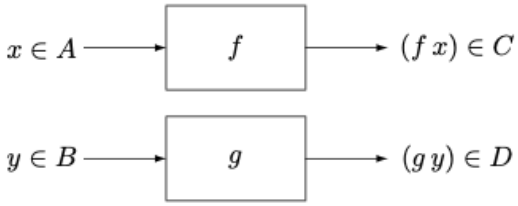
2. O diagrama de blocos



descreve o combinador funcional *produto*

$$f \times g = \langle f \cdot \pi_1, g \cdot \pi_2 \rangle \tag{F1}$$

que capta a aplicação *paralela* e *independente* de duas funções $A \xrightarrow{f} C$ e $B \xrightarrow{g} D$:

$$\begin{array}{ccc} A & B & A \times B \\ {\scriptstyle f}\downarrow & {\scriptstyle g}\downarrow & \downarrow{\scriptstyle f \times g} \\ C & D & C \times D \end{array}$$

(a) Mostre que $(f \times g)\ (x, y) = (f\ x, g\ y)$.

(b) Mostre ainda que

$$\pi_1 \cdot (f \times g) = f \cdot \pi_1 \tag{F2}$$
$$\pi_2 \cdot (f \times g) = g \cdot \pi_2 \tag{F3}$$
$$id \times id = id \tag{F4}$$
$$(f \times g) \cdot (h \times k) = f \cdot h \times g \cdot k \tag{F5}$$

Desenhe os diagramas destas igualdades e anime-as em Haskell, para f, g, h e k à sua escolha.

Resolução

(a) Mostre que $(f \times g)\ (x, y) = (f\ x, g\ y)$.

$(f \times g)\ (x, y)$

{ lei (10) }

$= \ < f \cdot \pi_1, g \cdot \pi_2 >\ (x, y)$

{ def. split }

$= \ ((f \cdot \pi_1)\ (x, y), (g \cdot \pi_2)\ (x, y))$

{ lei (72) }

$= \ (f(\pi_1\ (x, y)), g(\pi_2\ (x, y)))$

{ def. π_1, def. π_2, lei (79) }

$(f\ x, g\ y)$

In [1]:
```haskell
f >< g = split (f . p1) (g . p2)
    where
        p1 = fst
        p2 = snd
        split f g x = (f x, g x)

-- type checking
```

```
:t (><)

-- testing with f = succ; g = length; (x,y) = (2,[3,4,5])

(succ >< length) (2,[3,4,5]) == (succ 2, length [3,4,5])
```

```
(><) :: forall a1 a2 b1 b2. (a1 -> a2) -> (b1 -> b2) -> (a1, b1) ->
(a2, b2)
```

```
True
```

Resolução

(b) Mostre ainda que

$$\pi_1 \cdot (f \times g) = f \cdot \pi_1 \quad \text{(F2)}$$
$$\pi_2 \cdot (f \times g) = g \cdot \pi_2 \quad \text{(F3)}$$
$$id \times id = id \quad \text{(F4)}$$
$$(f \times g) \cdot (h \times k) = f \cdot h \times g \cdot k \quad \text{(F5)}$$

In [2]:

```
-- (F2) --

f = succ
g = length
p1 = fst
p2 = snd

-- type checking

:t p1 . (f >< g)
:t f . p1

-- testing with (x,y) = (2,[1,2,3])

(p1 . (f >< g)) (2,[1,2,3]) == (f . p1) (2,[1,2,3])
```

```
p1 . (f >< g) :: forall c (t :: * -> *) a. (Enum c, Foldable t) => (c,
t a) -> c
```

```
f . p1 :: forall c b. Enum c => (c, b) -> c
```

```
True
```

In [3]:

```
-- (F3) --

f = succ
g = length
p1 = fst
p2 = snd

-- type checking

:t p2 . (f >< g)
:t g . p2

-- testing

(p2 . (f >< g)) (2,[1,2,3]) == (g . p2) (2,[1,2,3])
```

```
p2 . (f >< g) :: forall a1 (t :: * -> *) a2. (Enum a1, Foldable t) =>
```

```
(a1, t a2) -> Int

g . p2 :: forall (t :: * -> *) a1 a2. Foldable t => (a1, t a2) -> Int
```

True

In [4]:

```
-- (F4) --

id x = x

-- testing

(id >< id) (2,[1,2,3]) == (2,[1,2,3])
```

True

In [5]:

```
-- (F5) --

f = succ
g = succ
h = double where double x = x * 2
k = length

-- type checking

:t (f >< g)
:t (h >< k)

-- testing

((f >< g) . (h >< k)) (2,[1,2,3]) == ((f . h) >< (g . k)) (2,[1,2,3])
```

```
(f >< g) :: forall a2 b2. (Enum a2, Enum b2) => (a2, b2) -> (a2, b2)

(h >< k) :: forall a2 (t :: * -> *) a. (Num a2, Foldable t) => (a2, t
a) -> (a2, Int)
```

True

In [6]:

```
-- applying

((f >< g) . (h >< k) ) (2,[1,2,3])
```

(5,4)