Programación Funcional (nivel medio)

Trataremos aspectos teóricos y prácticos de los lenguajes de programación funcional *pura*. Se les califica de esta manera porque con ellos la computación de funciones no produce efectos laterales, eliminándose así una de las mayores fuentes de errores de programación, lo que permite usar la sustitución simbólica ("sustitución de iguales por iguales") al definir funciones y razonar acerca de éstas. A esta familia pertenecen lenguajes como *Haskell*, *Gofer*, *Miranda* y *Clean*, mismos que también cuentan con un sistema de notación para definir funciones computables de una manera muy parecida a la acostumbrada en matemáticas. Desarrollaremos el curso alrededor del estudio y uso del lenguaje *Haskell*. Como ejemplo principal veremos un programa para calcular estructuras simétricas, lo que nos permitirá ilustrar la programación de operaciones propias de la geometría, la combinatoria y los grafos, así como la producción de representaciones para visualizar dichas estructuras mediante sistemas apropiados para estos fines, como lo son *VRML* y *Mathematica*.

Temario

- Definición y evaluación de funciones.
- Estructuras de datos.
- Demostración de propiedades de programas.
- Sistema de tipos.
- Modelos de computación (cálculo-λ).

A continuación se incluye el texto de los acetatos preparados para guiar la exposición de los temas tratados durante el curso. Por ser material de apoyo carece de redacción. La mayor parte se extrajo o adaptó de [2], [8], [5], [14] y [9]. En [13] (`http://cic2.iimas.unam.mx/~velarde`) se encuentra lo correspondiente al ejemplo de las estructuras simétricas.

Funciones en matemáticas y en programación funcional

Notación en matemáticas	Notación en lenguaje `Haskell`
	`type R = Float`
$f, g : \mathbb{R} \to \mathbb{R}$	`f,g :: R -> R`
$f(x) = x^2$	`f x = x^2`
$g(x) = 2x - 5$	`g x = 2*x-5`
$h = f \circ g$	`h = f.g`
$f(4)$	`> f(4)`
	`16.0`
	ó bien, omitiendo los paréntesis:
	`> f 4`
	`16.0`
$g(7.1)$	`> g 7.1`
	`9.2`
$h(7.1)$	`> h 7.1`
	`84.64`
$p : \mathbb{Z} \to \mathbb{B}$	`p :: Int -> Bool`
$p(n) = (3 < n) \wedge (n < 8)$	`p n = 3<n && n<8`
$c : S \to \mathbb{Z}$	`c :: String -> Int`
$c(s) = \|\text{cola}(ss)\|$	`c s = length(tail(s++s))`
$q = p \circ c$	`q = p.c`
	`> :t q`
	`q :: [Char] -> Bool`
	`> q "ab"`
	`False`
	`> q "abcd"`
	`True`
	`> q "abcde"`
	`False`

Notación en matemáticas	Notación en lenguaje `Haskell`

$I : A \to A$
$I(x) = x$
	Si no se declara el tipo, se infiere el más general:
	
	`iden x = x`
	
	`> :t iden`
	`iden :: a -> a`
	`> iden 2      > iden f      > iden p`
	`2             f             p`

Sean A, B y C conjuntos, $\circ : (A \to B) \times (C \to A) \to (C \to B)$	`> :t (.)` `(.) :: (a->b) -> (c->a) -> c->b`
Composición algebraica (de izquierda a derecha):	

$rs = s \circ r$	`r >>> s = s.r`
	(ó bien: `(>>>) r s = s.r`)
	`> :t (>>>)`
	`(>>>) :: (a->b) -> (b->c) -> a->c`
	
	`q'= c >>> p`
	`> :t q'`
	`q' :: [Char] -> Bool`
	`> q' "abcd"`
	`True`
	
	`h' = g >>> f`
	`h'' = f >>> g`
	
	`> h' 7.1                 > h'' 7.1`
	`84.64                    95.82`

<table>
<tr><th>Notación en matemáticas</th><th>Notación en lenguaje <code>Haskell</code></th></tr>
<tr><td>

$$\mathrm{ff}(x) = \begin{cases} x^2, & \text{si } x \le 2, \\ 3x, & \text{si } 2 < x < 4, \\ g(x), & \text{si } 4 \le x. \end{cases}$$

</td><td>

```
ff x | x<=2        = x^2
     | 2<x && x<4  = 3*x
     | 4<=x        = g x

> ff 2        > ff 3      > ff 6
4.0           9.0         7.0
```

</td></tr>
<tr><td>

$\mathrm{fac}, \mathrm{fac}' : \mathbb{N} \to \mathbb{N}$

$$\mathrm{fac}(n) = \begin{cases} 1, & \text{si } n = 0, \\ n\,\mathrm{fac}(n-1), & \text{si } n > 0. \end{cases}$$

$$\begin{aligned} \mathrm{fac}'(0) &= 1, \\ \mathrm{fac}'(n+1) &= (n+1)\,\mathrm{fac}'(n) \end{aligned}$$

</td><td>

```
fac, fac' :: Int -> Int

fac n  = if n==0 then 1
                 else n*fac(n-1)

fac' 0     = 1
fac' (n+1) = (n+1)*fac'(n)

> fac 5                > fac' 5
120                    120
```

</td></tr>
<tr><td></td><td>

```
divisores n =
  [d | d<-[1..n], n `mod` d == 0]

primo n = divisores n == [1,n]

> divisores 847
[1, 7, 11, 77, 121, 847]
> primo 847
False
> primo 853
True
> [p | p<-[830..880], primo p]
[839, 853, 857, 859, 863, 877]
```

</td></tr>
</table>

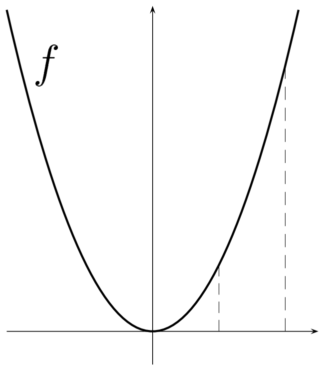
Notación en matemáticas	Notación en lenguaje `Haskell`
Producto vectorial:	
	`type R3 = (R,R,R)`
$\times : (\mathbb{R}^3)^2 \to \mathbb{R}^3$	`(>*<) :: R3 -> R3 -> R3`
$(x_1, y_1, z_1) \times (x_2, y_2, z_2) =$	`(x1,y1,z1) >*< (x2,y2,z2) =`
$(y_1 z_2 - y_2 z_1,$	`(y1*z2-y2*z1,`
$z_1 x_2 - z_2 x_1,$	`z1*x2-z2*x1,`
$x_1 y_2 - x_2 y_1)$	`x1*y2-x2*y1)`
Producto de un escalar por un elmento de $\mathbb{R}^3$:	
$a \in \mathbb{R}, (x, y, z) \in \mathbb{R}^3,$	`(<.>) :: R -> R3 -> R3`
$a(x, y, z) = (ax, ay, az)$	`a <.> (x,y,z) = (a*x,a*y,a*z)`
Producto interior de dos elementos de $\mathbb{R}^3$:	
$\bullet : (\mathbb{R}^3)^2 \to \mathbb{R}$	`(<*>) :: R3 -> R3 -> R`
$(x_1, y_1, z_1) \bullet (x_2, y_2, z_2) =$	`(x1,y1,z1) <*> (x2,y2,z2) =`
$x_1 x_2 + y_1 y_2 + z_1 z_2$	`x1*x2 + y1*y2 + z1*z2`
Norma o tamaño de un elementos de $\mathbb{R}^3$:	
$\lvert\,\rvert : \mathbb{R}^3 \to \mathbb{R}$	`norma :: R3 -> R`
$\lvert(x, y, z)\rvert = \sqrt{x^2 + y^2 + z^2}$	`norma (x,y,z) = sqrt (x^2+y^2+z^2)`
Triple producto escalar:	
	`tpe :: R3 -> R3 -> R3  -> R`
$A \bullet (B \times C)$	`tpe a b c = a <*> (b >*< c)`
$(3, 4, 5) \times (4, 5, 6)$	`> (3,4,5) >*< (4,5,6)`
	`(-1.0,2.0,-1.0)`
$2.1(3, 4, 5)$	`> 2.1 <.> (3,4,5)`
	`(6.3,8.4,10.5)`
$(3, 4, 5) \bullet (4, 5, 6)$	`> (3,4,5)<*>(4,5,6)`
	`62.0`
$\lvert(3, 4, 5)\rvert$	`> norma (3,4,5)`
	`7.07107`
$(3, 4, 5) \bullet (4, 5, 6) \times (2, -1, 3)$	`> tpe (3,4,5) (4,5,6) (2,-1,3)`
	`-7.0`

Notación en mats.	Notación en lenguaje `Haskell`

Suma de Riemann (aproximación a la integral de Riemann):

$S_{[a,b]}(f,n) = \sum_{i=0}^{n-1} f(x_i)\Delta x$

donde $\Delta x = \dfrac{b-a}{n}$




```
suma_de_Riemann f (a,b) n =
  sum [f(a+i*deltax)*deltax | i<-[0..n-1]]
  where deltax = (b-a)/n
> :t suma_de_Riemann
suma_de_Riemann :: (Fractional a, Enum a) =>
                        (a->a)->(a,a)->a -> a
> suma_de_Riemann f (1,2) 100
2.31835
> suma_de_Riemann f (1,2) 1000
2.33183
> suma_de_Riemann g (1,2) 1000
-2.001
> suma_de_Riemann h (1,2) 1000
4.33733
> suma_de_Riemann ff (1,6) 2000
30.3121
> suma_de_Riemann (\x->x^3) (0,2) 1000
3.992

suma_de_RiemannB f (a,b) n =
  sum [f(xi)*deltax | xi<-xis]
  where deltax = (b-a)/fromInt(n)
        xis = take n (iterate (+deltax) a)
> :t suma_de_RiemannB
suma_de_RiemannB ::
  Fractional a => (a->a)->(a,a)->Int -> a
> :t fromInt
fromInt :: Num a => Int -> a
> :t iterate
iterate :: (a -> a) -> a -> [a]
> suma_de_RiemannB h (1,2) 1000
4.33718
```

Cálculo lambda: un modelo de computación mediante funciones.

Definición Dados los símbolos 'λ', '(', ')' y '.', y un conjunto numerable $\mathcal{V}$ (de *variables*), son *términos–λ* (elementos de $\mathcal{T}_\lambda$):

1.	x	$x \in \mathcal{V}$	
2.	$(\lambda x.M)$	$x \in \mathcal{V}, M \in \mathcal{T}_\lambda$	*Abstracción.* Denota la función $x \mapsto M$.
3.	(MN)	$M, N \in \mathcal{T}_\lambda$	*Aplicación* (de la función M al argumento N).

Ejemplos:

$$(\lambda x.x), \quad (\lambda x.y), \quad (x(\lambda x.y)), \quad (\lambda x.(y(\lambda z.(xy)))).$$

Convenciones:

$$\begin{array}{lll} M_1M_2M_3\cdots M_n & \text{abrevia} & (((M_1M_2)M_3)\cdots M_n). \\ \lambda x_1x_2\cdots x_n.M & \text{abrevia} & (\lambda x_1.(\lambda x_2.\cdots(\lambda x_n.M))). \end{array}$$

Mas ejemplos:

$\mathbf{I} \equiv \lambda x.x$	*Identidad.*
$\circ \equiv \lambda fgx.f(gx)$	*Composición.*
$\Omega \equiv (\lambda x.xx)(\lambda x.xx)$	
$\mathbf{K} \equiv \lambda xy.x$	*Cancelador.*
$\mathbf{S} \equiv \lambda fgx.fx(gx)$	*Distribuidor.*
$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx))$	*Combinador de punto fijo (Church).*
$\Theta \equiv (\lambda xy.y(xxy))\,(\lambda xy.y(xxy))$	*Combinador de punto fijo (Turing).*

$\mathbf{0} \equiv \lambda fx.x$	
$\mathbf{1} \equiv \lambda fx.fx$	*Numerales de Church.*
$\mathbf{2} \equiv \lambda fx.f(fx)$	
$\vdots$	

$\mathbf{true} \equiv \lambda xy.x$

$\mathbf{false} \equiv \lambda xy.y$

$\mathbf{if} \equiv \lambda pxy.pxy$

$\mathbf{suc} \equiv \lambda nfx.f(nfx)$

$\mathbf{is0} \equiv \lambda n.n(\lambda x.\mathbf{false})\,\mathbf{true}$

<u>Definición</u> del conjunto de variables *libres* en un término–λ.

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) - \{x\} \\
\mathrm{FV}(MN) &= \mathrm{FV}(M) \cup \mathrm{FV}(N)
\end{aligned}$$

<u>Definición</u> del conjunto de variables *ligadas* en un término–λ.

$$\begin{aligned}
\mathrm{BV}(x) &= \emptyset \\
\mathrm{BV}(\lambda x.M) &= \mathrm{BV}(M) \cup \{x\} \\
\mathrm{BV}(MN) &= \mathrm{BV}(M) \cup \mathrm{BV}(N)
\end{aligned}$$

<u>Definición</u> de *sustitución* de L por las ocurrencias libres de y en P.

$$\begin{aligned}
x[L/y] &= \begin{cases} L, & \text{si } x \equiv y; \\ x, & \text{en caso contrario.} \end{cases} \\
(\lambda x.M)[L/y] &= \begin{cases} \lambda x.M, & \text{si } x \equiv y; \\ \lambda x.M[L/y], & \text{en caso contrario.} \end{cases} \\
(MN)[L/y] &= M[L/y]\, N[L/y].
\end{aligned}$$

<u>Definición</u> de *reglas de conversión*.

1.	$\lambda x.M \to_\alpha (\lambda y.M)[y/x]$	$y \notin \mathrm{FV}(M) \cup \mathrm{BV}(M)$	*Conversión–α*
2.	$(\lambda x.M)N \to_\beta M[N/x]$	$\mathrm{FV}(N) \cap \mathrm{BV}(M) = \emptyset$	*Conversión–β*
3.	$\lambda x.Mx \to_\eta M$	$x \notin \mathrm{FV}(M)$	*Conversión–η*

Ejemplos:

$\lambda x.xz \to_\alpha \lambda y.yz$	$\lambda x.(zy)x \to_\eta zy$
$\lambda x.xy \to_\alpha \lambda y.yy$ ✘	$\lambda x.(xy)x \to_\eta xy$ ✘
$(\lambda z.zy)(\lambda x.x) \to_\beta (\lambda x.x)y \to_\beta y$	$(\lambda x.Mx)N \to_\beta MN,\ x \notin \mathrm{FV}(M)$
$(\lambda x.\lambda y.xy)(yz) \to_\beta \lambda y.(yz)y$ ✘	$(\lambda x.(zy)x)N \to_\beta (zy)N$
$(\lambda x.\lambda v.xv)(yz) \to_\beta \lambda v.(vz)y$ ✔	$(\lambda x.(xy)x)N \to_\beta (Ny)N$

Ejemplos.

En cálculo–λ	En `Haskell`
$\lambda x.x^2$	`\x->x^2`
$\lambda x.2x-5$	`\x->2*x-5`
$(\lambda x.x^2)\,4$	`> (\x->x^2) 4`
	`16`
$(\lambda x.2x-5)\,7.1$	`(\x->2*x-5) 7.1`
	`9.2`
$(\lambda x.x^2)((\lambda x.2x-5)7.1)$	`> (\x->x^2) ((\x->2*x-5) 7.1)`
	`84.64`
	`> (\x -> (\y -> x*(y+5))) 2 3`
	`16`
	`> (\x -> \y -> x*(y+5)) 2 3`
	`16`
	`(\x y -> x*(y+5)) 2 3`
	`16`
	`> (\(x,y) ->x*(y+5)) (2,3)`
	`16`
	`> :t (\x -> \y -> x*(y+5))`
	`\x -> \y -> x*(y+5) :: Num a => a->a->a`
	`> :t (\x -> \y -> x*(y+5)) 2`
	`(\x -> \y -> x*(y+5)) 2 :: Num a => a->a`

Con $\quad \mathbf{K} \equiv \lambda xy.x\,,\quad \mathbf{S} \equiv \lambda fgx.fx(gx) \quad$ y $\;\circ \equiv \lambda fgx.f(gx)\,,$

$$\underline{\mathbf{S}(\mathbf{KS})\mathbf{K}f}gx \;\to_\beta\; \underline{(\mathbf{KS})f}(\mathbf{K}f)gx \;\to_\beta\; \underline{\mathbf{S}(\mathbf{K}f)gx} \;\to_\beta\; \underline{(\mathbf{K}f)x}(gx) \;\to_\beta\; f(gx)$$

O sea que el operador $\circ$ se podría definir así: $\circ \equiv \mathbf{S}(\mathbf{KS})\mathbf{K}$. En Haskell:

```
k = \x y -> x
s = \f g x -> f x (g x)
(|.|) = s (k s) k
> (p |.| c) "abcd"
True
```

Definición de *reducción* ($\rightarrow$). Sean M, M', N y N' términos–λ,

$$\frac{M \rightarrow_\beta M'}{M \rightarrow M'} \quad \frac{M \rightarrow_\eta M'}{M \rightarrow M'} \quad \frac{M \rightarrow M'}{\lambda x.M \rightarrow \lambda x.M'} \quad \frac{M \rightarrow M'}{MN \rightarrow M'N} \quad \frac{N \rightarrow N'}{MN \rightarrow MN'}$$

La cerradura transitiva de $\rightarrow$ se denotará mediante $\twoheadrightarrow$.

Un término–λ está en *forma normal* si no admite reducciones.

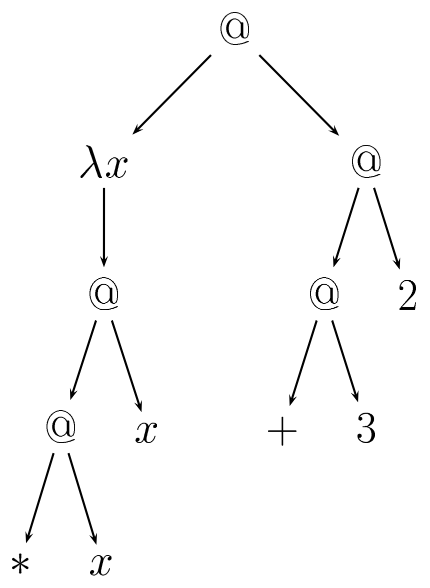
Un término–λ tiene forma normal si puede ser reducido a dicha forma.

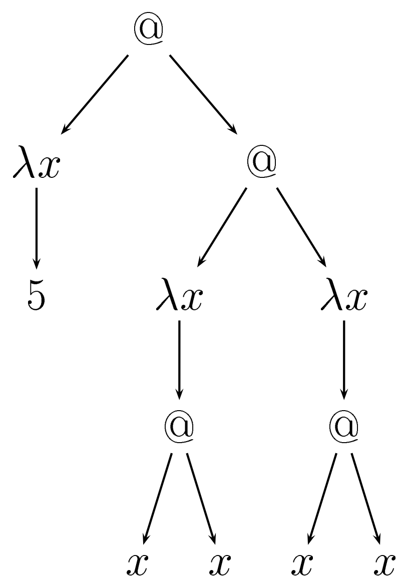
Ejemplos:

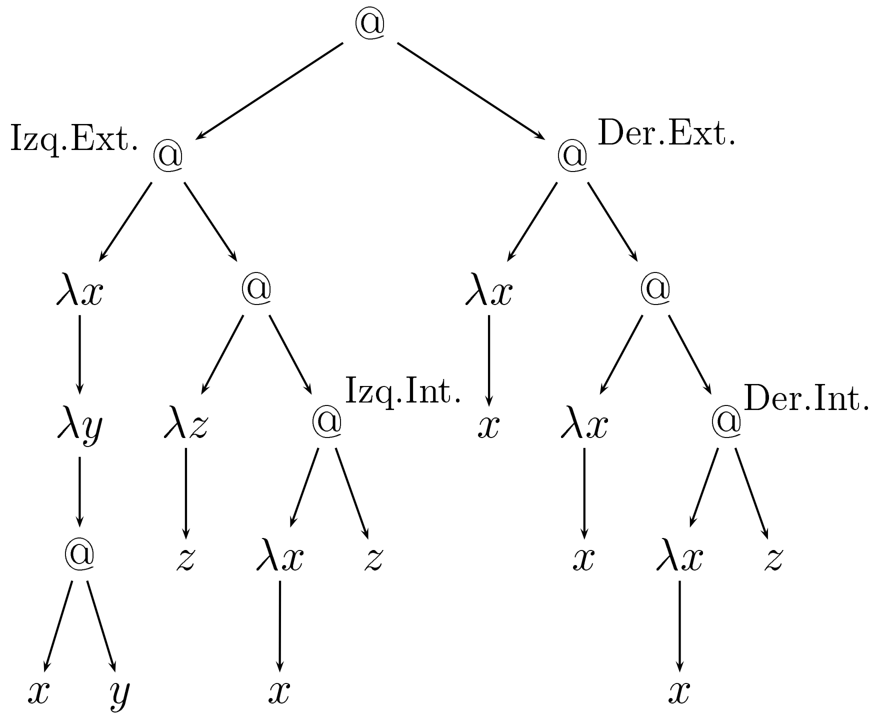
$$(\lambda z.zy)(\lambda x.x) \rightarrow_\beta (\lambda x.x)y \rightarrow_\beta y$$
$$(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \cdots$$

$$(\lambda x.x * x)\,(3+2) \rightarrow_\beta (\lambda x.x * x)\,5 \rightarrow_\beta 5 * 5 \rightarrow_\beta 25$$
$$(\lambda x.x * x)\,(3+2) \rightarrow_\beta (3+2) * (3+2) \rightarrow_\beta 5 * (3+2) \rightarrow_\beta 5 * 5 \rightarrow_\beta 25$$

$$(\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta 5$$
$$(\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta (\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta$$
$$\qquad (\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta (\lambda x.5)\,((\lambda x.xx)(\lambda x.xx)) \rightarrow_\beta \cdots$$





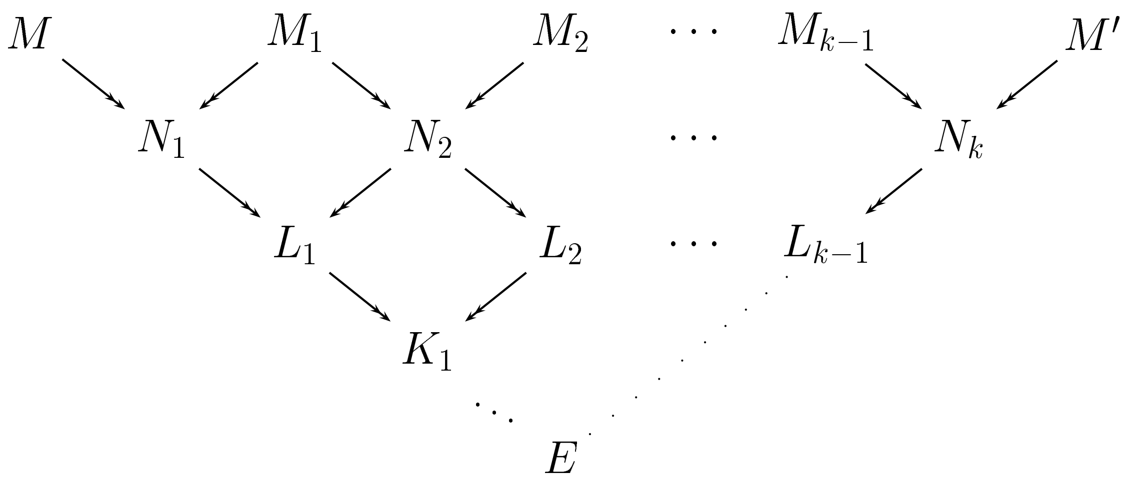


$$(\lambda x.\lambda y.xy)\ \ ((\lambda z.z)((\lambda x.x)z))\ \ ((\lambda x.x)\ ((\lambda x.x)((\lambda x.x)z)))$$

$$\begin{tikzcd}
 & M \arrow[dl, twoheadrightarrow] \arrow[dr, twoheadrightarrow] & \\
M_1 \arrow[dr, twoheadrightarrow] & & M_2 \arrow[dl, twoheadrightarrow] \\
 & L &
\end{tikzcd}$$

$$(M \twoheadrightarrow M_1 \wedge M \twoheadrightarrow M_2) \Rightarrow \exists_L(M_1 \twoheadrightarrow L \wedge M_2 \twoheadrightarrow L).\ \text{(Church-Rosser)}$$

Si un término–λ tiene forma normal, ésta es única.



$M = M'$ si hay E tal que $M \twoheadrightarrow E$ y $M' \twoheadrightarrow E$.

Reducción en *orden normal*. En cada paso:

- reducir $(\lambda x.M)N$ antes que M ó N y
- reducir M antes que N en MN.

La reducción de un término–λ M en orden normal lleva a la forma normal de M (si ésta existe).

Evaluación *perezosa* ("lazy").

```
> take 4 [0..]
[0,1,2,3]

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

> take 6 (iterate f 3)  where f x = x^2
[3,9,81,6561,43046721,1853020188851841]
```

Ejemplo: Criba de Eratóstenes [2, pp.174–176]

$$\begin{array}{cccccccccccccccc}
2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \ldots \\
\cancel{2} & 3 & \cancel{4} & 5 & \cancel{6} & 7 & \cancel{8} & 9 & \cancel{10} & 11 & \cancel{12} & 13 & \cancel{14} & 15 & \ldots \\
 & \cancel{3} & & 5 & & 7 & & \cancel{9} & & 11 & & 13 & & \cancel{15} & \ldots \\
 & & & \cancel{5} & & 7 & & & & 11 & & 13 & & & \ldots \\
 & & & & & \cancel{7} & & & & 11 & & 13 & & & \ldots \\
 & & & & & & & & & & \ddots & & & &
\end{array}$$

```
criba (p:xs) = [ x | x <- xs, (x `mod` p) /= 0 ]
primos = map head (iterate criba [2..])

> take 20 primos
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
```

- Un argumento se reduce tanto como se requiera para producir la respuesta.
- Tal reducción se efectúa sólo una vez.

A partir de la definición anterior,

```
criba (p:xs) = [ x | x <- xs, (x `mod` p) /= 0 ]
primos = map head (iterate criba [2..])
```

puede sintetizarse la siguiente, en la que sólo se usa una lista infinita:

```
primos = cribaR [2..]
cribaR (p:xs) = p : cribaR [x|x<-xs, (x `mod` p)/=0]
```

Para ello, la primera definición se reformula así:

```
primos = cribaR [2..]
cribaR xs = map head (iterate criba xs)
```

esta definición de `cribaR` es equivalente a:

```
cribaR (p:xs) = map head (iterate criba (p:xs))
```

y usando las definiciones de las funciones que aparecen en el lado derecho, se procede a transformar éste:

```
  = map head ((p:xs) : iterate criba (criba (p:xs)))

  = p : map head (iterate criba (criba (p:xs)))

  = p : map head (iterate criba [x|x<-xs, (x `mod` p)/=0])

  = p : cribaR [x|x<-xs, (x `mod` p)/=0]
```

con lo que se obtiene la definición recursiva de `cribaR`:

```
cribaR (p:xs) = p : cribaR [x|x<-xs, (x `mod` p)/=0]
```

Notación en `Haskell`	En notación–λ
`f x y = x * y`	$\mathbf{f} \equiv \lambda xy.\mathbf{mult}\, xy$
`fac n = if n==0 then 1` `         else n*fac(n-1)`	$\mathbf{fac} \equiv \lambda n.\mathbf{if}\,(\mathbf{is0}\, n)\, 1\, (\mathbf{mult}\, n\, (\mathbf{fac}\, (\mathbf{pre}\, n)))$ ✘

Recursión con combinadores de punto fijo [11, p.99]

Se dice que x es *punto fijo* de una función f si $f(x) = x$.

Un *combinador* es un término–λ sin variables libres (*cerrado*).

Un combinador $\mathbf{Y}$ es *de punto fijo* si $\mathbf{Y}F = F(\mathbf{Y}F)$, para cada término–$\lambda$ F. Es decir, si $(\mathbf{Y}F)$ es punto fijo de F.

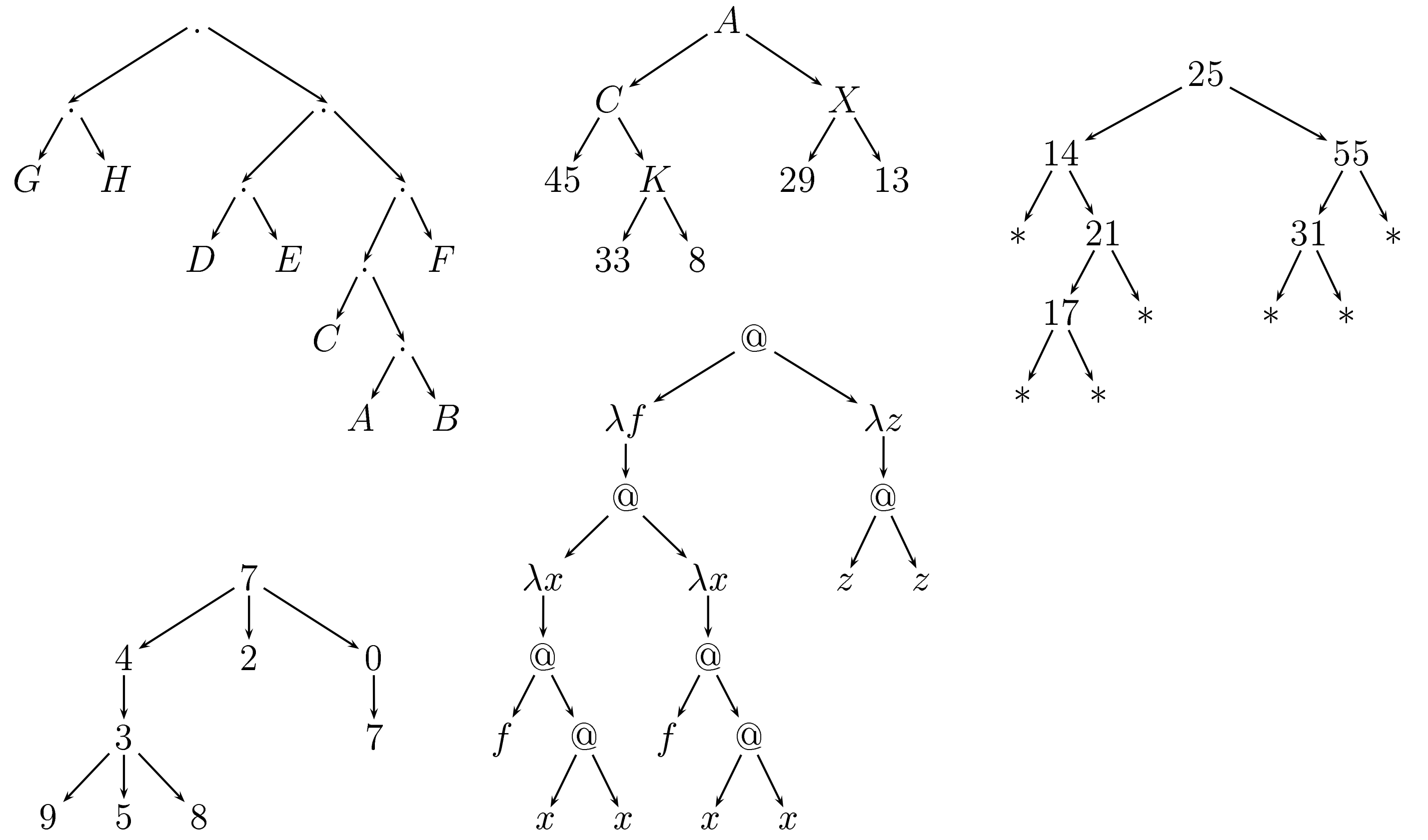
$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))\,(\lambda x.f(xx))$ es combinador de punto fijo porque

$\mathbf{Y}F \to (\lambda x.F(xx))\,(\lambda x.F(xx)) \to F\big((\lambda x.F(xx))\,(\lambda x.F(xx))\big) = F(\mathbf{Y}F)$

$$F \equiv \lambda fn.\mathbf{if}\,(\mathbf{is0}\, n)\, 1\, (\mathbf{mult}\, n\, (f(\mathbf{pre}\, n)))$$
$$\mathbf{fac} \equiv \mathbf{Y}F \quad ✔$$

$$\begin{aligned}
\mathbf{fac}\, 3 &\equiv \mathbf{Y}F\, 3 \\
&= F(\mathbf{Y}F)\, 3 \\
&\equiv \big(\lambda fn.\mathbf{if}\,(\mathbf{is0}\, n)\, 1\, (\mathbf{mult}\ n\, (f(\mathbf{pre}\ n)))\big)\, (\mathbf{Y}F)\, 3 \\
&\equiv \big(\lambda fn.\mathbf{if}\,(\mathbf{is0}\, n)\, 1\, (\mathbf{mult}\ n\, (f(\mathbf{pre}\ n)))\big)\, \mathbf{fac}\, 3 \\
&\to \big(\lambda n.\mathbf{if}\,(\mathbf{is0}\, n)\, 1\, (\mathbf{mult}\ n\, (\mathbf{fac}\, (\mathbf{pre}\ n)))\big)\, 3 \\
&\to \mathbf{if}\,(\mathbf{is0}\, 3)\, 1\, (\mathbf{mult}\ 3\, (\mathbf{fac}\, (\mathbf{pre}\ 3))) \\
&\twoheadrightarrow \mathbf{mult}\, 3\, \big(\mathbf{fac}\, (\mathbf{pre}\, 3)\big) \\
&\qquad \vdots \\
&\twoheadrightarrow \mathbf{mult}\, 3\, \big(\mathbf{mult}\, 2\, \big(\mathbf{mult}\, 1\, (\mathbf{fac}\, (\mathbf{pre}\, 1))\big)\big) \\
&\qquad \vdots \\
&\twoheadrightarrow \mathbf{mult}\, 3\, (\mathbf{mult}\, 2\, (\mathbf{mult}\, 1\, 1)) \\
&\twoheadrightarrow 6
\end{aligned}$$

Tipos de Datos. Ejemplo: Árboles [5, pp.106–110]




```
data BinTree a = Leaf a  |  BinTree a :^: BinTree a

data LabTree l a = Tip a | LFork l (LabTree l a) (LabTree l a)

data STree a = Empty  |  Split a (STree a) (STree a)

data RoseTree a = Node a [RoseTree a]

type Name = String
data Term = Var Name          -- variable
          | Lam Name Term     -- abstracción
          | Ap Term Term      -- aplicación


class Tree t where
  subtrees :: t -> [t]
```

```
instance Tree (BinTree a) where
  subtrees (Leaf n) = []
  subtrees (l :^: r) = [l,r]

instance Tree (LabTree l a) where
  subtrees (Tip x) = []
  subtrees (LFork x l r) = [l,r]

instance Tree (STree a) where
  subtrees Empty = []
  subtrees (Split x l r) = [l,r]

instance Tree (RoseTree a) where
  subtrees (Node x gts) = gts

instance Tree Term where
  subtrees (Var _) = []
  subtrees (Ap f x) = [f,x]
  subtrees (Lam v b) = [b]
```

```
depth :: Tree t => t -> Int
depth = (1+) . foldl max 0 . map depth . subtrees

> depth btree
6
> depth ltree
4
> depth stree
5
> depth rtree
4
> depth lamtree
7
```

```
class Tree t => DrawTree t where
  drawTree :: t -> String
  labTree  :: t -> String
  drawTree = drawTree' labTree
```

```
instance Show a => DrawTree (BinTree a) where
  labTree (Leaf n)  = show n
  labTree (_ :^: _) = "."

instance (Show l,Show a) => DrawTree (LabTree l a) where
  labTree (Tip x)       = show x
  labTree (LFork x _ _) = show x

instance Show a => DrawTree (STree a) where
  labTree Empty         = "*"
  labTree (Split x _ _) = show x

instance Show a => DrawTree (RoseTree a) where
  labTree (Node x _) = show x

instance DrawTree Term where
  labTree (Var v) = v
  labTree (Ap _ _) = "@"
  labTree (Lam v _) = "\"++v
```

```
drawTree' :: Tree t => (t -> String) -> t -> String
drawTree' f t = (unlines.snd) (pic t)
                 where pic t = ···
                                ⋮
```

```
btree1 = (Leaf 'G') :^: (Leaf 'H')
btree2 = (Leaf 'D') :^: (Leaf 'E')
btree3 = (Leaf 'A') :^: (Leaf 'B')
btree = btree1 :^: (btree2 :^: ((Leaf 'C') :^: btree3) :^: Leaf 'F')

ltree = LFork 'A' (LFork 'C' (Tip 45) (LFork 'K' (Tip 33) (Tip 8)))
                  (LFork 'X' (Tip 29) (Tip 13))

stree = Split 25 st1 st2
  where st1 = Split 14 Empty (Split 21 (Split 17 Empty Empty) Empty)
        st2 = Split 55 (Split 31 Empty Empty) Empty

rtree = Node 7 [Node 4 [Node 3 [Node 9 [], Node 5 [], Node 8 []]],
                Node 2 [], Node 0 [Node 7 []]]

lamtree = Ap (Lam "f" (Ap lamt lamt)) (Lam "z" (Ap (Var "z") (Var "z")))
          where lamt = Lam "x" (Ap (Var "f") (Ap (Var "x") (Var "x")))
```

```
dibujaArbol :: DrawTree a => a -> IO ()
dibujaArbol = putStr . drawTree
```

```
> dibujaArbol btree

--.--.--'G'
  |  |
  |  `--'H'
  |
  `--.--.--'D'
     |  |
     |  `--'E'
     |
     `--.--.--'C'
        |  |
        |  `--.--'A'
        |     |
        |     `--'B'
        |
        `--'F'
```

```
> dibujaArbol ltree

--'A'--'C'--45
    |    |
    |    `--'K'--33
    |         |
    |         `--8
    |
    `--'X'--29
         |
         `--13
```

```
> dibujaArbol stree

--25--14--*
   |   |
   |   `--21--17--*
   |       |   |
   |       |   `--*
   |       |
   |       `--*
   |
   `--55--31--*
       |   |
       |   `--*
       |
       `--*
```

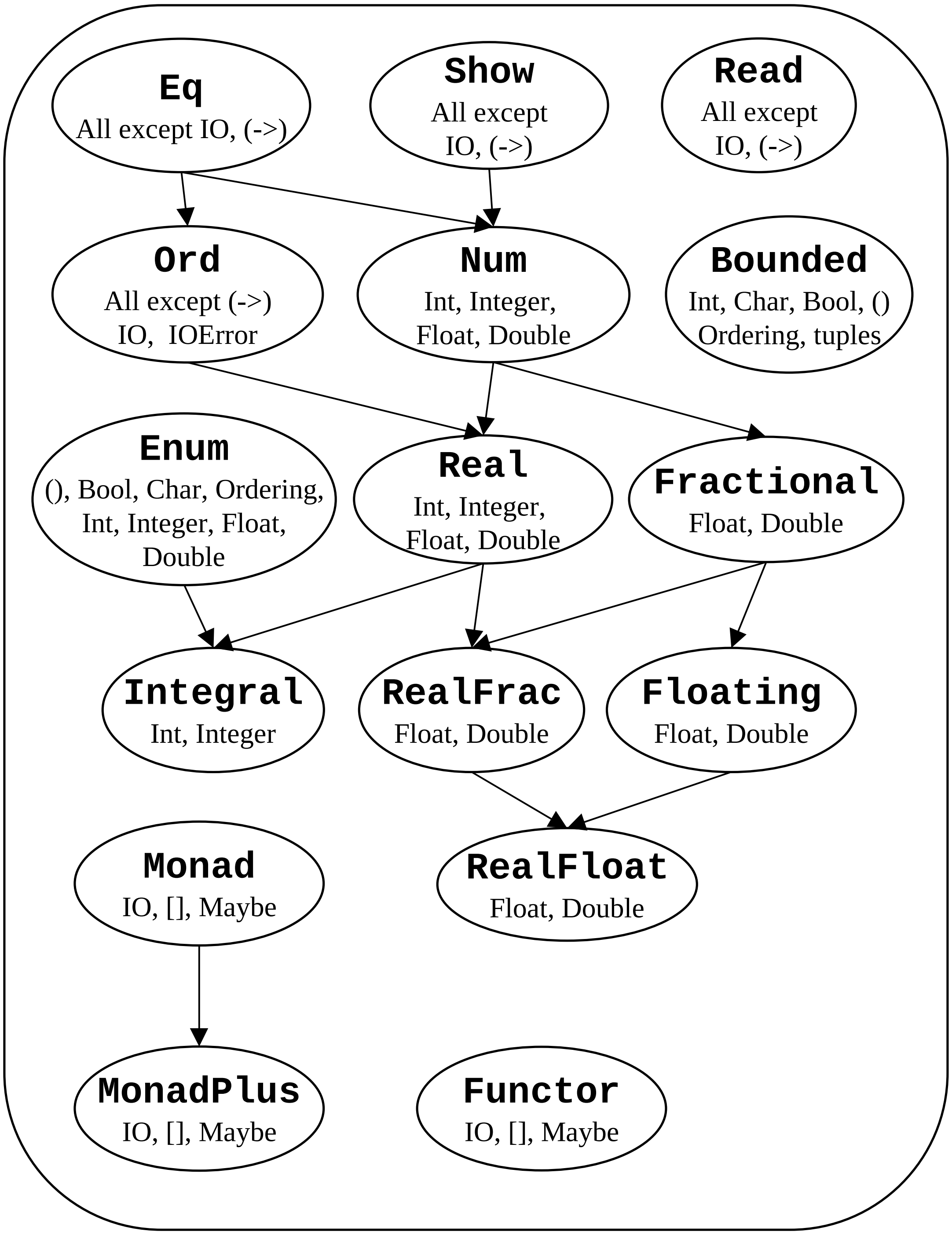
```
> dibujaArbol rtree

--7--4--3--9
  |     |
  |     |--5
  |     |
  |     `--8
  |
  |--2
  |
  `--0--7
```

```
> dibujaArbol lamtree

--@--\f--@--\x--@--f
  |      |      |
  |      |      `--@--x
  |      |         |
  |      |         `--x
  |      |
  |      `--\x--@--f
  |             |
  |             `--@--x
  |                |
  |                `--x
  |
  `--\z--@--z
         |
         `--z
```

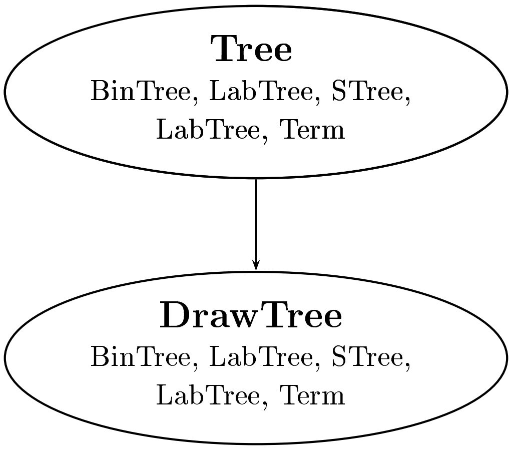
```
drawTree' f t = pstree f t 0
  where pstree f t n =
    case pst_ts of
      []-> margen n ++ nodo
      _ -> margen n ++ "\pstree{" ++ nodo ++ "}{" ++
                       concat pst_ts ++ "}"
        where pst_ts = [pstree f st (n+2) | st<-subtrees t]
              margen n = "\n" ++ (replicate n ' ')
              nodo = "\TR{$" ++ (f t) ++ "$}"
```

```
> dibujaArbol stree

\pstree{\TR{$25$}}{
  \pstree{\TR{$14$}}{
    \TR{$*$}
    \pstree{\TR{$21$}}{
      \pstree{\TR{$17$}}{
        \TR{$*$}
        \TR{$*$}}
      \TR{$*$}}}
  \pstree{\TR{$55$}}{
    \pstree{\TR{$31$}}{
      \TR{$*$}
      \TR{$*$}}
    \TR{$*$}}}
```

```
> dibujaArbol lamtree

\pstree{\TR{$@$}}{
  \pstree{\TR{$\lambda f$}}{
    \pstree{\TR{$@$}}{
      \pstree{\TR{$\lambda x$}}{
        \pstree{\TR{$@$}}{
          \TR{$f$}
          \pstree{\TR{$@$}}{
            \TR{$x$}
            \TR{$x$}}}}
      \pstree{\TR{$\lambda x$}}{
        \pstree{\TR{$@$}}{
          \TR{$f$}
          \pstree{\TR{$@$}}{
            \TR{$x$}
            \TR{$x$}}}}}}
  \pstree{\TR{$\lambda z$}}{
    \pstree{\TR{$@$}}{
      \TR{$z$}
      \TR{$z$}}}}
```



Standard Haskell Classes



Ejemplos [9, apéndice A]

```
class Eq a where    -- Definida en el módulo Prelude.hs
  (==), (/=) :: a -> a -> Bool
  x /= y  =  not (x==y)

instance Eq a => Eq (BinTree a) where
  (Leaf m) == (Leaf n)  =  m == n
  (u:^:v) == (x:^:y)  =  u==x && v==y
  _ == _  =  False

> btree1 == btree3              > btree == btree
False                           True
```

```
head                 :: [a] -> a    -- Definidas en Prelude.hs
head (x:_)            = x

tail                 :: [a] -> [a]
tail (_:xs)           = xs

(++)                 :: [a] -> [a] -> [a]
[]     ++ ys          = ys
(x:xs) ++ ys          = x : (xs ++ ys)

map                  :: (a -> b) -> [a] -> [b]
map f xs              = [ f x | x <- xs ]
```

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)

instance Functor [] where
  fmap = map
```

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a
  p >> q  = p >>= \_-> q
  fail s  = error s
```

Mónadas [14, pp.24–35]

```
module Monadas where

data Term = Con Int | Div Term Term  deriving (Show)

eval :: Term -> Int                          -- Evaluador básico --
eval (Con a) = a
eval (Div t u) = eval t `div` eval u

buena = (Div(Div(Con 1972)(Con 2))(Con 23))
mala = (Div(Con 1)(Con 0))
```

```
Monadas> eval buena           Monadas> eval mala
42                            Program error: primDivInt 1 0
```

```
                   -- Excepciones --
data M a = Raise Exception | Return a
type Exception = String

eval :: Term -> M Int
eval (Con a) = Return a
eval (Div t u) =
  case eval t of
    Raise e -> Raise e
    Return a ->
      case eval u of
        Raise e -> Raise e
        Return b -> if b==0 then Raise "div. entre 0"
```

```
Monadas> eval buena           Monadas> eval mala
Return 42                     Raise "div. entre 0"
```

```
                   -- Estados --
type M a = Estado -> (a,Estado)
type Estado = Int

eval :: Term -> M Int
eval (Con a) x = (a,x)
eval (Div t u) x =
  let (a,y) = eval t x in
  let (b,z) = eval u y in
  (a `div` b, z+1)

Monadas> eval buena 0
(42,2)
```

```
                   -- Salida --
type M a = (Output,a)
type Output = String

eval :: Term -> M Int
eval (Con a) = (line (Con a) a, a)
eval (Div t u) =
  let (x,a) = eval t in
  let (y,b) = eval u in
  (x ++ y ++ line (Div t u) (a `div` b), a `div` b)

line :: Term -> Int -> Output
line t a = "eval(" ++ show t ++ ") <= " ++ show a ++ "\n"

Monadas> let (out,_) = eval buena in putStr out
eval(Con 1972) <= 1972
eval(Con 2) <= 2
eval(Div (Con 1972) (Con 2)) <= 986
eval(Con 23) <= 23
eval(Div (Div (Con 1972) (Con 2)) (Con 23)) <= 42
```

```
                    -- Salida (continuación)--
--Con:
  (line (Div t u) (a ‘div‘ b) ++ y ++ x, a ‘div‘ b)
--en lugar de:
  (x ++ y ++ line (Div t u) (a ‘div‘ b), a ‘div‘ b)

Monadas> let (out,_) = eval buena in putStr out
eval(Div (Div (Con 1972) (Con 2)) (Con 23)) <= 42
eval(Con 23) <= 23
eval(Div (Con 1972) (Con 2)) <= 986
eval(Con 2) <= 2
eval(Con 1972) <= 1972
```

```
unit :: a -> M a
(<*>) :: M a -> (a -> M b) -> M b
```

```
type M a = a                          -- Evaluador básico bis --
unit a = a
a <*> k = k a

eval :: Term -> M Int
eval (Con a) = unit a
eval (Div t u) = eval t <*> \a -> eval u
                        <*> \b -> unit (a ‘div‘ b)

Monadas> eval buena
42
```

```
                    -- Excepciones bis --
unit a = Return a
m <*> k = case m of
            Raise e -> Raise e
            Return a -> k a

raise :: Exception -> M a
raise e = Raise e

eval :: Term -> M Int
eval (Con a) = unit a
eval (Div t u) = eval t <*> \a -> eval u
                        <*> \b -> if b==0
                                    then raise "div. entre 0"
                                    else unit (a `div` b)

Monadas> eval buena           Monadas> eval mala
Return 42                     Raise "div. entre 0"
```

```
                    -- Estados bis --
unit a = \x -> (a,x)
m <*> k = \x -> let (a,y) = m x in
                let (b,z) = k a y in
                (b,z)

tick :: M ()
tick = \x -> ((),x+1)

eval :: Term -> M Int
eval (Con a) = unit a
eval (Div t u) = eval t <*> \a -> eval u
                        <*> \b -> tick
                                  <*> \()->unit (a `div` b)

Monadas> eval buena 0
(42,2)
```

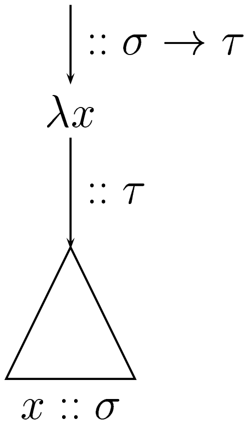
```
                    -- Salida bis --
unit a = (" ",a)
m <*> k = let (x,a) = m in
          let (y,b) = k a in
          (x++y,b)

out :: Output -> M ()
out x = (x,())

eval :: Term -> M Int
eval (Con a) = out (line (Con a) a) <*> \()->unit a
eval (Div t u) = eval t <*> \a -> eval u
                     <*> \b -> out (line (Div t u)(a `div` b))
                               <*> \()->unit (a `div` b)

Monadas> let (out,_) = eval buena in putStr out
eval(Con 1972) <= 1972
eval(Con 2) <= 2
eval(Div (Con 1972) (Con 2)) <= 986
eval(Con 23) <= 23
eval(Div (Div (Con 1972) (Con 2)) (Con 23)) <= 42
```

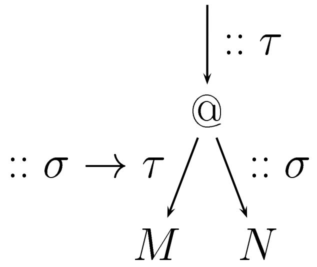
```
class Monada m where
  unit :: a -> m a
  (<*>) :: m a -> (a -> m b) -> m b
```

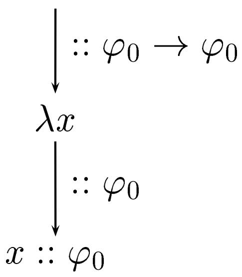
```
instance Monada M where
  unit a = (" ",a)
  m <*> k = let (x,a) = m in
            let (y,b) = k a in
            (x++y,b)
```

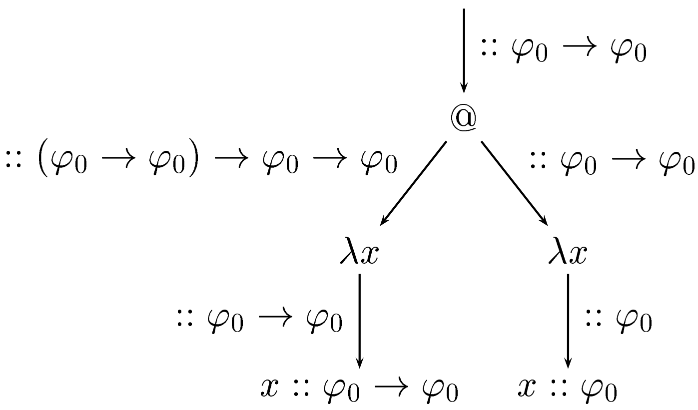
⋮

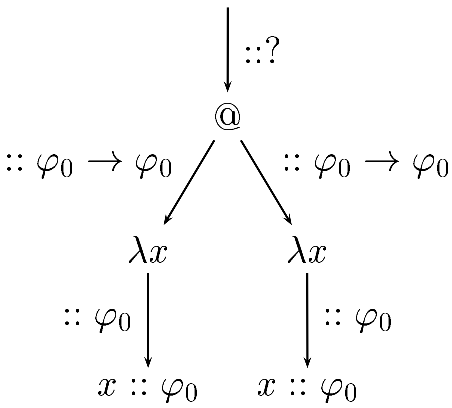
Inferencia de tipos de datos [11, pp.188–203]

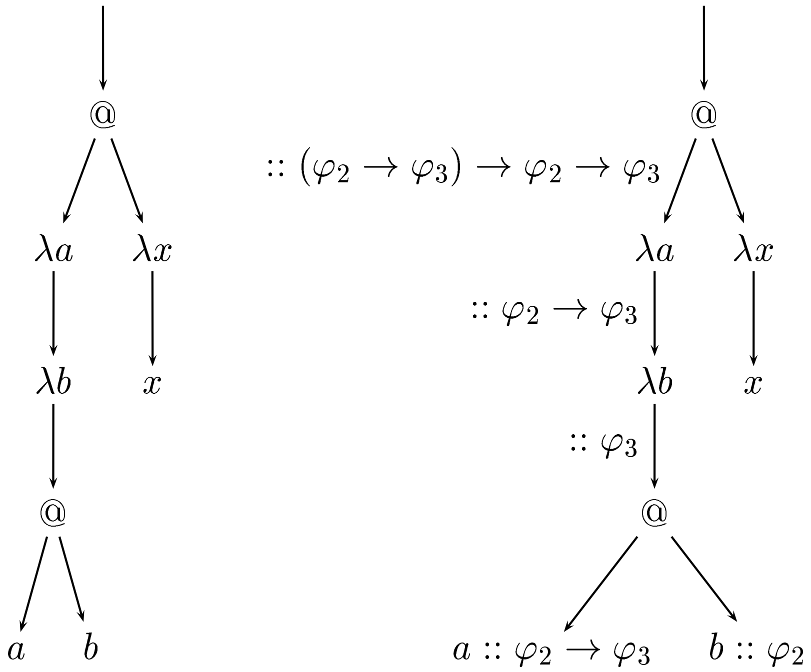


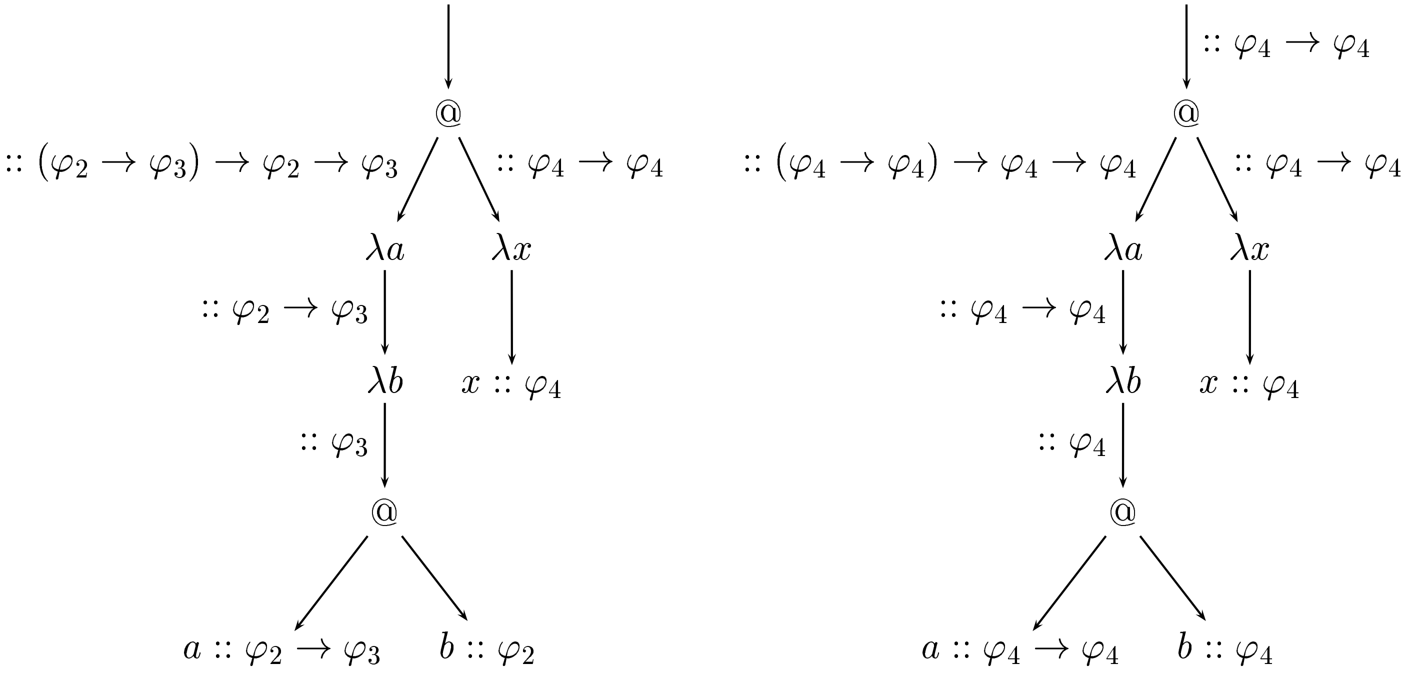


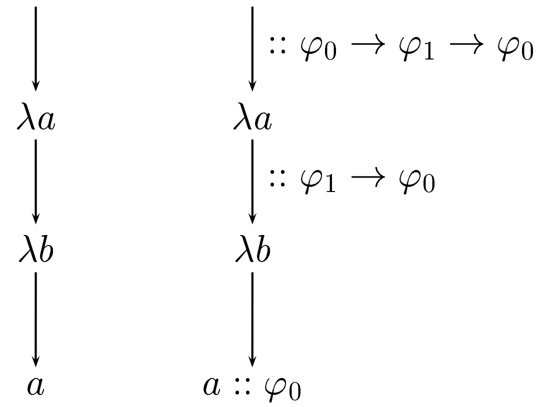







@
λa λx
λb x
@
a b
@
$:: (\varphi_2 \to \varphi_3) \to \varphi_2 \to \varphi_3$
λa λx
$:: \varphi_2 \to \varphi_3$
λb x
$:: \varphi_3$
@
$a :: \varphi_2 \to \varphi_3$ $b :: \varphi_2$


@
$:: (\varphi_2 \to \varphi_3) \to \varphi_2 \to \varphi_3$
$:: \varphi_4 \to \varphi_4$
λa λx
$:: \varphi_2 \to \varphi_3$
λb $x :: \varphi_4$
$:: \varphi_3$
@
$a :: \varphi_2 \to \varphi_3$ $b :: \varphi_2$
$:: \varphi_4 \to \varphi_4$
@
$:: (\varphi_4 \to \varphi_4) \to \varphi_4 \to \varphi_4$
$:: \varphi_4 \to \varphi_4$
λa λx
$:: \varphi_4 \to \varphi_4$
λb $x :: \varphi_4$
$:: \varphi_4$
@
$a :: \varphi_4 \to \varphi_4$ $b :: \varphi_4$


λa
λb
a
$:: \varphi_0 \to \varphi_1 \to \varphi_0$
λa
$:: \varphi_1 \to \varphi_0$
λb
$a :: \varphi_0$

```
module Unification where

data Type = Typevar Tvar | Arrow Type Type  deriving(Eq,Show)
type Tvar = Int
type Substitution = Type -> Type

unify :: Type -> Type -> Substitution
unify (Typevar phi0) (Typevar phi1) = sub phi0 (Typevar phi1)
unify (Typevar phi)  tau = if not_in tau phi then sub phi tau
                                             else error "no unifican"
unify sigma          (Typevar phi)  = unify (Typevar phi) sigma
unify (Arrow sigma0 sigma1)
      (Arrow tau0 tau1) = res1.res0
                          where res0 = unify sigma0 tau0
                                res1 = unify (res0 sigma1) (res0 tau1)

sub :: Tvar -> Type -> Substitution
sub phi0 tau = s
  where s (Typevar phi1) = if phi0 == phi1 then tau
                                           else Typevar phi1
        s (Arrow sigma rho) = Arrow ((sub phi0 tau) sigma)
                                    ((sub phi0 tau) rho)

not_in :: Type -> Tvar -> Bool
not_in (Typevar phi0) phi1 = phi1 /= phi0
not_in (Arrow sigma tau) phi1 = not_in sigma phi1 && not_in tau phi1

> (unify (Typevar 2) (Arrow (Typevar 4) (Typevar 3))) (Typevar 2)
Arrow (Typevar 4) (Typevar 3)

> (unify (Typevar 2) (Arrow (Typevar 2) (Typevar 3))) (Typevar 2)
Program error: no unifican

> map (unify (Arrow (Typevar 2) (Typevar 3)) (Arrow (Typevar 4) (Typevar 5)))
      [Typevar 1,Typevar 2,Typevar 3,Typevar 4]
[Typevar 1,Typevar 4,Typevar 5,Typevar 4]

t1 = (Arrow (Typevar 2) (Arrow (Typevar 3) (Typevar 3)))
t2 = (Arrow (Arrow (Typevar 0) (Typevar 1)) (Typevar 0))
s = unify t1 t2
> map s [Typevar 0, Typevar 1, Typevar 2, Typevar 3]
[Arrow (Typevar 3) (Typevar 3),
 Typevar 1,
 Arrow (Arrow (Typevar 3) (Typevar 3)) (Typevar 1),
 Typevar 3]
```

```
Curry-type assignment algorithm (p.196, Plasmeijer)

data Lterm = Var V | Abstr V Lterm | Ap Lterm Lterm  deriving(Eq,Show)
type V = Char
type Assumption = (V,Type)
type Basis = [Assumption]
type FreshTvars = [Tvar]
type Tresults = (FreshTvars,Basis,Type)

curry' :: FreshTvars -> Lterm -> Tresults
curry' (phi:ftvs) (Var x)
  = (ftvs,[(x,Typevar phi)], Typevar phi)
curry' (phi:ftvs) (Ap f g)
  = (ftvs'', basis, s (Typevar phi))
    where
      basis = substitute s (basleft ++ basright)
      s     = unify rho (Arrow sigma (Typevar phi))
      (ftvs',basleft,rho) = curry' ftvs f
      (ftvs'',basright,sigma) = curry' ftvs' g
curry' (phi:ftvs) (Abstr x g)
  = (ftvs',basis,s (Arrow (Typevar phi) tau))
    where
      (basis,s) = check_assmpt (x,Typevar phi) bas
      (ftvs',bas,tau) = curry' ftvs g

substitute :: Substitution -> Basis -> Basis
substitute s [] = []
substitute s ((x,sigma):rest) = (x, s sigma):substitute s rest

check_assmpt :: Assumption -> Basis -> (Basis,Substitution)
check_assmpt ass [] = ([],identsub)
check_assmpt (x,sigma) ((y,tau):rest)
  = if y==x then (substitute (s2.s1) bas1, s2.s1)
            else (substitute s ((y,tau):bas), s)
  where s1        = unify sigma tau
        (bas1,s2) = check_assmpt (x, s1 sigma) rest
        (bas,s)   = check_assmpt (x, sigma) rest

identsub :: Substitution
identsub  = id
```

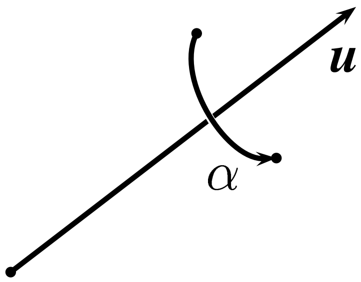
```
ftvs :: [Tvar]
ftvs = [0..10]
lterm = Abstr 'f' (Ap (Ap (Var 'f') (Var 'x'))  (Abstr 'g' (Var 'z')) )
```

```
> curry' ftvs lterm
([7,8,9,10],
 [('x',Typevar 4),('z',Typevar 6)],
 Arrow
   (Arrow (Typevar 4) (Arrow (Arrow (Typevar 5) (Typevar 6)) (Typevar 1)))
   (Typevar 1))
```

```
              ( a4 ->  ( (a5->a6) -> a1 ) ) -> a1
```

```
> :t (\x z -> (\f -> f x (\g->z)))
\x z -> \f -> f x (\g -> z) :: a -> b -> (a -> (c -> b) -> d) -> d
```

```
              (  a ->    ( c-> b) -> d  )   -> d
              (  a ->  ( ( c-> b) -> d  ) ) -> d
              ( a4 ->  ( (a5->a6) -> a1 ) ) -> a1
```



Teorema 0.1 *La matriz de la rotación de ángulo α alrededor del eje $\boldsymbol{u}$ es:*

$$\begin{pmatrix} a^2+b^2-c^2-d^2 & 2(bc-ad) & 2(bd+ac) \\ 2(bc+ad) & a^2-b^2+c^2-d^2 & 2(cd-ab) \\ 2(bd-ac) & 2(cd+ab) & a^2-b^2-c^2+d^2 \end{pmatrix}$$

en que

$$a = \cos(\alpha/2) \qquad y \qquad (b,c,d) = \operatorname{sen}(\alpha/2)\,\frac{\boldsymbol{u}}{\|\boldsymbol{u}\|}$$

$\square$

$$\mathcal{H} = \{a + x\,\boldsymbol{i} + y\boldsymbol{j} + z\,\boldsymbol{k} \mid a, x, y, z \in \mathbb{R};\ \boldsymbol{i}^2 = \boldsymbol{j}^2 = \boldsymbol{k}^2 = -1;$$
$$\boldsymbol{ij} = -\boldsymbol{ji} = \boldsymbol{k},\ \boldsymbol{jk} = -\boldsymbol{kj} = \boldsymbol{i},\ \boldsymbol{ki} = -\boldsymbol{ik} = \boldsymbol{j}\}$$

Sea $q = a + \boldsymbol{v}$ $\quad$ (con $\boldsymbol{v} = x\,\boldsymbol{i} + y\boldsymbol{j} + z\,\boldsymbol{k}$).

Definición: $\overline{q} = a - \boldsymbol{v}$

Definición: $\|q\|^2 = q\overline{q} \quad (= a^2 + x^2 + y^2 + z^2)$

Propiedades 0.2

i. $\boldsymbol{v} = -\overline{\boldsymbol{v}}$

ii. $q \in \mathbb{R}^3 \Leftrightarrow q = -\overline{q}$

iii. $(a_1 + \boldsymbol{v}_1)(a_2 + \boldsymbol{v}_2) = (a_1 a_2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2) + a_1\,\boldsymbol{v}_2 + a_2\,\boldsymbol{v}_1 + \boldsymbol{v}_1 \times \boldsymbol{v}_2$

iv. $\boldsymbol{v}_1\boldsymbol{v}_2 = -\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 + \boldsymbol{v}_1 \times \boldsymbol{v}_2$

v. $\overline{q_1\, q_2} = \overline{q_2}\ \overline{q_1}$

vi. $\boldsymbol{v}_1 \perp \boldsymbol{v}_2 \Leftrightarrow \boldsymbol{v}_1\boldsymbol{v}_2 = \boldsymbol{v}_1 \times \boldsymbol{v}_2$

$\square$

Definición: Sea $\boldsymbol{p} \in \mathbb{R}^3$, $\|\boldsymbol{p}\| = 1$. $\quad T_{\boldsymbol{p}} \colon \mathbb{R}^3 \to \mathbb{R}^3$, $\ T_{\boldsymbol{p}}(\boldsymbol{v}) = \boldsymbol{p}\,\boldsymbol{v}\,\boldsymbol{p}$

Propiedades 0.3

i. $\overline{T_{\boldsymbol{p}}(\boldsymbol{v})} = \overline{\boldsymbol{p}\,\boldsymbol{v}\,\boldsymbol{p}} = \overline{\boldsymbol{p}}\ \overline{\boldsymbol{v}}\ \overline{\boldsymbol{p}} = -\boldsymbol{p}\,\boldsymbol{v}\,\boldsymbol{p} = -T_{\boldsymbol{p}}(\boldsymbol{v})$

ii. $T_{\boldsymbol{p}}(\boldsymbol{v}_1 + \boldsymbol{v}_2) = T_{\boldsymbol{p}}(\boldsymbol{v}_1) + T_{\boldsymbol{p}}(\boldsymbol{v}_2), \quad T_{\boldsymbol{p}}(\lambda\,\boldsymbol{v}) = \lambda T_{\boldsymbol{p}}(\boldsymbol{v})$

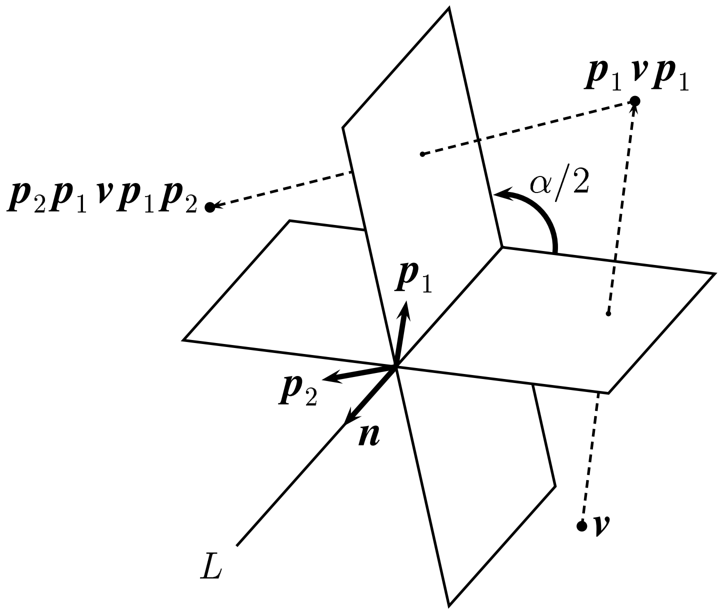
iii. $\boldsymbol{v} \perp \boldsymbol{p} \Rightarrow T_{\boldsymbol{p}}(\boldsymbol{v}) = \boldsymbol{v}$

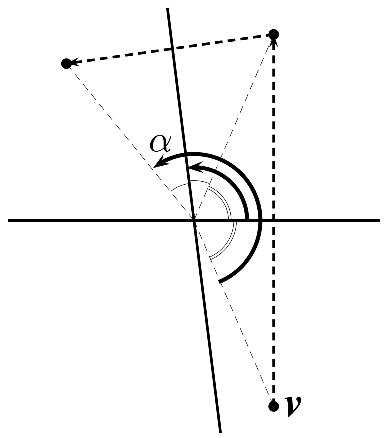
iv. $T_{\boldsymbol{p}}(\boldsymbol{p}) = \boldsymbol{p}\,\boldsymbol{p}\,\boldsymbol{p} = (-\boldsymbol{p}\,\overline{\boldsymbol{p}})\,\boldsymbol{p} = -\boldsymbol{p}$

v. $T_{\boldsymbol{p}}$ es una reflexión en el plano perpendicular a $\boldsymbol{p}$

$\square$

Toda rotación (de ángulo α) alrededor de una recta L puede verse como una composición de dos reflexiones, en planos que se cortan en L y forman un ángulo $\alpha/2$ (perpendiculares a $\boldsymbol{p}_1$ y $\boldsymbol{p}_2$, respect.)





Tenemos entonces que la rotación es tal que

$$\boldsymbol{v} \mapsto \boldsymbol{p}_2\boldsymbol{p}_1\,\boldsymbol{v}\,\boldsymbol{p}_1\boldsymbol{p}_2 = \boldsymbol{p}_2\boldsymbol{p}_1\,\boldsymbol{v}\,\overline{\boldsymbol{p}_2\boldsymbol{p}_1} = (-\boldsymbol{p}_2\boldsymbol{p}_1)\,\boldsymbol{v}\,\overline{(-\boldsymbol{p}_2\boldsymbol{p}_1)}$$

Definiendo

$$\begin{aligned} q &= -\boldsymbol{p}_2\boldsymbol{p}_1 \\ &= -[-\boldsymbol{p}_2 \cdot \boldsymbol{p}_1 + \boldsymbol{p}_2 \times \boldsymbol{p}_1] \\ &= \boldsymbol{p}_1 \cdot \boldsymbol{p}_2 + \boldsymbol{p}_1 \times \boldsymbol{p}_2 \\ &= \cos(\alpha/2) + \operatorname{sen}(\alpha/2)\boldsymbol{n} \end{aligned}$$

y llamando R_q a la rotación, hemos obtenido que

$$R_q(\boldsymbol{v}) = q\,\boldsymbol{v}\,\bar{q}$$

Algunas definiciones en *Haskell*

```
type Angulo = R             -- (R y R3 como en págs. 2 y 5)
type Eje = R3               --
type Rotacion = R3 -> R3

rotacion :: Angulo -> Eje -> Rotacion
rotacion angulo eje = ( (matRot angulo eje) *** )
```

```
type MatrizRot = [R3]

matRot :: Angulo -> Eje -> MatrizRot
matRot angulo eje =
  [ ( a^2+b^2-c^2-d^2 , 2*(b*c-a*d)     , 2*(a*c+b*d)     ),
    ( 2*(a*d+b*c)     , a^2-b^2+c^2-d^2 , 2*(c*d-a*b)     ),
    ( 2*(b*d-a*c)     , 2*(a*b+c*d)     , a^2-b^2-c^2+d^2 ) ]
  where
    (a,b,c,d) = ( cos beta, k*x, k*y, k*z )
    beta = angulo/2
    k = sin beta / norma eje                     -- (ver pág. 5)
    (x,y,z) = eje
```

```
(***) :: MatrizRot -> R3 -> R3
m *** p = (x,y,z)
          where [x,y,z] = [ r <*> p | r<-m ]  -- (ver pág. 5)
```

```
ejeAlfa :: Eje
ejeAlfa = (0.0, 0.0, 1.0)

rotAlfa :: Rotacion
rotAlfa = rotacion ((2.0*pi)/5.0) ejeAlfa

ptoB = (0.607,0.0,0.795)

> rotAlfa ptoB
(0.188,0.577,0.795)
```

La función `rotacion` también se puede definir directamente en términos del álgebra de cuaternios, de acuerdo a la def. de R_q:

```
type Cuaternio = (R,R,R,R)

(|*|) :: Cuaternio -> Cuaternio -> Cuaternio
(a1,x1,y1,z1) |*| (a2,x2,y2,z2) = (a,x,y,z)
  where a = a1*a2 - v1<*>v2                          -- (ver pág. 5)
        (x,y,z) = a1<.>v2 + a2<.>v1 + v1>*<v2  --
        v1 = (x1,y1,z1)
        v2 = (x2,y2,z2)

conjugado :: Cuaternio -> Cuaternio
conjugado (a,x,y,z) = (a,-x,-y,-z)

consCuaternio :: Angulo -> Eje -> Cuaternio
consCuaternio angulo eje = ( cos beta, k*x, k*y, k*z )
  where beta = angulo/2
        k = sin beta / norma eje
        (x,y,z) = eje
```

```
r :: Cuaternio -> R3 -> R3                             -- (Rq)
r q v = parteVec ( q |*| v’ |*| conjugado q )
  where parteVec (a,x,y,z) = (x,y,z)
        v’ = let (x,y,z) = v in (0,x,y,z)

rotacion :: Angulo -> Eje -> Rotacion
rotacion angulo eje = r (consCuaternio angulo eje)
```

```
rotAlfa :: Rotacion
rotAlfa = rotacion ((2.0*pi)/5.0) ejeAlfa

> take 5 (iterate rotAlfa ptoB)
[(0.607,0.0,0.795), (0.188,0.577,0.795), (-0.491,0.357,0.795),
 (-0.491,-0.359,0.795), (0.188,-0.577,0.795)]
```

Referencias y biblografía

[1] R. Bird and O. de Moor. *Algebra of Programming.* C.A.R. Hoare series editor. Prentice Hall, 1997.

[2] R. Bird and P. Wadler. *Intr. to Functional Programming.* Prentice Hall, 1987.

[3] A.J.T. Davie. *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, 1992.

[4] P. Hudak and J.H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5):T1–T53, May 1992.

[5] Mark P. Jones. Functional programming with overloading and higher–order polymorphism. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 97–136. Springer–Verlag, 1995.

[6] R. Lalement. *Computation as Logic.* C.A.R. Hoare series editor. Prentice Hall, 1993.

[7] John Launchbury. Graph algorithms with a functional flavour. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 308–331. Springer–Verlag, 1995.

[8] L.C. Paulson. Foundations of functional programming. Technical report, Computer Laboratory, University of Cambridge, 1995. (http://www.cl.cam.ac.uk/users/lcp/papers/Notes/Founds-FP.pdf).

[9] J. Peterson and K. Hammond (editors) et al. Report on the programming language Haskell, a non–strict purely functional language (version 1.4). Technical report, Yale University and University of St. Andrews, April 1997. (http://haskell.org).

[10] B.C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[11] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison–Wesley, 1993.

[12] P.A. Strooper and M.H. van Emden. Describing symmetrical structures in logic. Technical report, Department of Computer Science, University of Victoria, February 1992. Report LP–20.

[13] C. Velarde. Construcción de imágenes de estructuras simétricas mediante programación funcional. In *Memorias del Congreso Visual–98*, DGSCA, UNAM, abril 1998.

[14] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, number 925 in Lecture Notes in Computer Science, pages 24–52. Springer–Verlag, 1995.