultl\ (x : x' : xs)\ =\ ultl\ (\ (seg\ x\ x') : xs\ )$$

$ultl\ (x : x' : xs)$
$\equiv$ { desplegar }
$seg\ x\ (ultl\ (x' : xs))$
$\equiv$ { desplegar }
$seg\ x\ (seg\ x'\ (ultl\ xs))$
$\equiv$ { asociatividad de $seg$ }
$seg\ (seg\ x\ x')\ (ultl\ xs)$
$\equiv$ { plegar }
$ultl\ ((seg\ x\ x') : xs)$

A partir de aquí intentaremos introducir con ayuda de una abstracción (un cualificador) un cálculo en una sola pasada

$lult\ (x : x' : xs)$
$\equiv$ { desplegar }

$$
\begin{array}{ll}
 & reemplaza\ (ultl\ (x : x' : xs))\ (x : x' : xs) \\
\equiv & \{\text{ instanciación de } reemplaza \text{ dos veces }\} \\
 & ultl\ (x : x' : xs) : ultl(x : x' : xs) : reemplaza\ (ultl(x : x' : xs))xs \\
\equiv & \{\text{ abstracción }\} \\
 & m : m : u\ \mathbf{where}\ m : u\ =\ ultl(x : x' : xs) : reemplaza(ultl(x : x' : xs))xs
\end{array}
$$

Vamos ahora a transformar la expresión cualificadora última con objeto de eliminar las funciones *reemplaza* y *ultl*

$$
\begin{array}{ll}
 & ultl\ (x : x' : xs)\ :\ reemplaza\ ultl\ (x : x' : xs)\ xs \\
\equiv & \{\text{ propiedad de } ultl \text{ demostrada anteriormente }\} \\
 & ultl\ (seg\ x\ x') : xs)\ :\ reemplaza\ (ultl\ ((seg\ x\ x') : xs))xs \\
\equiv & \{\text{ instanciación de } reemplaza\ \} \\
 & reemplaza\ (ultl\ ((seg\ x\ x') : xs))\ ((seg\ x\ x') : xs) \\
\equiv & \{\text{ instanciación de } lult\ \} \\
 & lult\ ((seg\ x\ x') : xs) \\
\equiv & \{\text{ definición de } seg\ \} \\
 & lult\ (x' : xs)
\end{array}
$$

de donde finalmente:

$$
\begin{array}{lcl}
lult\ [x] & = & [x] \\
lult\ (x : x' : xs) & = & m : m : u\ \mathbf{where}\ m : u\ =\ lult\ (x' : xs)
\end{array}
$$

Eliminando directamente las funciones auxiliares *reemplaza* y *ultl*:

$$
\begin{array}{ll}
 & lult\ (x : x' : xs) \\
\equiv & \{\text{ 2)}lult\ \} \\
 & reemplaza\ (ultl\ (x : x' : xs))\ (x : x' : xs) \\
\equiv & \{\text{ 2)}ultl \text{ dos veces y 2)}reemplaza \text{ dos veces }\} \\
 & ultl\ xs\ :\ ultl\ xs\ :\ reemplaza\ (ultl\ xs)\ xs \\
\equiv & \{\text{ abstracción }\} \\
 & m : m : u\ \mathbf{where}\ m : u\ =\ ultl\ xs\ :\ reemplaza\ (ultl\ xs)\ xs \\
\equiv & \{\text{ 2)}reemplaza\ \} \\
 & m : m : u\ \mathbf{where}\ m : u\ =\ reemplaza\ (ultl\ xs)\ (x' : xs) \\
\equiv & \{\text{ 2)}ultl\ \} \\
 & m : m : u\ \mathbf{where}\ m : u\ =\ reemplaza\ (ultl\ (x' : xs))\ (x' : xs) \\
\equiv & \{\text{ plegar } reemplaza\ \} \\
 & m : m : u\ where\ m : u\ =\ lult\ (x' : xs)
\end{array}
$$

con lo cual ya se puede reescribir la función *lult* para que sólo dé una pasada a la lista, pues para listas de un elemento se aplicará la misma primera ecuación que teníamos y para listas con dos o más elementos se aplicará la ecuación obtenida en la deducción anterior

$$
\begin{array}{lcl}
lult\ [x] & = & [x] \\
lult\ (\_ : x' : xs) & = & m : m : u\ \mathbf{where}\ m : u\ =\ lult\ (x' : xs)
\end{array}
$$

**Solución al Ejercicio 16.19** (pág. 425).– En el método de Pettorossi–Skowron la idea es considerar la función *reemplaza* que duplica una estructura lista; es la misma que la del Ejercicio 16.18, pero invirtiendo el orden de los argumentos:

$$\begin{array}{l} reemplaza :: [a] \to a \to [a] \\ reemplaza\ [\,]\ \ \_ = [\,] \\ reemplaza\ (x : xs)\ y = y : reemplaza\ xs\ y \end{array}$$

Consideremos ahora una función

$$fult :: [a] \to (a,\ a \to [a])$$

de forma que *fult xs* devuelve un par $(u,\ f)$ donde $u$ es el último de la lista y $f$ tiene el mismo comportamiento que la función *reemplaza*, de tal suerte tendremos que la siguiente función devuelve la lista de últimos

$$lult'\ xs = f\ u\ \mathbf{where}\ (u, f) = fult\ xs$$

El caso base para *fult* se deduce de

$$\begin{array}{ll} & lult'\ [x] = f\ u\ \mathbf{where}\ (u, f) = fult\ [x] \\ \equiv & \{\ \} \\ & [x] = f\ u\ \mathbf{where}\ (x, f) = fult\ [x] \\ \equiv & \{\ \} \\ & [x] = f\ x\ \mathbf{where}\ (x, f) = fult\ [x] \\ \Rightarrow & \\ & f = \lambda\, y \to [y] \quad \wedge \quad fult\ [x] = (x, \lambda\, y \to [y]) \end{array}$$

y es fácil intuir el comportamiento en el caso general:

$$\begin{array}{l} fult\ [x] = (x,\ \lambda\, y \to [y]) \\ fult\ (\_ : xs) = (u,\ \lambda\, y \to y : f\ y) \\ \qquad\qquad \mathbf{where}\ (u, f) = fult\ xs \end{array}$$

La corrección de la función $lult'$ se puede hacer, bien demostrando (por inducción estructural) que $lultl = lult'\ l$ donde *lult* es la versión en dos pasadas dada en el Ejercicio 16.18:

$$\begin{array}{l} lult\ [x] = [x] \\ lult\ u@(\_ : \_ : \_) = reemplaza\ (ultl\ u)\ u \end{array}$$

o bien directamente, demostrando

$$\begin{array}{l} \forall\, xs \,.\, xs :: [a],\ xs \neq [\,] \,. \\ \qquad (cabeza\ (lult'\ xs) = ultl\ xs) \quad \wedge \quad (ti\ (lult'\ xs)) \end{array}$$

donde $ultl$ es la función del apartado 2 del Ejercicio 16.18:

$$
\begin{array}{lcl}
ultl & :: & [a] \rightarrow a \\
ultl\ [x] & = & x \\
ultl\ (x : xs) & = & ultl\ xs
\end{array}
$$

y $ti$ comprueba si todos los elementos de una lista son iguales:

$$
\begin{array}{lcl}
ti & :: & (Eq\ a) \Rightarrow [a] \rightarrow Bool \\
ti\ [x] & = & True \\
ti\ (x : x' : xs) & = & x == x'\ \&\&\ ti\ (x' : xs)
\end{array}
$$

Para la demostración directa, se tiene que

— *Caso Base*: (trivial)

— *Paso Inductivo*: Usando el siguiente resultado

$$
\begin{array}{ll}
 & lult'\ (x : x' : xs) \\
\equiv & \{\ lult \text{ y } fult \text{ dos veces}\ \} \\
 & u\ :\ u\ :\ g\ u\ \mathbf{where}\ (u, g)\ =\ fult\ xs
\end{array}
$$

la demostración del paso de inducción sería:

$$
\begin{array}{ll}
 & cabeza\ (lult'\ (x : x' : xs)) \\
\equiv & \{\ \text{definición de } cabeza \text{ y resultado anterior}\ \} \\
 & u\ \mathbf{where}\ (u, g)\ =\ fult\ xs \\
\equiv & \{\ lult \text{ y } fult\ \} \\
 & cabeza\ (lult\ (x' : xs)) \\
\equiv & \{\ \text{hipótesis de inducción}\ \} \\
 & ultl\ (x' : xs) \\
\equiv & \{\ 2) ultl\ \} \\
 & ultl\ (x : x' : xs)
\end{array}
$$

con lo cual

$$
\begin{array}{ll}
 & ti\ (lult'\ (x : x' : xs)) \\
\equiv & \{\ ti \text{ y lo anterior}\ \} \\
 & ti\ (u\ :\ g\ u)\ \mathbf{where}\ (u, g)\ =\ fult\ xs \\
\equiv & \{\ lult\ \} \\
 & ti\ (\ lult\ (x' : xs)\ ) \\
\Leftarrow & \\
 & HI
\end{array}
$$

**Solución al Ejercicio 16.20** (pág. 425).– Para resolver el problema veamos en primer lugar uno algo más simple; se trata de escribir una función que reemplace todos los nodos

de un árbol (de enteros, por ejemplo) por un elemento dado, de suerte que tal función, parcializada convenientemente, replica la misma estructura del árbol:

$$
\begin{array}{l}
f\ ::\ \acute{A}rbol\ \to\ Int\ \to \acute{A}rbol \\
f\ Vacío\ \ \ \ \ \ \ \ \_\ =\ Vacío \\
f\ (Nodo\ i\ x\ d)\ y\ =\ Nodo\ (f\ i\ y)\ y\ (f\ d\ y)
\end{array}
$$

de forma que $f\ a$, que tiene por tipo $Int\ \to \acute{A}rbol$, se define también como

$$
\begin{array}{l}
f\ Vacío\ \ \ \ \ \ \ \ =\ \lambda \_\ \to\ Vacío \\
f\ (Nodo\ i\ x\ d)\ =\ \lambda\, y\ \to\ Nodo\ (f\ i\ y)\ y\ (f\ d\ y)
\end{array}
$$

También podemos recorrer (en una pasada) un árbol para obtener el máximo; mezclando ambas pasadas obtenemos la siguiente solución, que resulta ser una generalización del *método de Pettorossi-Skowron*:

$$
\begin{array}{l}
fmax\ \ \ \ \ \ \ \ \ \ \ \ \ \ ::\ \acute{A}rbol\ \to\ (Int\ \to \acute{A}rbol,\ Int) \\
fmax\ Vacío\ \ \ \ \ \ \ \ =\ (\lambda\, y\ \to\ Vacío,\ minInt) \\
fmax\ (Nodo\ i\ c\ d)\ =\ (\lambda\, y\ \to\ Nodo\ (f1\ y)\ y\ (f2\ y),\ max3\ m1\ c\ m2) \\
\quad \mathbf{where} \\
\quad\ (f1,\ m1)\ \ \ \ =\ fmax\ i \\
\quad\ (f2,\ m2)\ \ \ \ =\ fmax\ d \\
\quad\ max3\ x\ y\ z\ =\ max\ (max\ x\ y)\ z \\
\quad\ max\ x\ y\ \ \ \ =\ \mathbf{if}\ x < y\ \mathbf{then}\ y\ \mathbf{else}\ x \\
amax\ \ ::\ \acute{A}rbol\ \to \acute{A}rbol \\
amax\ a\ =\ f\ m\ \mathbf{where}\ (f,\ m)\ =\ fmax\ a
\end{array}
$$

**Solución al Ejercicio 16.21** (pág. 425).– Podemos generalizar la solución del Ejercicio 16.20 en la forma (ver capítulo 4, donde se describen las listas por comprensión):

$$
\begin{array}{l}
fmin\ V\ \ \ \ \ \ \ \ =\ (\lambda \_\ \to\ V\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ ,\ maxInt) \\
fmin\ (H\ x\ )\ \ \ \ =\ (\lambda\, y\ \to\ H\ y\ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ ,\ x) \\
fmin\ (N\ x\ as)\ =\ (\lambda\, y\ \to\ N\ y\ [f\ y \mid (f, \_) \leftarrow\ ps]\ ,\ m) \\
\quad \mathbf{where} \\
\quad\ ps\ \ \ \ =\ map\ fmin\ as \\
\quad\ m\ \ \ \ \ =\ foldr\ min\ [x \mid (\_, x) \leftarrow\ ps]\ maxInt \\
\quad\ min\ =\ \lambda\, x\ y\ \to\ \mathbf{if}\ x < y\ \mathbf{then}\ x\ \mathbf{else}\ y
\end{array}
$$

**Solución al Ejercicio 16.23** (pág. 443).– Sea $long\ xs\ \equiv\ \#xs$ y $mezcla\ \equiv\ mez$; hay que probar

$$\forall\, u\ \mathbf{.}\ u\ ::\ [a]\ \mathbf{.}\ (\forall\, v\ \mathbf{.} v\ ::\ [b]\ \mathbf{.}\ p\ u\ v)$$

donde el predicado $p$ viene dado por

$$p\ u\ v\ =\ (\ \#(mez\ u\ v) == min\ \#u\ \#v\ )$$

Procedemos por inducción sobre la primera lista:

— *Caso Base*:

$$
\begin{array}{ll}
 & p\ [\,]\ v \\
\equiv & \\
 & \#(mez\ [\,]\ v)\ =\ min\ 0\ \#v \\
\equiv & \{\,1)\ mez, \#v \geq 0\,\} \\
 & \#[\,]\ =\ 0 \\
\equiv & \\
 & Cierto
\end{array}
$$

— *Paso Inductivo*: Se trata de probar la implicación

$$
\begin{array}{lll}
(HI) & & \forall\ v'\ \centerdot\ v'\ ::\ [b]\ \centerdot\ p\ u\ v' \\
 & \Rightarrow & \\
 & & \forall\ v\ \centerdot\ v\ ::\ [b]\ \centerdot\ p\ (x:u)\ v
\end{array}
$$

que podemos escribir:

$$\forall\ v\ \centerdot\ v\ ::\ [b]\ \centerdot\ HI\ \Rightarrow\ p\ (x:u)\ v$$

Si $v$ es vacía, el consecuente es cierto:

$$
\begin{array}{ll}
 & p\ (x:u)\ [\,] \\
\equiv & \\
 & \#(mez\ (x:u)\ [\,])\ =\ min\ \#(x:u)\ \#[\,] \\
\equiv & \{\,(1)\ mez, \#(x:u) \geq 1\,\} \\
 & \#[\,]\ =\ \#[\,]
\end{array}
$$

Finalmente, si la lista no es vacía:

$$
\begin{array}{ll}
 & p\ (x:u)\ (y:v) \\
\equiv & \\
 & \#(mez\ (x:u)\ (y:v))\ =\ min\ \#(x:u)\ \#(y:v) \\
\equiv & \{\,(3) mez, 2) \#\,\} \\
 & \#((x,y):mez\ u\ v)\ =\ min\ (1+\#u)\ (1+\#v) \\
\equiv & \{\,\text{propiedades de } min\,\} \\
 & 1\ +\ \#(mez\ u\ v)\ =\ 1\ +\ min\ \#u\ \#v \\
\Leftarrow & \\
 & HI
\end{array}
$$

**Solución al Ejercicio 16.24** (pág. 443).– Por inducción estructural sobre $u$:

— *Caso Base*:

$$
\begin{array}{ll}
& [\,] +\!\!+ v \setminus\!\setminus [\,] \\
\equiv & \{ +\!\!+ \} \\
& v \setminus\!\setminus [\,] \\
\equiv & \{ \setminus\!\setminus \} \\
& v
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
& (x : u) +\!\!+ v \setminus\!\setminus (x : u) \\
\equiv & \{ +\!\!+ \} \\
& x : (u +\!\!+ v) \setminus\!\setminus (x : u) \\
\equiv & \{ \setminus\!\setminus \} \\
& x : (u +\!\!+ v)\ `del`\ x \setminus\!\setminus u \\
\equiv & \{ del \} \\
& u +\!\!+ v \setminus\!\setminus u \\
\equiv & \{ \text{ hipótesis de inducción } \} \\
& v
\end{array}
$$

**Solución al Ejercicio 16.25** (pág. 443).– El caso base para la primera es trivial, y el paso inductivo:

$$
\begin{array}{ll}
& (x : u) +\!\!+ [\,] \\
\equiv & \{ 2) +\!\!+ \} \\
& x : (u +\!\!+ [\,]) \\
\equiv & \{ \text{ hipótesis de inducción } \} \\
& x : u
\end{array}
$$

La segunda propiedad se prueba directamente:

$$
\begin{array}{ll}
& [x] +\!\!+ u \\
\equiv & \\
& (x : [\,]) +\!\!+ u \\
\equiv & \{ 2) +\!\!+ \} \\
& x : ([\,] +\!\!+ u) \\
\equiv & \{ 1) +\!\!+ \} \\
& x : u
\end{array}
$$

**Solución al Ejercicio 16.26** (pág. 444).– Usando un parámetro acumulador se obtiene:

$$
\begin{array}{l}
inv :: [a] \to [a] \\
inv\ w = inv2\ w\ [\,]
\end{array}
$$

$$
\begin{array}{l}
inv2 :: [a] \to [a] \to [a] \\
inv2\ [\,]\ \ \ \ \ \ w = w \\
inv2\ (x : xs)\ w = inv2\ xs\ (x : w)
\end{array}
$$

cuya corrección se abordó en el Ejercicio 6.34.

**Solución al Ejercicio 16.27** (pág. 444).– Utilizando la definición

$$
\begin{array}{l}
fib\ n = fn\ \mathbf{where}\ (fn,\ fnm1) = parFib\ n \\
\\
parFib\ 0 \ \ \ \ \ \ \ \ = (0, 1) \\
parFib\ (n + 1) = (fnm1,\ fnm1 + fn)\ \mathbf{where}\ (fn,\ fnm1) = parFib\ n
\end{array}
$$

se tiene la siguiente secuencia de reducciones (se abrevia $parFib$ mediante $pf$ y $\mathbf{where}$ mediante $wh$):

$$
\begin{array}{l}
\quad fib\ 2 \\
\Longrightarrow \\
\quad f2\ wh\ (f2, f3) = pf\ 2 \\
\Longrightarrow \\
\quad f2\ wh(f2, f3) = (f2, f2 + f1)\ wh\ (f1, f2) = pf\ 1 \\
\Longrightarrow \\
\quad f2\ wh(f2, f3) = (f2, f2 + f1)\ wh\ (f1, f2) = (f1, f1 + f0)\ wh\ (f0, f1) = pf\ 0 \\
\Longrightarrow \\
\quad f2\ wh\ (f2, f3) = (f2, f2 + f1)\ wh\ (f1, f2) = (f1, f1 + f0)\ wh\ (f0, f1) = (0, 1) \\
\Longrightarrow \\
\quad f2\ wh\ (f2, f3) = (f2, f2 + 1)\ wh\ (1, f2) = (1, 1 + 0) \\
\Longrightarrow \\
\quad f2\ wh(f2, f3) = (f2, f2 + 1)\ wh\ (1, f2) = (1, 1) \\
\Longrightarrow \\
\quad 1\ wh\ (1, f3) = (1, 1 + 1) \\
\Longrightarrow \\
\quad 1
\end{array}
$$

mientras que con la definición

$$
\begin{array}{l}
fib\ n \mid n \geq 2 \ = fib(n - 1)\ +\ fib(n - 2) \\
\qquad \mid n == 0 \ = 0 \\
\qquad \mid n == 1 \ = 1
\end{array}
$$

se tiene

$$
\begin{array}{l}
\quad fib\ 2 \\
\Longrightarrow
\end{array}
$$

$$
\begin{array}{ll}
 & \mathit{fib}\ 1\ +\ \mathit{fib}\ 0 \\
\Longrightarrow & \\
 & 1\ +\ \mathit{fib}0 \\
\Longrightarrow & \\
 & 1\ +\ 0 \\
\Longrightarrow & \\
 & 1
\end{array}
$$

La demostración requerida es:

$$
\begin{array}{ll}
 & pf(n+2) \\
\equiv & \{\text{ definición de } parFib\ \} \\
 & (f2, f2+f1)\ wh\ (f1, f2) = pf(n+1) \\
\equiv & \{\text{ definición de } parFib\ \} \\
 & (f2, f2+f1)\ wh\ (f1, f2) = (f1, f1+f0)\ wh\ \{(f1, f2) = pf(n+1);\ (f0, f1) = pf\ n\} \\
\equiv & \{\ f2 = f1+f0\ \} \\
 & (f1+f0,\ f2+f1)\ wh\ \{(f1, f2) = pf(n+1);\ (f0, f1) = pf\ n\} \\
\equiv & \{\text{ definición de } +:\} \\
 & (f1,\ f2) +:\ (f0,\ f1)\ wh\ \{(f1, f2) = pf(n+1);\ (f0, f1) = pf\ n\} \\
\equiv & \\
 & pf(n+1) +:\ pf\ n
\end{array}
$$

**Solución al Ejercicio 16.28** (pág. 444).–

— *Caso Base*: ($n \equiv 0$)

$$
\begin{array}{ll}
 & f_{0+m}\ \equiv\ f_m\ \cdot\ f_{0+1}\ +\ f_{m-1}\ \cdot\ f_0 \\
\equiv & \\
 & f_m\ \equiv\ f_m\ \cdot\ f_1\ +\ f_{m-1}\ \cdot\ f_0 \\
\equiv & \{\ f_0 \equiv 0; f_1 \equiv 1\ \} \\
 & \mathit{Cierto}
\end{array}
$$

— *Paso Inductivo*: ($n \geq 0$)

$$
\begin{array}{ll}
 & f_{(n+1)+m} \\
\equiv & \{\text{ asociatividad y conmutatividad de } +\ \} \\
 & f_{n+(m+1)} \\
\equiv & \{\text{ hipótesis de inducción}\ \} \\
 & f_{m+1}\ \cdot\ f_{n+1}\ +\ f_m\ \cdot\ f_n \\
\equiv & \{\text{ definición de } f_k\ \} \\
 & (f_m + f_{m-1})\ \cdot\ f_{n+1}\ +\ f_m\ \cdot\ f_n \\
\equiv & \{\text{ distributividad de } \cdot \text{ respecto a } +\ \} \\
 & \underline{f_m \cdot f_{n+1}}\ +\ f_{m-1}\ \cdot\ f_{n+1}\ +\ \underline{f_m\ \cdot\ f_n} \\
\equiv & \{\text{ distributividad de } \cdot \text{ respecto a } +\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & f_m \cdot (f_{n+1} + f_n) + f_{m-1} \cdot f_{n+1} \\
\equiv & \{ \text{definición de } f_k \} \\
 & f_m \cdot f_{n+2} + f_{m-1} \cdot f_{n+1}
\end{array}
$$

**Solución al Ejercicio 16.29** (pág. 444).– Consideremos la definición

$$long' \; xs \; n \; = \; long \; xs \; + \; n - - \text{(*)}$$

que podemos transformar según:

$$
\begin{array}{ll}
 & long' \; [\,] \; n \\
\equiv & \{ \text{desplegar } (*) \} \\
 & long \; [\,] \; + \; n \\
\equiv & \{ \text{desplegar } long \} \\
 & n
\end{array}
$$

$$
\begin{array}{ll}
 & long' \; (x : xs) \; n \\
\equiv & \{ \text{desplegar } (*) \} \\
 & long \; (x : xs) \; + \; n \\
\equiv & \{ \text{desplegar } long \} \\
 & 1 + \; long \; xs \; + \; n \\
\equiv & \{ \text{asoc. y conmut.} \} \\
 & long \; xs \; + \; (n + 1) \\
\equiv & \{ \text{plegar} \} \\
 & long' \; xs \; (n + 1)
\end{array}
$$

de donde hemos eliminado la función $long$:

$$
\begin{array}{lll}
long' \; [\,] & \quad n \; = \; n \\
long' \; (x : xs) \; n & = & long' \; xs \; (n + 1)
\end{array}
$$

que introducimos en la forma:

$$long \; xs \; = \; long' \; xs \; 0$$

La corrección es trivial.

**Solución al Ejercicio 16.30** (pág. 445).–

$$
\begin{array}{ll}
 & iteraHasta \; f \; p \; x \\
\equiv & \{ \text{desplegar } iteraHasta \} \\
 & primero \; p \; (iterate \; f \; x) \\
\equiv & \{ \text{desplegar } primero \} \\
 & primero \; p \; (x : iterate \; f \; (f \; x)) \\
\equiv &
\end{array}
$$

$x$, si $p\ x$
$primero\ p\ (iterate\ f\ (f\ x))$, en otro caso
$\equiv$ { plegar $iteraHasta$ }
$x$, si $p\ x$
$iteraHasta\ f\ p\ (f\ x)$, en otro caso

**Solución al Ejercicio 16.31** (pág. 445).– Sea la función auxiliar:

$$g\ a\ =\ (suma\ a,\ nEle\ a) \text{ -- definición}$$

Tenemos entonces:

$$media\ a\ =\ u\ `div`\ v\ \mathbf{where}\ (u, v)\ =\ g\ a$$

Además:

$g\ Vacío$
$\equiv$ { definición }
$(suma\ Vacío, nEle\ Vacío)$
$\equiv$ { instanciación }
$(0, 0)$

$g\ (Hoja\ n)$
$\equiv$ { definición }
$(suma\ (Hoja\ n), nEle\ (Hoja\ n))$
$\equiv$ { instanciación }
$(n, 1)$

$g\ (Nodo\ i\ n\ d)$
$\equiv$ { definición }
$(\ suma\ (Nodo\ i\ n\ d),\ nEle\ (Nodo\ i\ n\ d)\ )$
$\equiv$ { instanciación }
$(\ suma\ i\ +\ n\ +\ suma\ d,\ nEle\ i\ +\ 1\ +\ nEle\ d\ )$
$\equiv$ { abstracción }
$(s\ +\ n\ +\ s',\ p\ +\ 1\ +\ p')$
$\mathbf{where}\ \{(s, p)\ =\ (suma\ i,\ nEle\ i);\ (s', p')\ =\ (suma\ d,\ nEle\ d)\}$
$\equiv$ { definición }
$(s + n + s',\ p + 1 + p')\ \mathbf{where}\ \{(s, p)\ =\ g\ i;\ (s', p')\ =\ g\ d\}$

luego queda

$$\begin{array}{lll} g\ Vacío & = & (0, 0) \\ g\ (Hoja\ n) & = & (n, 1) \\ g\ (Nodo\ i\ n\ d) & = & (s + n + s',\ p + 1 + p')\ \mathbf{where}\ \begin{array}{l}(s, p)\ =\ g\ i \\ (s', p')\ =\ g\ d\end{array} \end{array}$$

**Solución al Ejercicio 16.32** (pág. 445).– Basta con demostrar la primera de las igualdades. Obsérvese que $aplana$ es ineficiente al utilizar la concatenación; la solución que se oferta es con un acumulador (el segundo argumento de $aplanaCon$). Probaremos por inducción sobre $a$ que

$$\forall\ a, r\ \centerdot\ a\ ::\ Árbol\ [\alpha],\ r\ ::\ [\alpha]\ \centerdot$$

$$aplanaCon\ a\ r \quad = \quad aplana\ a +\!\!+ r$$

— *Caso Base*: ($a \equiv\ Vacío$)

$aplanaCon\ Vacío\ r\ =\ aplana\ Vacío +\!\!+ r$
$\equiv\equiv$
$r\ =\ [\,] +\!\!+ r$

— *Paso Inductivo*: ($a \equiv\ Nodo\ i\ x\ d$)

$aplanaCon\ (Nodo\ i\ x\ d)\ r\ =\ aplana\ (Nodo\ i\ x\ d) +\!\!+ r$
$\equiv\equiv$
$aplanaCon\ i\ (x : aplanaCon\ d\ r)\ =\ aplana\ i +\!\!+ x : aplana\ d +\!\!+ r$
$\equiv\equiv$ { hipótesis de inducción dos veces }
$aplana\ i +\!\!+ (x : aplana\ d +\!\!+ r)\ =\ aplana\ i +\!\!+ x : aplana\ d +\!\!+ r$
$\equiv\equiv$
$Cierto$

**Solución al Ejercicio 16.33** (pág. 446).–

1. La función $nClaves$ es:

$$nClaves\ a\ =\ pliegaA\ f\ 0\ a\ \mathbf{where}\ f\ \_\ u\ v\ =\ u\ +\ v\ +\ 1$$

2. En el caso de que el árbol sea $Vacío$

$nClaves\ Vacío$
$\equiv\equiv$ { instanciación }
$pliegaA\ f\ 0\ Vacío\ \mathbf{where}\ f\ \_\ u\ v\ =\ u\ +\ v\ +\ 1$
$\equiv\equiv$ { 1) $pliegaA$ }
$0$

mientras que si el árbol es de la forma $Nodo\ i\ x\ d$:

$nClaves\ (Nodo\ i\ x\ d)$
$\equiv\equiv$ { instanciación }
$pliegaA\ f\ 0\ (Nodo\ i\ x\ d)\ \mathbf{where}\ f\ \_\ u\ v\ =\ u\ +\ v\ +\ 1$
$\equiv\equiv$ { 2)$pliegaA$ }
$f\ x\ (pliegaA\ f\ 0\ i)\ (pliegaA\ f\ 0\ d)\ \mathbf{where}\ f\ \_\ u\ v\ =\ u\ +\ v\ +\ 1$
$\equiv\equiv$ { cualificador **where** }

$$
\begin{array}{ll}
& (pliegaA\ f\ 0\ i)\ +\ (pliegaA\ f\ 0\ d)\ +\ 1 \\
\equiv & \{\text{ plegar } nClaves \} \\
& nClaves\ i\ +\ nClaves\ d\ +\ 1
\end{array}
$$

Obsérvese que en la demostración no se utiliza la hipótesis de inducción.

3. Se considera una función auxiliar

$$
\begin{array}{l}
equiAux\ ::\ Árbol\ a\ \rightarrow\ (Int,\ Bool) \\
equiAux\ a\ =\ (nClaves\ a,\ equi\ a) \text{ -- abstracción}
\end{array}
$$

de donde

$$
\begin{array}{ll}
& equiAux\ Vacío \\
\equiv & \{\text{ abstracción }\} \\
& (nClaves\ Vacío,\ equi\ Vacío) \\
\equiv & \{\text{ apartado anterior, (1)}equi\ \} \\
& (0,\ True)
\end{array}
$$

$$
\begin{array}{ll}
& equiAux\ (Nodo\ i\ x\ d) \\
\equiv & \{\text{ abstracción }\} \\
& (nClaves\ (Nodo\ i\ x\ d),\ equi\ (Nodo\ i\ x\ d)) \\
\equiv & \{\text{ apartado anterior, (2)}equi\ \} \\
& (\ nClaves\ i\ +\ nClaves\ d\ +\ 1,\ equi\ i\ \&\&\ equi\ d\ \&\&\ abs\ (ni - nd) \leq\ 1\ ) \\
& \textbf{where}\ ni\ =\ nClaves\ i;\ nd\ =\ nClaves\ d \\
\equiv & \{\text{ abstracción }\} \\
& (ni\ +\ nd\ +\ 1,\ fi\ \&\&\ fd\ \&\&\ abs\ (ni - nd) \leq (1) \\
& \textbf{where}\ (ni, fi)\ =\ equiAux\ i;\ (nd, fd)\ =\ equiAux\ d
\end{array}
$$

y finalmente:

$$
\begin{array}{l}
equi\ a\ =\ f\ \textbf{where}\ (\_, f)\ =\ equiAux\ a \\
\\
\begin{array}{ll}
equiAux\ Vacío & = (0,\ True) \\
equiAux\ (Nodo\ i\ x\ d) & = (ni\ +\ nd\ +\ 1,\ fi\ \&\&\ fd\ \&\&\ abs\ (ni - nd) \leq 1) \\
& \quad \textbf{where}\ (ni, fi)\ =\ equiAux\ i \\
& \qquad\quad (nd, fd)\ =\ equiAux\ d
\end{array}
\end{array}
$$

4. Se puede hacer de la siguiente forma:

$$
\begin{array}{l}
equi\ a\ =\ f\ \textbf{where} \\
\quad (\_, f) \qquad\qquad\qquad\qquad = pliegaA\ g\ (0,\ True)\ a \\
\quad g\ (n, f)\ (ni, fi)\ (nd, fd)\ = \\
\qquad (ni\ +\ nd\ +\ 1,\ f\ \&\&\ fi\ \&\&\ fd\ \&\&\ abs\ (ni - nd) \leq 1)
\end{array}
$$

5. La función *aÁrbol* se define mediante:

$$
\begin{array}{l}
aÁrbol :: [a] \to Árbol\ a \\
aÁrbol\ [\,] \quad = Vacío \\
aÁrbol\ (x : xs) = Nodo\ (aÁrbol\ ys)\ x\ (aÁrbol\ zs) \\
\quad \textbf{where}\ (ys, zs) = splitAt\ (length\ xs\ `div`\ 2)\ xs
\end{array}
$$

Hay que demostrar

$$\forall\, xs \;\centerdot\; xs :: [a] \;\centerdot\; \qquad aLista\ .\ aÁrbol\ xs = xs$$

donde

$$
\begin{array}{l}
aLista\ Vacío \quad = [\,] \\
aLista\ (Nodo\ i\ x\ d) = x\ :\ aLista\ i +\!\!+ aLista\ d
\end{array}
$$

En efecto, procediendo por inducción sobre $xs$:

— *Caso Base*:

$$
\begin{array}{ll}
& aLista\ (aÁrbol\ [\,]) = [\,] \\
\equiv & \\
& aLista\ Vacío = [\,] \\
\equiv & \{\,(1) aLista\,\} \\
& Cierto
\end{array}
$$

— *Paso Inductivo*:

$$
\begin{array}{ll}
& aLista\ (aÁrbol\ (x : xs)) \\
\equiv & \{\,(2) aÁrbol\,\} \\
& aLista\ (\ Nodo\ (aÁrbol\ ys)\ x\ (aÁrbol\ zs)\ ) \\
& \textbf{where}\ (ys, zs) = splitAt\ (length\ xs\ `div`\ 2)\ xs \\
\equiv & \{\,(2) aLista\,\} \\
& x : aLista\ (aÁrbol\ ys) +\!\!+ aLista\ (aÁrbol\ zs) \\
& \textbf{where}\ (ys, zs) = splitAt\ (length\ xs\ `div`\ 2)\ xs \\
\equiv & \{\ \text{hipótesis de inducción}\,\} \\
& x : (ys +\!\!+ zs) \\
& \textbf{where}\ (ys, zs) = splitAt\ (length\ xs\ `div`\ 2)\ xs \\
\equiv & \{\ \text{propiedades de } splitAt\,\} \\
& x : xs
\end{array}
$$

Para las pruebas se pueden utilizar:

$$pru1 \;=\; aÁrbol\,[1,2,3,4,5,6,7]$$
$$pru2 \;=\; [\;equi\;(aÁrbol\,[1..x]) \mid x \leftarrow\; [1..]\;]$$

**Solución al Ejercicio 16.34** (pág. 447).–

1. Asignando tipos a las variables y al resultado de la primera ecuación:

$$f \;::\; \alpha,\; z \;::\; \beta,\; pliegaA \;::\; \alpha \;\to\; \beta \;\to\; Árbol\;\tau \;\to\; \beta$$

y para la segunda:

$$i \;::\; Árbol\;\tau,\; x \;::\; \tau,\; d \;::\; Árbol\;\tau,$$
$$f\;x\;(pliegaA\;f\;z\;i)(pliegaA\;f\;z\;d) \;::\; \beta$$

con lo cual:

$$\begin{array}{ll} & f\;x\;(pliegaA\;f\;z\;i)\;(pliegaA\;f\;z\;d) \;::\; \beta \\ (a) & \vdash\{\;\} \\ & \exists\,\gamma \mid (pliegaA\;f\;z\;d) \;::\; \gamma \;\wedge\; f\;x\;(pliegaA\;f\;z\;i) \;::\; \gamma \;\to\; \beta \end{array}$$

$$\begin{array}{ll} & f\;x\;(pliegaA\;f\;z\;i) \;::\; \gamma \;\to\; \beta \\ (a) & \vdash\{\;\} \\ & \exists\,\delta \mid (pliegaA\;f\;z\;i) \;::\; \delta \;\wedge\; f\;x \;::\; \delta \;\to\; \gamma \;\to\; \beta \end{array}$$

$$\begin{array}{ll} & f\;x \;::\; \delta \;\to\; \gamma \;\to\; \beta \\ (a) & \vdash\{\;\} \\ & \exists\,\tau \mid x \;::\; \tau \;\wedge\; f \;::\; \tau \;\to\; \delta \;\to\; \gamma \;\to\; \beta \end{array}$$

$$\begin{array}{ll} & f \;::\; \alpha \;\wedge\; f \;::\; \tau \;\to\; \delta \;\to\; \gamma \;\to\; \beta \\ (i) & \vdash\{\;\} \\ & \alpha \;=\; \tau \;\to\; \delta \;\to\; \gamma \;\to\; \beta \end{array}$$

$$\begin{array}{ll} & pliegaA\;f\;z\;d \;::\; \gamma \;\wedge\; d \;::\; Árbol\;\tau \\ (a') & \vdash\{\;\} \\ & pliegaA\;f\;z \;::\; Árbol\;\tau \;\to\; \gamma \end{array}$$

siendo $(a')$ la regla obtenida al aplicar la regla $(a)$ y después la $(i)$:

$$\begin{array}{ll} & pliegaA\;f\;z\;i \;::\; \delta \;\wedge\; d \;::\; Árbol\;\tau \\ (a') & \vdash\{\;\} \\ & pliegaA\;f\;z \;::\; Árbol\;\tau \;\to\; \delta \end{array}$$

$$
\begin{array}{l}
\quad pliegaA\ f\ z\ ::\ \acute{A}rbol\ \tau\ \rightarrow\ \gamma\ \wedge\ pliegaA\ f\ z\ ::\ \acute{A}rbol\ \tau\ \rightarrow\ \delta \\
(i) \vdash \{\ \} \\
\quad \gamma\ =\ \delta
\end{array}
$$

$$
\begin{array}{l}
\quad pliegaA\ f\ z\ ::\ \acute{A}rbol\ \tau\ \rightarrow\ \delta\ \wedge\ z\ ::\ \beta \\
(a') \vdash\!\!\!\!\!\!\{\ \} \\
\quad pliegaA\ f\ ::\ \beta\ \rightarrow\ \acute{A}rbol\ \tau\ \rightarrow\ \delta
\end{array}
$$

$$
\begin{array}{l}
\quad pliegaA\ f\ ::\ \beta\ \rightarrow\ \acute{A}rbol\ \tau\ \rightarrow\ \delta\ \wedge\ f\ ::\ \alpha \\
(a') \vdash\!\!\!\!\!\!\{\ \} \\
\quad pliegaA\ ::\ \alpha\ \rightarrow\ \beta\ \rightarrow\ \acute{A}rbol\ \tau\ \rightarrow\ \delta
\end{array}
$$

$$
\begin{array}{l}
\quad pliegaA\ ::\ \alpha\ \rightarrow\ \beta\ \rightarrow\ \acute{A}rbol\ \tau\ \rightarrow\ \delta\ \wedge\ pliegaA\ ::\ \alpha\ \rightarrow\ \beta\ \rightarrow\ \acute{A}rbol\ \tau\ \rightarrow\ \beta \\
(i) \vdash \{\ \} \\
\quad \delta\ =\ \beta
\end{array}
$$

con lo cual tendremos que:

$$pliegaA\ ::\ (\tau\ \rightarrow\ \beta\ \rightarrow\ \beta\ \rightarrow\ \beta)\ \rightarrow\ \beta\ \rightarrow\ \acute{A}rbol\ \tau\ \rightarrow\ \beta$$

2. La función $prof$ es:

$$prof\ a\ =\ pliegaA\ f\ 0\ a\ \mathbf{where}\ f\ \_\ u\ v\ =\ max\ u\ v\ +\ 1$$

3. En el caso de que el árbol sea $Vacío$

$$
\begin{array}{ll}
 & prof\ Vacío \\
\equiv & \{\ \text{instanciación}\ \} \\
 & pliegaA\ f\ 0\ Vacío\ \mathbf{where}\ f\ \_\ u\ v\ =\ max\ u\ v\ +\ 1 \\
\equiv & \{\ (1)pliegaA\ \} \\
 & 0
\end{array}
$$

mientras que si el árbol es de la forma $Nodo\ i\ x\ d$:

$$
\begin{array}{ll}
 & prof\ (Nodo\ i\ x\ d) \\
\equiv & \{\ \text{instanciación}\ \} \\
 & pliegaA\ f\ 0\ (Nodo\ i\ x\ d)\ \mathbf{where}\ f\ \_\ u\ v\ =\ max\ u\ v\ +\ 1 \\
\equiv & \{\ (2)pliegaA\ \} \\
 & f\ x\ (pliegaA\ f\ 0\ i)\ (pliegaA\ f\ 0\ d)\ \mathbf{where}\ f\ \_\ u\ v\ =\ max\ u\ v\ +\ 1 \\
\equiv & \{\ \text{cualificador}\ \mathbf{where}\ \} \\
 & max\ (pliegaA\ f\ 0\ i)\ (pliegaA\ f\ 0\ d)\ +\ 1 \\
\equiv & \{\ \text{plegar}\ prof\ \}
\end{array}
$$

$max\ (prof\ i)\ (prof\ d)\ +\ 1$

Obsérvese que la demostración no utiliza la hipótesis de inducción.

4. Se considera una función auxiliar

$$\begin{array}{l} balAux\ ::\ Árbol\ a\ \rightarrow\ (Int,\ Bool) \\ balAux\ a\ =\ (prof\ a,\ bal\ a) \text{ -- abstracción} \end{array}$$

de donde

$$\begin{array}{ll} & balAux\ Vacío \\ \equiv & \{\text{ abstracción }\} \\ & (prof\ Vacío,\ bal\ Vacío) \\ \equiv & \{\text{ apartado anterior, (1)}bal\ \} \\ & (0,\ True) \end{array}$$

$$\begin{array}{ll} & balAux\ (Nodo\ i\ x\ d) \\ \equiv & \{\text{ abstracción }\} \\ & (prof\ (Nodo\ i\ x\ d),\ bal\ (Nodo\ i\ x\ d)) \\ \equiv & \{\text{ apartado anterior, (2)}bal\ \} \\ & (\ max\ (prof\ i)\ (prof\ d)\ +\ 1,\ bal\ i\ \&\&\ bal\ d\ \&\&\ abs\ (pi\ -\ pd) \leq 1\ ) \\ & \textbf{where}\ \{\ pi\ =\ prof\ i;\ pd\ =\ prof\ d\ \} \\ \equiv & \{\text{ abstracción }\} \\ & (max\ pi\ pd\ +\ 1,\ fi\ \&\&\ fd\ \&\&\ abs(pi\ -\ pd) \leq 1) \\ & \textbf{where}\ \{(pi, fi)\ =\ balAux\ i;\ (pd, fd)\ =\ balAux\ d\} \end{array}$$

y finalmente:

$$bal\ a\ =\ f\ \textbf{where}\ (\_, f)\ =\ balAux\ a$$

$$\begin{array}{ll} balAux\ Vacío & = (0,\ True) \\ balAux\ (Nodo\ i\ x\ d) & = (max\ pi\ pd\ +\ 1,\ fi\ \&\&\ fd\ \&\&\ abs(pi\ -\ pd) \leq 1) \\ & \quad \textbf{where}\ (pi, fi)\ =\ balAux\ i \\ & \qquad\quad (pd, fd)\ =\ balAux\ d \end{array}$$

5. Se puede hacer de la siguiente forma:

$$\begin{array}{l} bal\ a\ =\ f\ \textbf{where} \\ \quad (\_, f)\ =\ pliegaA\ g\ (0,\ True)\ a \\ \quad g\ (n, f)\ (pi, fi)\ (pd, fd)\ = \\ \qquad (max\ pi\ pd\ +\ 1,\ f\ \&\&\ fi\ \&\&\ fd\ \&\&\ abs(pi\ -\ pd) \leq 1) \end{array}$$

**Solución al Ejercicio 16.35** (pág. 447).–

1. Se consideran las siguientes definiciones:

$$\begin{array}{lcl} nEle & = & pliegaP\ (\lambda\, x\ y\ \rightarrow\ 1 + y)\ 0 \\ suma & = & pliegaP\ (+)\ 0 \\ aLista & = & pliegaP\ (:)\ [\,] \end{array}$$

Para la $suma'$ conviene modificar la función $pliegaP$

$$\begin{array}{lcl} pliegaP1 & :: & (a \rightarrow\ b \rightarrow\ b)\ \rightarrow\ Pila\ a\ \rightarrow\ b \\ pliegaP1\ f\ (Cima\ x\ Vacía) & = & x \\ pliegaP1\ f\ (Cima\ x\ p) & = & f\ x\ (pliegaP1\ f\ p) \\ suma' & = & pliegaP1\ (+) \end{array}$$

Probemos, por inducción estructural

$$\forall\, p\ \centerdot p\ ::\ Pila\ Int\ \centerdot\ suma\ p\ =\ foldr\ (+)\ 0\ (aLista\ p)$$

— *Caso Base*:

$$\begin{array}{ll} & suma\ Vacía\ =\ foldr\ (+)\ 0\ (aLista\ Vacía) \\ \equiv & \{\,\text{def. } suma \text{ y } aLista\,\} \\ & 0\ =\ foldr\ (+)\ 0\ [\,] \\ \equiv & \{\,\text{def. } foldr\,\} \\ & 0\ =\ 0 \end{array}$$

— *Paso Inductivo*:

$$\begin{array}{ll} & suma\ (Cima\ x\ p)\ =\ foldr\ (+)\ 0\ (aLista\ (Cima\ x\ p)) \\ \equiv & \{\,\text{def. } suma \text{ y } aLista\,\} \\ & x\ +\ suma\ p\ =\ foldr\ (+)\ 0\ (x : aLista\ p) \\ \equiv & \{\,\text{def. } foldr\,\} \\ & x\ +\ suma\ p\ =\ x\ +\ foldr\ (+)\ 0\ (aLista\ p) \\ \Leftarrow & \{\ (is)\,\} \\ & HI \end{array}$$

2. Probaremos el primero

$$\begin{array}{ll} & nEle\ (Cima\ x\ p) \\ \equiv & \{\,\text{def}\,\} \\ & pliegaP\ (\lambda\, x\ y\ \rightarrow\ 1 + y)\ 0\ (Cima\ x\ p) \end{array}$$

$$
\begin{array}{ll}
\equiv & \{\,(1) pliegaP\,\} \\
& (\lambda\, x\; y\; \rightarrow\; 1 + y)\; x\; (pliegaP\; (\lambda\, x\; y\; \rightarrow\; 1 + y)\; 0\; p) \\
\equiv & \{\,\text{aplicación}\,\} \\
& 1\; +\; (pliegaP\; (\lambda\, x\; y\; \rightarrow\; 1 + y)\; 0\; p) \\
\equiv & \{\,\text{def. de } nEle\,\} \\
& 1\; +\; nEle\; p
\end{array}
$$

y los otros predicados se prueban igual.

3. Introducimos la definición auxiliar

$$sumYnEle\; p\; =\; (suma\; p,\; nEle\; p)$$

de forma que

$$media\; p\; =\; s\; /\; n\; \mathbf{where}\; (s, n)\; =\; sumYnEle\; p$$

Ahora bien

$$
\begin{array}{ll}
& sumYnEle\; Vacía \\
\equiv & \{\,\text{instanciación}\,\} \\
& (suma\; Vacía,\; nEle\; Vacía) \\
\equiv & \{\,\text{def. de } suma \text{ y } nEle\,\} \\
& (\; pliegaP\; (+)\; 0\; Vacía,\; pliegaP\; (\lambda\, x\; y\; \rightarrow\; 1 + y)\; 0\; Vacía\;) \\
\equiv & \{\,pliegaP\,\} \\
& (0,\; 0)
\end{array}
$$

$$
\begin{array}{ll}
& sumYnEle\; (Cima\; x\; p) \\
\equiv & \{\,\text{instanciación}\,\} \\
& (suma\; (Cima\; x\; p),\; nEle\; (Cima\; x\; p)) \\
\equiv & \{\,\text{prop. del apartado 2}\,\} \\
& (x\; +\; suma\; p,\; 1\; +\; nEle\; p) \\
\equiv & \{\,\text{abstracción}\,\} \\
& (x + s, 1 + p)\; \mathbf{where}\; (s, p) = (suma\; p,\; nEle\; p) \\
\equiv & \{\,\text{plegar}\,\} \\
& (x + s, 1 + p)\; \mathbf{where}\; (s, p) = sumYnEle\; p
\end{array}
$$

y finalmente:

$$
\begin{array}{lll}
sumYnEle\; Vacía & = & (0,\; 0) \\
sumYnEle\; (Cima\; x\; p) & = & (x + s, 1 + p)\; \mathbf{where}\; (s, p) = sumYnEle\; p
\end{array}
$$

4. Para definir $aPila$ necesitamos una función auxiliar con un acumulador

$$\begin{array}{l} aPila2 :: [a] \to Pila\ a \to Pila\ a \\ aPila2\ [\,]\ \ \ \ \ \ p = p \\ aPila2\ (x : xs)\ p = aPila2\ xs\ (Cima\ x\ p) \end{array}$$

para entonces definir:

$$aPila\ xs = aPila2\ xs\ Vacía$$

Probaremos

$$(*) \qquad \begin{array}{l} \forall\, xs, p \,\centerdot\, xs :: [a], p :: Pila\ a \,\centerdot \\ \qquad aLista\ (aPila2\ xs\ p) = inv\ xs +\!\!+ aLista\ p \end{array}$$

ya que de aquí tendremos

$$\begin{array}{ll} & aLista\ (aPila2\ xs\ Vacía) = inv\ xs +\!\!+ aLista\ Vacía \\ \equiv & \{\ aPila,\ aLista\ \} \\ & aLista\ (aPila\ xs) = inv\ xs +\!\!+ [\,] \\ \equiv & \{\ [\,] \text{ unidad de } +\!\!+\ \} \\ & aLista\ (aPila\ xs) = inv\ xs \end{array}$$

que es lo que se pedía. Veamos $(*)$ por inducción sobre $xs$; es decir, probaremos

$$\begin{array}{l} \forall\, xs \,\centerdot\, xs :: [a] \,\centerdot \\ \qquad \forall\, p \,\centerdot\, p :: Pila\ a \,\centerdot\, aLista\ (aPila2\ xs\ p) = inv\ xs +\!\!+ aLista\ p \end{array}$$

— *Caso Base*: se tiene, $\forall\, p \,\centerdot\, p :: Pila\ a$,

$$\begin{array}{ll} & aLista\ (aPila2\ [\,]\ p) = inv\ [\,] +\!\!+ aLista\ p \\ \equiv & \{\ (1)aPila2,\ (1)inv\ \} \\ & aLista\ p = [\,] +\!\!+ aLista\ p \\ \equiv & \{\ (1)+\!\!+\ \} \\ & Cierto \end{array}$$

— *Paso Inductivo*: la hipótesis de inducción es

$$\forall\, p \,\centerdot\, p :: Pila\ a \,\centerdot\, aLista\ (aPila2\ xs\ p) = inv\ xs +\!\!+ aLista\ p$$

Entonces, $\forall\, p \,\centerdot\, p :: Pila\ a$,

$$
\begin{array}{ll}
& aLista\ (aPila2\ (x : xs)\ p) \ = \ inv\ (x : xs) +\!\!+ aLista\ p \\
\equiv & \{\ aPila2,\ inv\ \} \\
& aLista\ (aPila2\ xs\ (Cima\ x\ p)) \ = \ (inv\ xs +\!\!+ [x]) +\!\!+ aLista\ p \\
\equiv & \{\ \text{def. } (:),\ \text{asociativ. de } +\!\!+\ \} \\
& aLista\ (aPila2\ xs\ (Cima\ x\ p)) \ = \ inv\ xs +\!\!+ (x : aLista\ p) \\
\equiv & \{\ aLista\ \} \\
& aLista\ (aPila2\ xs\ (Cima\ x\ p)) \ = \ inv\ xs +\!\!+ aLista\ (Cima\ x\ p) \\
\Leftarrow & \{\ \text{obsérvese que la } HI \text{ es } \forall p;\ \text{en particular, para } p \equiv Cima\ x\ p\ \} \\
& HI
\end{array}
$$

**Solución al Ejercicio 16.37** (pág. 449).– En primer lugar se dará la semántica de las construcciones condicionales; para una sentencia condicional $if\_then\_else$:

$$Cond2 \ = \ \mathbf{if}\ b\ \mathbf{then}\ S1\ \mathbf{else}\ S2$$

es fácil dar el valor de $[\![\ Cond2\ ]\!]\ \rho$ para cada entorno $\rho$:

$$[\![\ Cond2\ ]\!]\ \rho \ = \ \mathbf{if}\ [\![\ b\ ]\!]\ \rho\ \mathbf{then}\ [\![\ S1\ ]\!]\ \rho\ \mathbf{else}\ [\![\ S2\ ]\!]\ \rho$$

mientras que para una sentencia $if\_then$:

$$Cond1 \ = \ \mathbf{if}\ b\ \mathbf{then}\ S$$

se puede aprovechar el resultado obtenido para $if\_then\_else$ considerando $Nada$ como parte $else$:

$$[\![\ Cond1\ ]\!]\ \rho \ = \ \mathbf{if}\ [\![\ b\ ]\!]\ \rho\ \mathbf{then}\ [\![\ S\ ]\!]\ \rho\ \mathbf{else}\ \rho$$

Ahora ya estamos en disposición de dar la semántica del programa $max3$ en cuestión, que consta de la composición secuencial de dos sentencias condicionales $if\_then$:

$$
\begin{array}{l}
max3\ ::\ (Int, Int, Int)\ \rightarrow\ (Int, Int, Int) \\
max3\ =\ c2\ .\ c1\ \mathbf{where} \\
\quad c1\ (x, y, z)\ =\ \mathbf{if}\ y\ >\ x\ \mathbf{then}\ (y, y, z)\ \mathbf{else}\ (x, y, z) \\
\quad c2\ (x, y, z)\ =\ \mathbf{if}\ z\ >\ x\ \mathbf{then}\ (z, y, z)\ \mathbf{else}\ (x, y, z)
\end{array}
$$

**Solución al Ejercicio 16.38** (pág. 449).– La semántica del programa $ordena$ en cuestión vendrá dada por la semántica de un bucle cuyo cuerpo es la composición secuencial de tres sentencias condicionales $if\_then$:

$$
\begin{array}{l}
ordena\ ::\ (Int, Int, Int, Int)\ \rightarrow\ (Int, Int, Int, Int) \\
ordena\ (a, b, c, d)\ =\ \mathbf{if}\ (a\ >\ b)\ ||\ (b\ >\ c)\ ||\ (c\ >\ d)\ \mathbf{then} \\
\qquad\qquad\qquad\qquad (ordena\ .\ c3\ .\ c2\ .\ c1)\ (a, b, c, d) \\
\qquad\qquad\qquad \mathbf{else} \\
\qquad\qquad\qquad\quad (a, b, c, d) \\
\quad \mathbf{where} \\
\qquad c1\ (a, b, c, d)\ =\ \mathbf{if}\ a\ >\ b\ \mathbf{then}\ (b, a, c, d)\ \mathbf{else}\ (a, b, c, d) \\
\qquad c2\ (a, b, c, d)\ =\ \mathbf{if}\ b\ >\ c\ \mathbf{then}\ (a, c, b, d)\ \mathbf{else}\ (a, b, c, d) \\
\qquad c3\ (a, b, c, d)\ =\ \mathbf{if}\ c\ >\ d\ \mathbf{then}\ (a, b, d, c)\ \mathbf{else}\ (a, b, c, d)
\end{array}
$$

Finalmente se da una definición con guardas de la función $ordena$ en HASKELL a efectos comparativos:

$$\begin{array}{l} ordena\ (a,\ b,\ c,\ d)\ = \\ \quad \mid\ a\ >\ b \qquad\qquad = ordena\ (b,\ a,\ c,\ d) \\ \quad \mid\ b\ >\ c \qquad\qquad = ordena\ (a,\ c,\ b,\ d) \\ \quad \mid\ c\ >\ d \qquad\qquad = ordena\ (a,\ b,\ d,\ c) \\ \quad \mid\ otherwise \qquad = (a,\ b,\ c,\ d) \end{array}$$

**Solución al Ejercicio 16.39** (pág. 449).– En la primera equivalencia se exige que $x$ no aparezca en $e'$, pues si $x$ apareciera en $e'$ no se tendría la equivalencia; por ejemplo,

$$[\![\ x\ :=\ 0;\ x\ :=\ x\ +1\ ]\!] \quad \not\equiv \quad [\![\ x\ :=\ x\ +\ 1\ ]\!]$$

y la demostración de la equivalencia es

$$\begin{array}{ll} & [\![\ x := e;\ x := e'\ ]\!]\ \rho \\ \equiv & \\ & [\![\ x := e'\ ]\!]\ \rho\{x := e\} \\ \equiv & \\ & \rho\{x := e\}\{x := e'\} \\ \equiv & \ \{\ \text{si } x \text{ no aparece en } e'\ \} \\ & \rho\{x := e'\} \\ \equiv & \\ & [\![\ x := e'\ ]\!]\ \rho \end{array}$$

En la segunda equivalencia hay que imponer que $m \not\equiv\ y$, pues en el caso de que $m \equiv\ y$ las sentencias no serían equivalentes; es decir,

$$[\![\ m := x;\ \mathbf{if}\ \mathit{False}\ \mathbf{then}\ m := y\ ]\!] \not\equiv [\![\ \mathbf{if}\ \ y > x\ \ \mathbf{then}\ \ m := y\ \ \mathbf{else}\ \ m := x\ ]\!]$$

y la demostración de la equivalencia es

$$\begin{array}{ll} & [\![\ m := x;\ \mathbf{if}\ y > m\ \mathbf{then}\ m := y\ ]\!]\ \rho \\ \equiv & \\ & [\![\ \mathbf{if}\ y > m\ \mathbf{then}\ m := y\ ]\!]\ \rho\ \{m := x\} \\ \equiv & \\ & \mathbf{if}\ \ [\![\ y > m\ ]\!]\rho\{m := x\}\ \ \mathbf{then}\ \ [\![\ m := y\ ]\!]\rho\{m := x\}\ \ \mathbf{else}\ \ \rho\{m := x\} \\ \equiv & \\ & \mathbf{if}\ \ \rho\{m := x\}y\ >\ \rho\{m := x\}m\ \ \mathbf{then}\ \ [\![\ m := y\ ]\!]\rho\{m := x\}\ \ \mathbf{else}\ \ \rho\{m := x\} \\ \equiv & \ \{\ \text{si } x \not\equiv y, \text{ por la prop. anterior: } [\![\ m := x; m := y\ ]\!]\rho = [\![\ m := y\ ]\!]\rho\ \} \\ & \mathbf{if}\ \ \rho\ y\ >\ \rho\ x\ \ \mathbf{then}\ \ \rho\{m := x\}\ \ \mathbf{else}\ \ \rho\{m := x\} \\ \equiv & \\ & \mathbf{if}\ \ [\![\ y > x\ ]\!]\ \rho\ \ \mathbf{then}\ \ \rho\{m := y\}\ \ \mathbf{else}\ \ \rho\{m := x\} \\ \equiv & \\ & [\![\ \mathbf{if}\ \ y > x\ \ \mathbf{then}\ \ m := y\ \ \mathbf{else}\ \ m := x\ ]\!]\ \rho \end{array}$$

## 21.13. INTRODUCCIÓN AL $\lambda$–CÁLCULO

**Solución al Ejercicio 17.42** (pág. 491).– Para la tercera, tenemos la ecuación

$$sub\ (f, g)\ x\ =\ f\ x\ (g\ x)$$

Sean las asignaciones iniciales $f\ ::\ \alpha$, $g\ ::\ \beta$, $x\ ::\ \gamma$, $f\ x\ (g\ x)\ ::\ \delta$; entonces podemos deducir que $sub\ ::\ (\alpha, \beta) \rightarrow\ \gamma\ \rightarrow\ \delta$, con lo cual

$$\begin{array}{l} \quad f\ x\ (g\ x)\ ::\ \delta \\ (a) \vdash\{\ \} \\ \quad \exists\, \tau\ \mid\ (g\ x\ ::\ \tau)\ \wedge\ (f\ x\ ::\ \tau\ \rightarrow\ \delta) \end{array}$$

$$\begin{array}{l} \quad g\ x\ ::\ \tau \\ (a) \vdash\{\ \} \\ \quad \exists\, \gamma\ \mid\ (x\ ::\ \gamma)\ \wedge\ (g\ ::\ \gamma\ \rightarrow\ \tau) \end{array}$$

$$\begin{array}{l} \quad (g\ ::\ \beta)\ \wedge\ (g\ ::\ \gamma\ \rightarrow\ \tau) \\ (i) \vdash\{\ \} \\ \quad \beta\ =\ (\gamma\ \rightarrow\ \tau) \end{array}$$

$$\begin{array}{l} \quad f\ x\ ::\ \tau\ \rightarrow\ \delta \\ (a) \vdash\{\ \} \\ \quad \exists\, \gamma\ \mid\ (x\ ::\ \gamma)\ \wedge\ (f\ ::\ \gamma\ \rightarrow (\tau\ \rightarrow\ \delta)) \end{array}$$

$$\begin{array}{l} \quad (f\ ::\ \alpha)\ \wedge\ (f\ ::\ \gamma\ \rightarrow\ (\tau\ \rightarrow\ \delta)) \\ (i) \vdash\{\ \} \\ \quad \alpha\ =\ \gamma\ \rightarrow\ (\tau\ \rightarrow\ \delta) \end{array}$$

$$sub\ ::\ (\gamma\ \rightarrow\ (\tau\ \rightarrow\ \delta), \gamma\ \rightarrow\ \tau)\ \rightarrow\ \gamma\ \rightarrow\ \delta$$

Ahora le toca el turno a la primera. Escribamos la ecuación en la forma

$$omega\ y\ f\ =\ f\ (f\ y)$$

Sean las asignaciones de tipos iniciales $y\ ::\ \alpha$, $f\ ::\ \beta$, $f\ (f\ y)\ ::\ \gamma$; entonces podemos deducir que $omega\ ::\ \alpha\ \rightarrow\ \beta \rightarrow\ \gamma$, con lo cual

$$\begin{array}{l} \quad f\ (f\ y)\ ::\ \gamma \\ (a) \vdash\{\ \} \\ \quad \exists\, \delta\ \mid\ (f\ y\ ::\ \delta)\ \wedge\ (f\ ::\ \delta\ \rightarrow\ \gamma) \end{array}$$

$$\begin{array}{l} \quad (f\ ::\ \beta)\ \wedge\ (f\ ::\ \delta\ \rightarrow\ \gamma) \\ (i) \vdash\{\ \} \\ \quad \beta\ =\ (\delta\ \rightarrow\ \gamma) \end{array}$$

$$
\begin{array}{l}
\quad f\ y\ ::\ \delta \\
(a) \vdash \{\ \} \\
\quad \exists\, \alpha\ |\ (y\ ::\ \alpha)\ \wedge\ (f\ ::\ \alpha \rightarrow \delta)
\end{array}
$$

$$
\begin{array}{l}
\quad (f\ ::\ \alpha \rightarrow \delta)\ \wedge\ (f\ ::\ \delta \rightarrow \gamma) \\
(i) \vdash \{\ \} \\
\quad (\alpha \rightarrow \delta)\ =\ (\delta \rightarrow \gamma)
\end{array}
$$

$$
\begin{array}{l}
\quad (\alpha \rightarrow \delta)\ =\ (\delta \rightarrow \gamma) \\
(i) \vdash \{\ \} \\
\quad (\alpha\ =\ \delta)\ \wedge\ (\delta\ =\ \gamma)
\end{array}
$$

$$omega\ ::\ \gamma \rightarrow (\gamma \rightarrow \gamma) \rightarrow \gamma$$

Veamos ahora la cuarta. Escribamos la ecuación en la forma

$$g\ x\ z\ =\ f\ ((f\ x)\ z)$$

Sean las asignaciones de tipos iniciales $x\ ::\ \alpha$, $z\ ::\ \beta$, $f\ ((f\ x)\ z)\ ::\ \gamma$; entonces podemos deducir que $g\ ::\ \alpha \rightarrow \beta \rightarrow \gamma$, con lo cual

$$
\begin{array}{l}
\quad f\ ((f\ x)\ z)\ ::\ \gamma \\
(a) \vdash \{\ \} \\
\quad \exists\, \delta\ |\ ((f\ x)\ z\ ::\ \delta)\ \wedge\ (f\ ::\ \delta \rightarrow \gamma)
\end{array}
$$

$$
\begin{array}{l}
\quad (f\ x)\ z\ ::\ \delta \\
(a) \vdash \{\ \} \\
\quad \exists\, \beta\ |\ (z\ ::\ \beta)\ \wedge\ (f\ x\ ::\ \beta \rightarrow \delta)
\end{array}
$$

$$
\begin{array}{l}
\quad f\ x\ ::\ \beta \rightarrow \delta \\
(a) \vdash \{\ \} \\
\quad \exists\, \alpha\ |\ (x\ ::\ \alpha)\ \wedge\ (f\ ::\ \alpha \rightarrow (\beta \rightarrow \delta))
\end{array}
$$

$$
\begin{array}{l}
\quad (f\ ::\ \alpha \rightarrow (\beta \rightarrow \delta))\ \wedge\ (f\ ::\ \delta \rightarrow \gamma) \\
(i) \vdash \{\ \} \\
\quad (\alpha \rightarrow (\beta \rightarrow \delta))\ =\ (\delta \rightarrow \gamma)
\end{array}
$$

$$
\begin{array}{l}
\quad (\alpha \rightarrow (\beta \rightarrow \delta))\ =\ (\delta \rightarrow \gamma) \\
(i) \vdash \{\ \} \\
\quad (\delta\ =\ \alpha)\ \wedge\ (\gamma\ =\ (\beta \rightarrow \delta))
\end{array}
$$

$$g\ ::\ \alpha \rightarrow \beta \rightarrow (\beta \rightarrow \alpha)$$

Ahora nos ocuparemos de la segunda. Escribamos la ecuación en la forma

$$cosa\ f\ x\ g\ =\ g\ f\ (f\ x\ g)$$

Sean las asignaciones de tipos iniciales $f :: \alpha,\ x :: \beta,\ g :: \gamma,\ g\ f\ (f\ x\ g) :: \delta$; entonces $cosa :: \alpha \to \beta \to \gamma \to \delta$, con lo cual

$$
\begin{array}{l}
\quad g\ f\ (f\ x\ g) :: \delta \\
(a) \vdash \{\ \} \\
\quad \exists\, \tau \mid (f\ x\ g :: \tau) \wedge (g\ f :: \tau \to \delta)
\end{array}
$$

$$
\begin{array}{l}
\quad f\ x\ g :: \tau \\
(a) \vdash \{\ \} \\
\quad \exists\, \gamma \mid (g :: \gamma) \wedge (f\ x :: \gamma \to \tau)
\end{array}
$$

$$
\begin{array}{l}
\quad f\ x :: \gamma \to \tau \\
(a) \vdash \{\ \} \\
\quad \exists\, \beta \mid (x :: \beta) \wedge (f :: \beta \to (\gamma \to \tau))
\end{array}
$$

$$
\begin{array}{l}
\quad (f :: \beta \to (\gamma \to \tau)) \wedge (f :: \alpha) \\
(i) \vdash \{\ \} \\
\quad \alpha = \beta \to (\gamma \to \tau)
\end{array}
$$

$$
\begin{array}{l}
\quad g\ f :: \tau \to \delta \\
(a) \vdash \{\ \} \\
\quad \exists\, \alpha \mid (f :: \alpha) \wedge (g :: \alpha \to (\tau \to \delta))
\end{array}
$$

$$
\begin{array}{l}
\quad (g :: \alpha \to (\tau \to \delta)) \wedge (g :: \gamma) \\
(i) \vdash \{\ \} \\
\quad \gamma = \alpha \to (\tau \to \delta)
\end{array}
$$

luego $cosa$ no tiene un tipo correcto porque $\alpha$ depende de $\gamma$ y a su vez $\gamma$ depende de $\alpha$; se dice que el tipo de $cosa$ es *recursivo*. Finalmente, y como no hay quinto malo

$$fix\ f\ x\ =\ f\ x\ (fix\ f)$$

Sean las asignaciones iniciales $f :: \alpha,\ x :: \beta,\ fx\ (fix\ f) :: \gamma$; entonces $fix :: \alpha \to \beta \to \gamma$, con lo cual

$$
\begin{array}{l}
\quad f\ x\ (fix\ f) :: \gamma \\
(a) \vdash \{\ \} \\
\quad \exists\, \delta \mid (fix\ f :: \delta) \wedge (f\ x :: \delta \to \gamma)
\end{array}
$$

$$
\begin{array}{l}
\quad fix\ f :: \delta \\
(a) \vdash \{\ \} \\
\quad \exists\, \alpha \mid (f :: \alpha) \wedge (fix :: \alpha \to \delta)
\end{array}
$$

$$
\begin{array}{l}
\quad (fix :: \alpha \to \delta) \wedge (fix :: \alpha \to \beta \to \gamma) \\
(i) \vdash \{\ \} \\
\quad \delta = (\beta \to \gamma)
\end{array}
$$

$$
\begin{array}{l}
\quad f\ x\ ::\ \delta\ \rightarrow\ \gamma \\
(a) \vdash \{\ \} \\
\quad \exists\ \beta\ \mid\ (x\ ::\ \beta)\ \wedge\ (f\ ::\ \beta\ \rightarrow\ (\delta\ \rightarrow\ \gamma))
\end{array}
$$

$$
\begin{array}{l}
\quad (f\ ::\ \alpha)\ \wedge\ (f\ ::\ \beta\ \rightarrow\ (\delta\ \rightarrow\ \gamma)) \\
(i) \vdash \{\ \} \\
\quad \alpha\ =\ \beta\ \rightarrow\ (\delta\ \rightarrow\ \gamma)
\end{array}
$$

$$fix\ ::\ (\beta\ \rightarrow\ (\beta\ \rightarrow\ \gamma)\ \rightarrow\ \gamma)\ \rightarrow\ \beta\ \rightarrow\ \gamma$$

**Solución al Ejercicio 17.48** (pág. 497).– No necesariamente, ya que el teorema de parametricidad concluye, $\forall\ f\ \centerdot\ f\ ::\ a\ \rightarrow\ b\ \centerdot$

$$map\ f\ .\ r_a\ =\ r_b\ .\ map\ f$$

siempre que el tipo de $r$ sea $r\ ::\ \forall\ a\ \centerdot\ [a]\ \rightarrow\ [a]$. Pero la función $(par\ <|)$ tiene por tipo

$$(par\ <|)\ ::\ [Int]\ \rightarrow\ [Int]$$

es decir, no es polimórfica. Un contraejemplo sería:

$$
\begin{array}{l}
(map\ inc\ \ .(par\ <|))\ [1,2,3]\ =\ [3] \\
((par\ <|)\ .\ (map\ inc))\ [1,2,3]\ =\ [2,4]
\end{array}
$$