Cálculo de Programas
T10 (cont.)

(a) Trees with data in their leaves :

$$\mathsf{T} = \mathsf{LTree}\ A \qquad \begin{cases} \mathsf{F}\ X = A + X^2 \\ \mathsf{F}\ f = id + f^2 \end{cases} \qquad \mathsf{in} = [Leaf\ , Fork]$$

Haskell: $\mathbf{data}\ \mathsf{LTree}\ a = Leaf\ a \mid Fork\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a)$

(b) Trees whose data of type A are stored in their nodes:

$$\mathsf{T} = \mathsf{BTree}\ A \qquad \begin{cases} \mathsf{F}\ X = 1 + A \times X^2 \\ \mathsf{F}\ f = id + id \times f^2 \end{cases} \qquad \mathsf{in} = [\underline{Empty}\ , Node]$$

Haskell: $\mathbf{data}\ \mathsf{BTree}\ a = Empty \mid Node\ (a, (\mathsf{BTree}\ a, \mathsf{BTree}\ a))$

(c) Full trees — data in both leaves and nodes:

$$\mathsf{T} = \mathsf{FTree}\ B\ A \qquad \begin{cases} \mathsf{F}\ X = B + A \times X^2 \\ \mathsf{F}\ f = id + id \times f^2 \end{cases} \qquad \mathsf{in} = [Unit\ , Comp]$$

Haskell: $\mathbf{data}\ \mathsf{FTree}\ b\ a = Unit\ b \mid Comp\ (a, (\mathsf{FTree}\ b\ a, \mathsf{FTree}\ b\ a))$

(d) Expression trees:

$$\mathsf{T} = \mathsf{Expr}\ V\ O \qquad \begin{cases} \mathsf{F}\ X = V + O \times X^* \\ \mathsf{F}\ f = id + id \times \mathsf{map}\ f \end{cases} \qquad \mathsf{in} = [Var\ , Term]$$

Haskell: $\mathbf{data}\ \mathsf{Expr}\ v\ o = Var\ v \mid Term\ (o, [\mathsf{Expr}\ v\ o])$

(a) Trees with data in their leaves :

$$\mathsf{T} = \mathsf{LTree}\ A \qquad \begin{cases} \mathsf{B}\ (X, Y) = X + Y^2 \\ \mathsf{B}\ (g, f) = g + f^2 \end{cases} \qquad \mathsf{in} = [Leaf\ , Fork]$$

Haskell: $\mathbf{data}\ \mathsf{LTree}\ a = Leaf\ a \mid Fork\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a)$

(b) Trees whose data of type A are stored in their nodes:

$$\mathsf{T} = \mathsf{BTree}\ A \qquad \begin{cases} \mathsf{B}\ (X, Y) = 1 + X \times Y^2 \\ \mathsf{B}\ (g, f) = id + g \times f^2 \end{cases} \qquad \mathsf{in} = [\underline{Empty}\ , Node]$$

Haskell: $\mathbf{data}\ \mathsf{BTree}\ a = Empty \mid Node\ (a, (\mathsf{BTree}\ a, \mathsf{BTree}\ a))$

(c) Full trees — data in both leaves and nodes:

$$\mathsf{T} = \mathsf{FTree}\ B\ A \qquad \begin{cases} \mathsf{B}\ (Z, X, Y) = Z + X \times Y^2 \\ \mathsf{B}\ (h, g, f) = h + g \times f^2 \end{cases} \qquad \mathsf{in} = [Unit\ , Comp]$$

Haskell: $\mathbf{data}\ \mathsf{FTree}\ b\ a = Unit\ b \mid Comp\ (a, (\mathsf{FTree}\ b\ a, \mathsf{FTree}\ b\ a))$

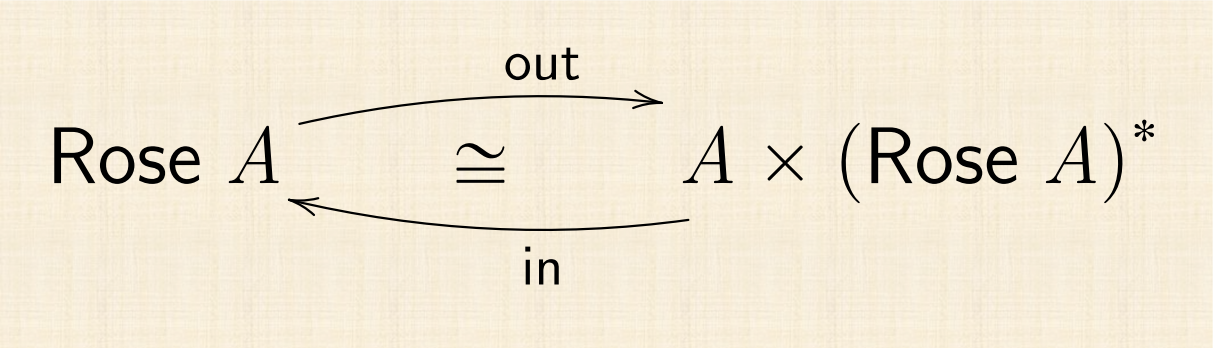
(d) Expression trees:

$$\mathsf{T} = \mathsf{Expr}\ V\ O \qquad \begin{cases} \mathsf{B}\ (Z, X, Y) = Z + X \times Y^* \\ \mathsf{B}\ (h, g, f) = h + g \times \mathsf{map}\ f \end{cases} \qquad \mathsf{in} = [Var\ , Term]$$

Haskell: $\mathbf{data}\ \mathsf{Expr}\ v\ o = Var\ v \mid Term\ (o, [\mathsf{Expr}\ v\ o])$

‘Chasing’ the base functor

```haskell
data Rose a = Rose a [Rose a] deriving Show

inRose = uncurry Rose

outRose (Rose a x) = (a,x)
```



$$\text{Rose } A \underset{\text{in}}{\overset{\text{out}}{\cong}} A \times (\text{Rose } A)^*$$

$$\mathbf{B}\,(X, Y) = X \times Y^*$$

$$\mathbf{B}\,(f, g) = f \times g^*$$

$$A \times Y^*$$

$$X \times Y^*$$

```haskell
data Rose a = Rose a [Rose a] deriving Show

inRose = uncurry Rose

outRose (Rose a x) = (a,x)
```

$$\mathbf{B}\,(X, Y) = X \times Y^*$$
$$\mathbf{B}\,(f, g) = f \times g^*$$

```haskell
f >< map g
```

```haskell
data Rose a = Rose a [Rose a] deriving Show

inRose = uncurry Rose

outRose (Rose a x) = (a,x)
```

```haskell
data Rose a = Rose a [Rose a] deriving Show

inRose = uncurry Rose

outRose (Rose a x) = (a,x)
```

```haskell
baseRose f g = f >< map g

recRose g = baseRose id g
```

```haskell
data Rose a = Rose a [Rose a] deriving Show

inRose = uncurry Rose

outRose (Rose a x) = (a,x)
```

```haskell
baseRose f g = f >< map g

recRose g = baseRose id g
```

```haskell
cataRose g = g . (recRose (cataRose g)) . outRose
```

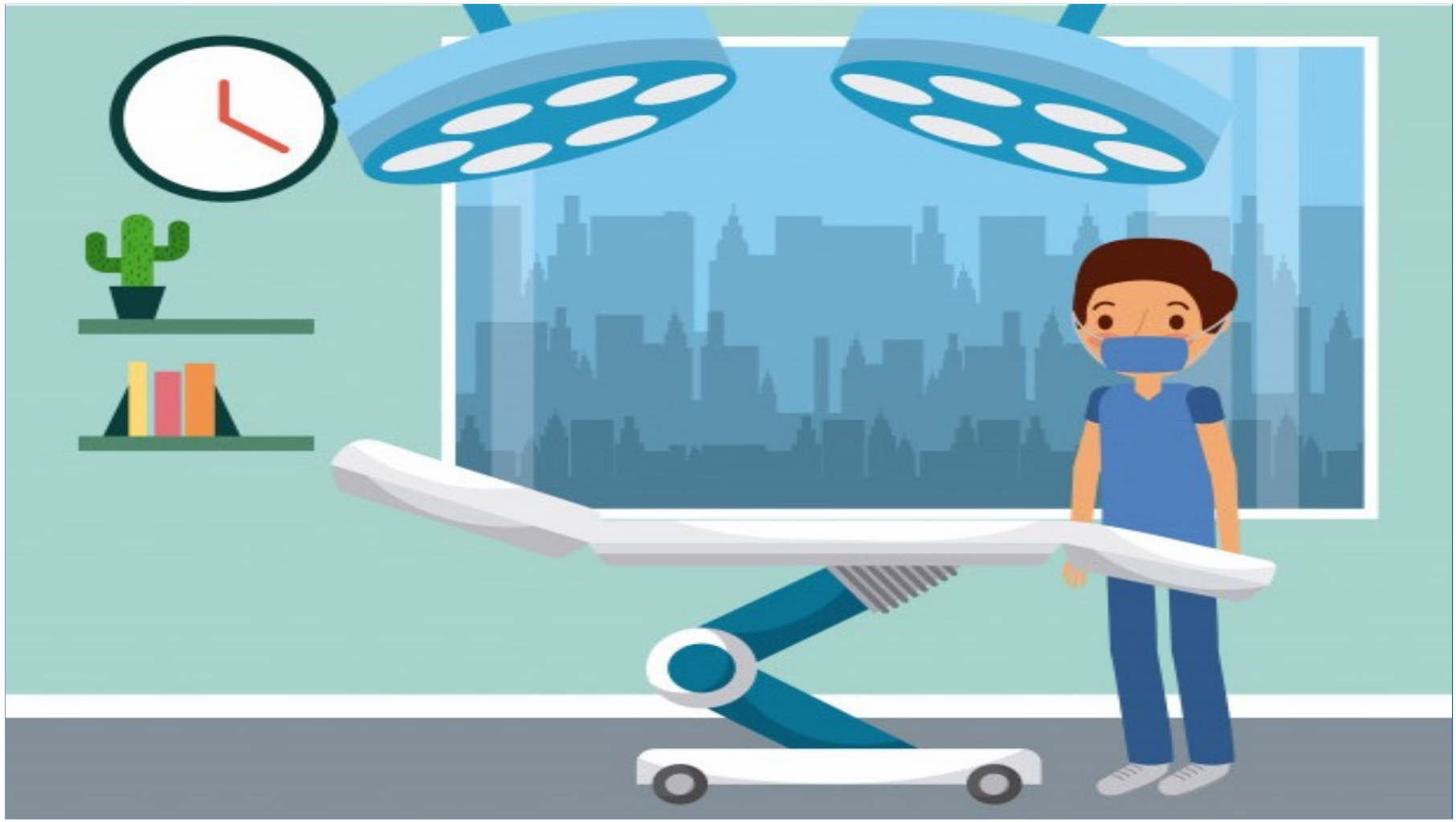
```haskell
data Rose a = Rose a [Rose a] deriving Show

inRose = uncurry Rose

outRose (Rose a x) = (a,x)
```

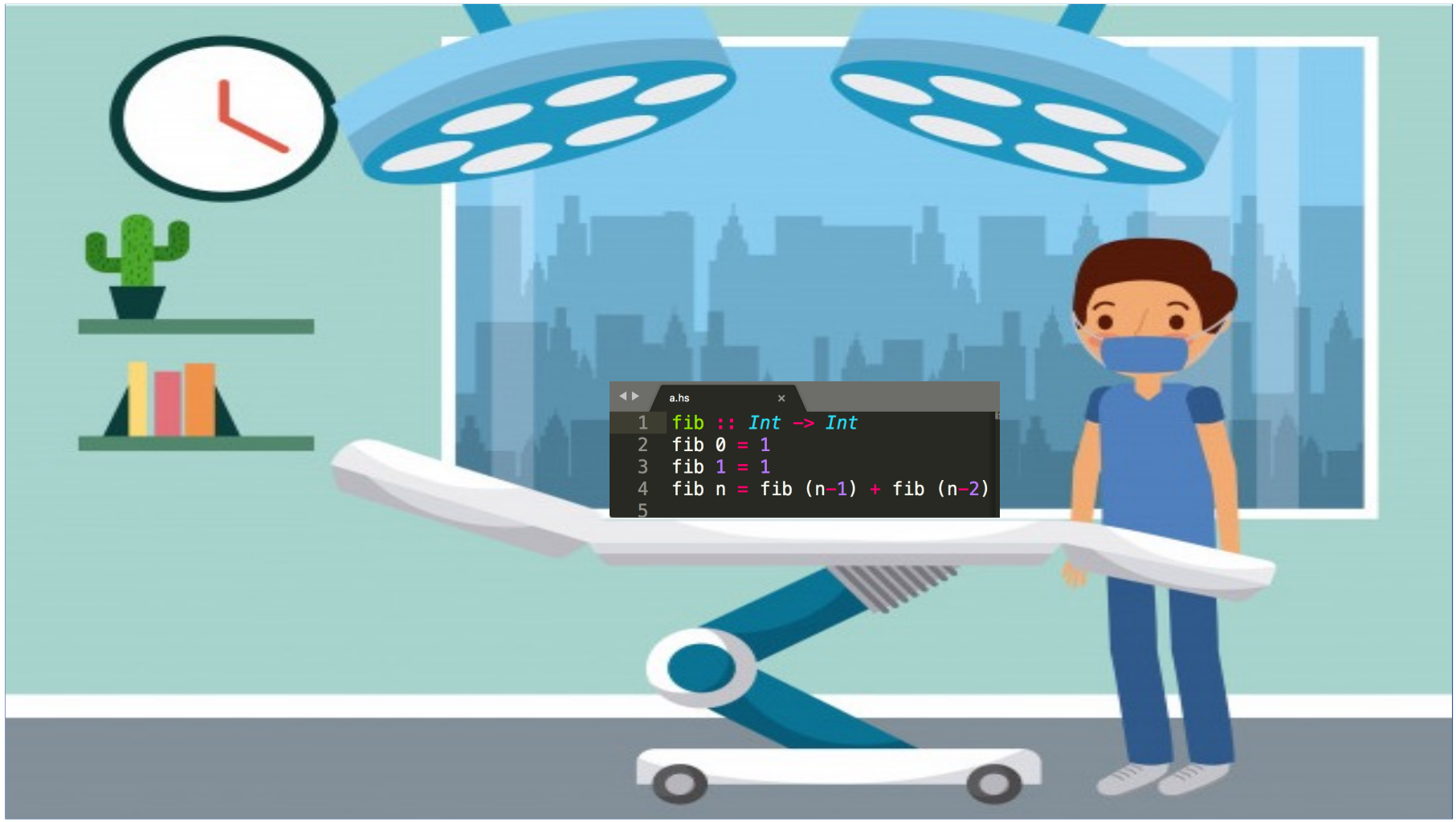
```haskell
baseRose f g = f >< map g

recRose g = baseRose id g
```

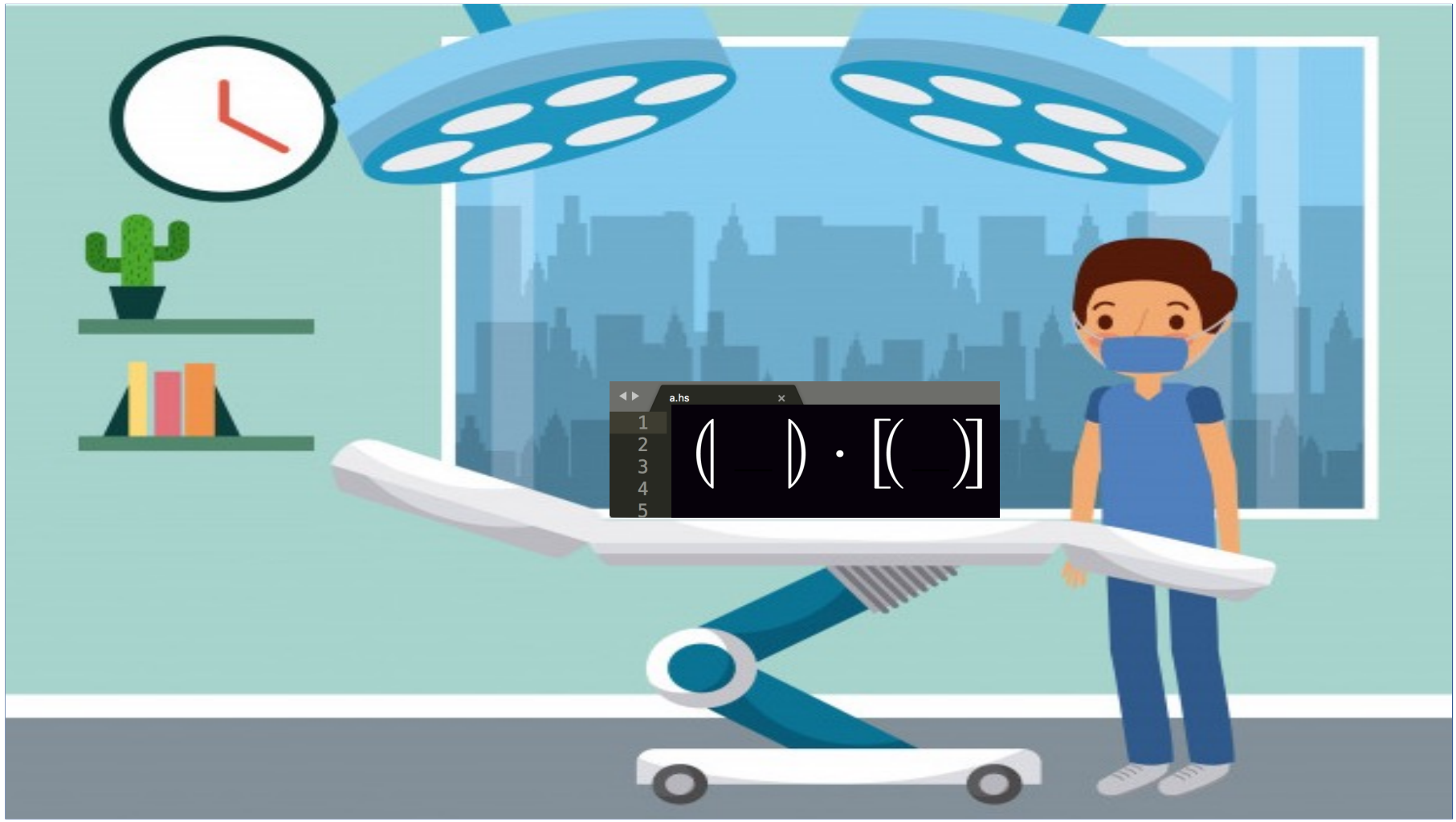
```haskell
cataRose g = g . (recRose (cataRose g)) . outRose
```

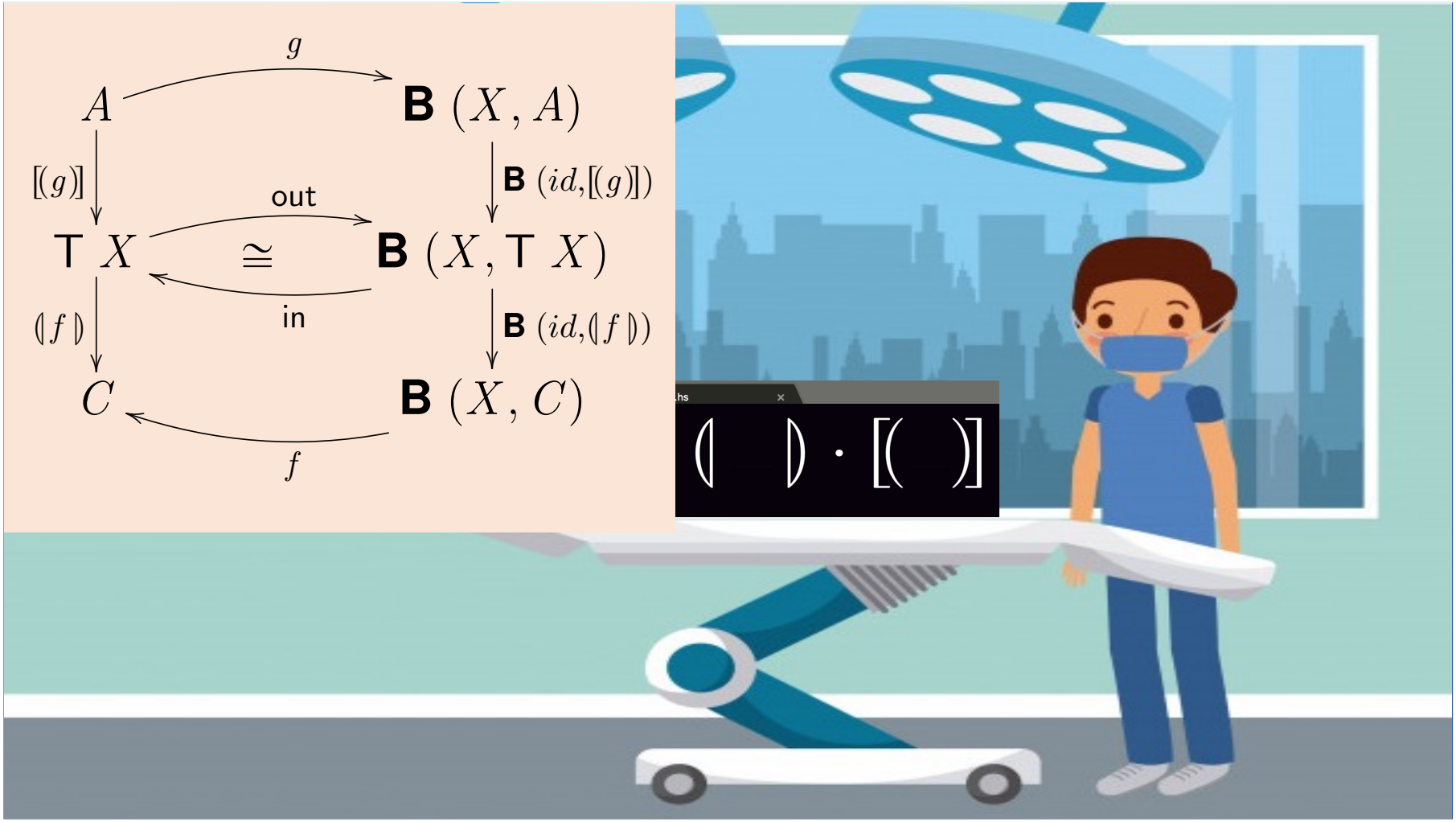
```haskell
instance Functor Rose
         where fmap f = cataRose ( inRose . baseRose f id )
```

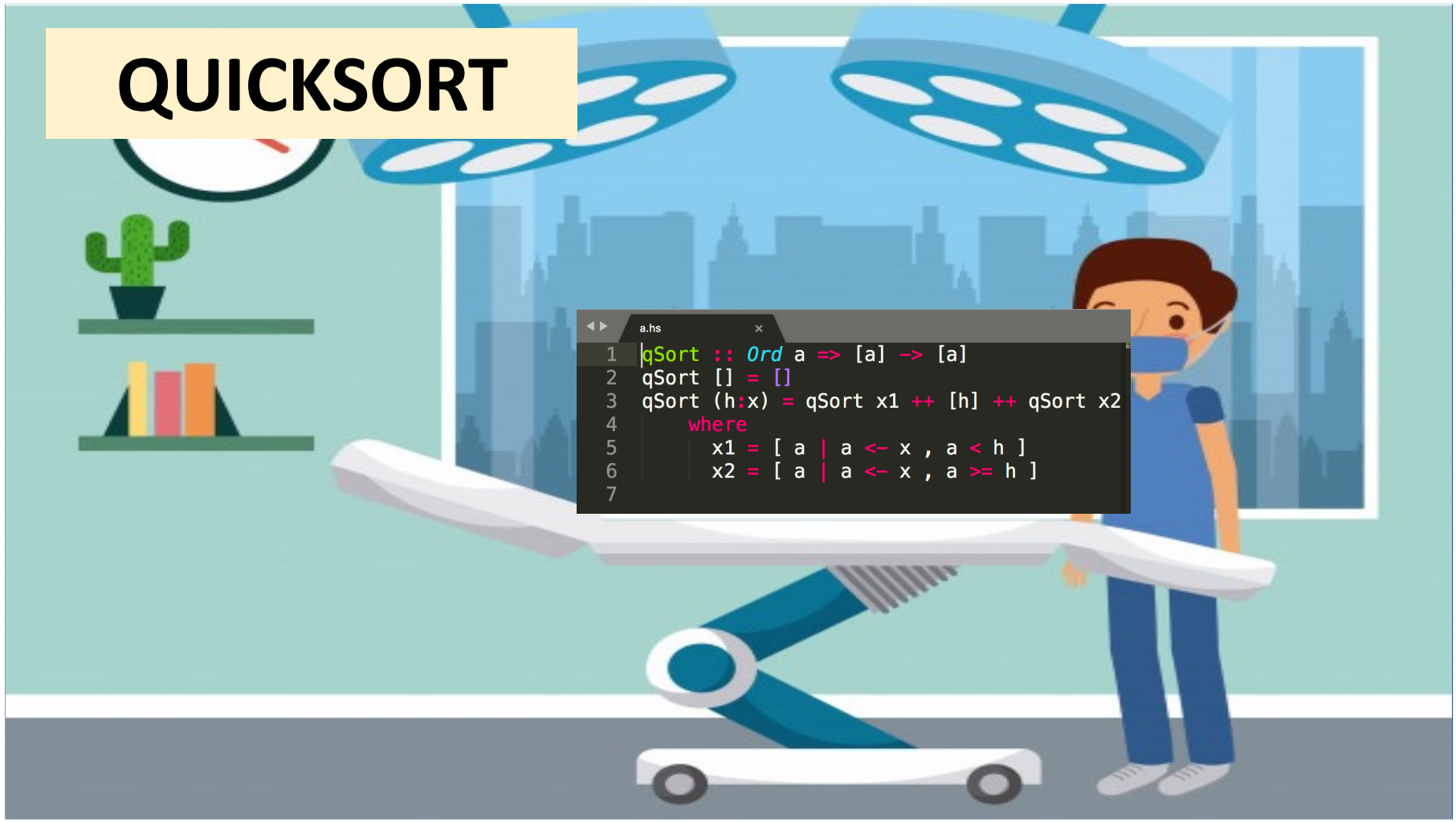
Algorithmic *Hylo-Factorization* (examples)




a.hs
1 fib :: Int -> Int
2 fib 0 = 1
3 fib 1 = 1
4 fib n = fib (n-1) + fib (n-2)
5


a.hs
1
2
3
4
5
( ) · [( )]

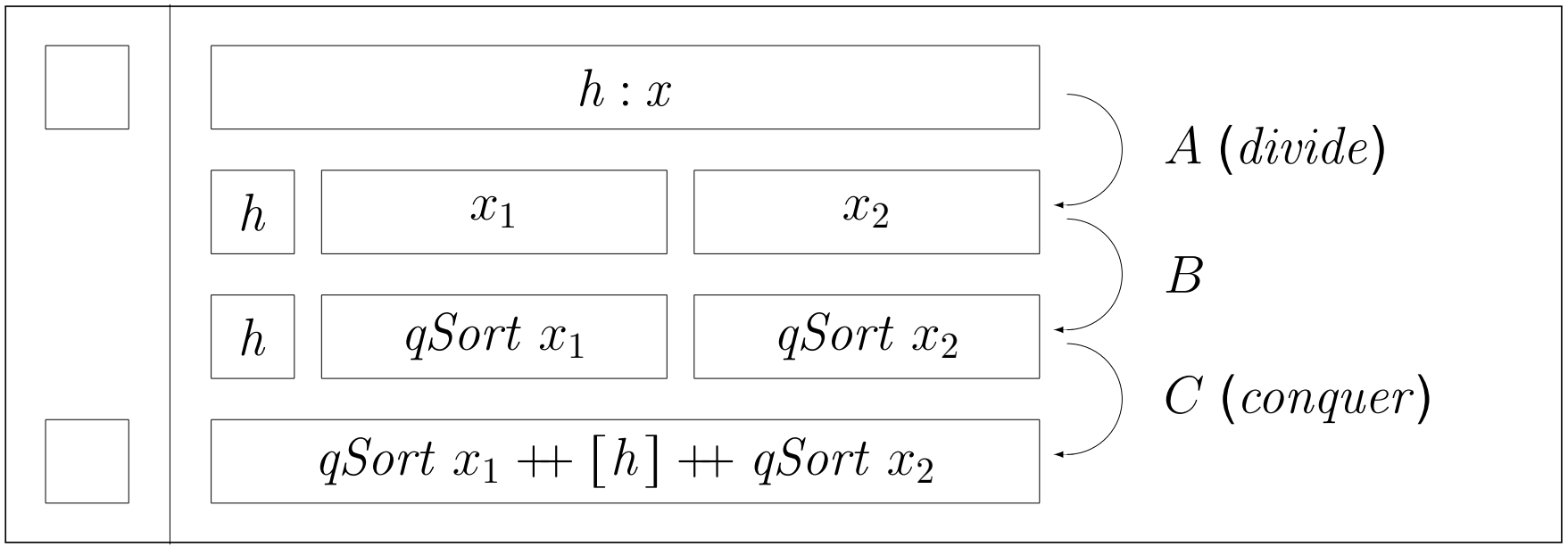

g
A
B (X, A)
[(g)]
B (id,[(g)])
out
T X
≅
B (X, T X)
in
(| f |)
B (id,(| f |))
C
B (X, C)
f
(| _ |) · [(_)]


QUICKSORT
a.hs
1 qSort :: Ord a => [a] -> [a]
2 qSort [] = []
3 qSort (h:x) = qSort x1 ++ [h] ++ qSort x2
4 where
5 x1 = [a | a <- x , a < h]
6 x2 = [a | a <- x , a >= h]
7

QUICKSORT

a.hs

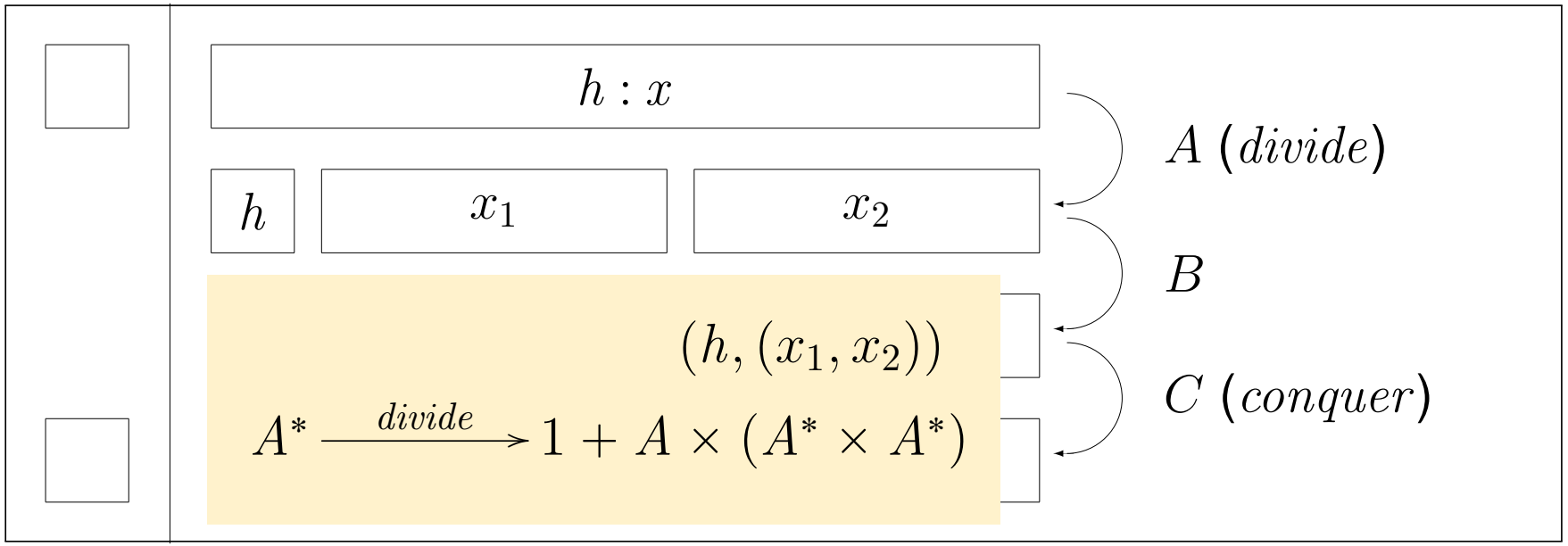
```haskell
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (h:x) = qSort x1 ++ [h] ++ qSort x2
    where
      x1 = [ a | a <- x , a < h ]
      x2 = [ a | a <- x , a >= h ]

```



QUICKSORT

a.hs

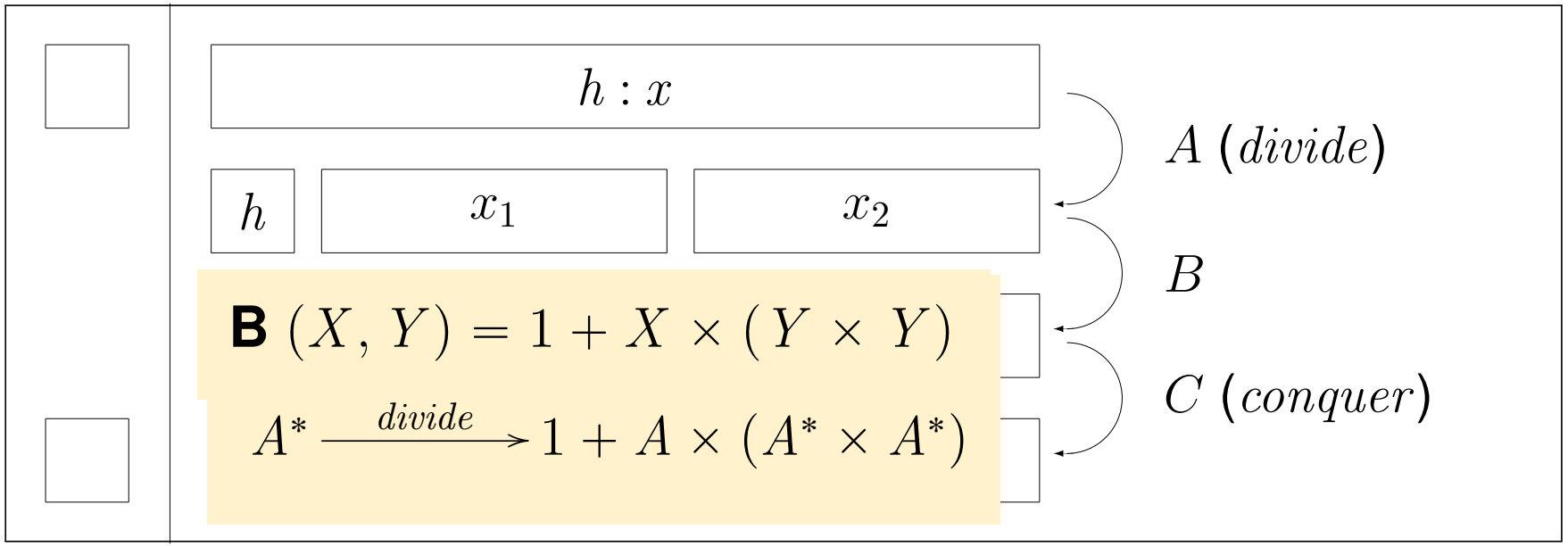
```haskell
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (h:x) = qSort x1 ++ [h] ++ qSort x2
    where
      x1 = [ a | a <- x , a < h ]
      x2 = [ a | a <- x , a >= h ]

```



QUICKSORT

a.hs

```haskell
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (h:x) = qSort x1 ++ [h] ++ qSort x2
    where
      x1 = [ a | a <- x , a < h ]
      x2 = [ a | a <- x , a >= h ]

```



QUICKSORT

$$\mathbf{B}\,(X, Y) = 1 + X \times (Y \times Y)$$

Description	T X	B (X, Y)	B (id, f)	B (f, id)
"Right" Lists	List X	$1 + X \times Y$	$id + id \times f$	$id + f \times id$
"Left" Lists	LList X	$1 + Y \times X$	$id + f \times id$	$id + id \times f$
Non-empty Lists	NList X	$X + X \times Y$	$id + id \times f$	$f + f \times id$
Binary Trees	BTree X	$1 + X \times Y^2$	$id + id \times f^2$	$id + f \times id$
"Leaf" Trees	LTree X	$X + Y^2$	$id + f^2$	$f + id$

QUICKSORT

a.hs

```haskell
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (h:x) = qSort x1 ++ [h] ++ qSort x2
    where
      x1 = [ a | a <- x , a < h ]
      x2 = [ a | a <- x , a >= h ]

```

$$\begin{cases} qSort \cdot \mathsf{nil} = \mathsf{nil} \\ qSort \cdot \mathsf{cons} = f_2 \cdot (id \times (qSort \times qSort)) \cdot g_2 \end{cases}$$

$$f_2\ (h, (y_1, y_2)) = y_1 +\!\!+ [h] +\!\!+ y_2$$

$$g_2\ (h, x) = (h, (x_1, x_2))$$
$$\textbf{where}$$
$$x_1 = [a \mid a \leftarrow x, a < h]$$
$$x_2 = [a \mid a \leftarrow x, a \geqslant h]$$

QUICKSORT

$$\mathbf{B}\ (X, Y) = 1 + X \times (Y \times Y)$$

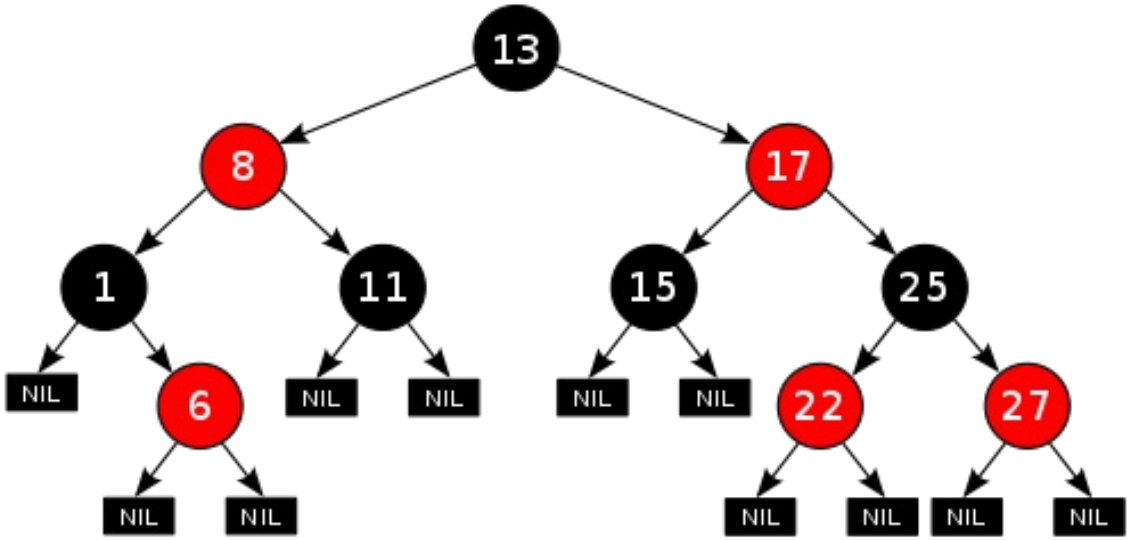
$$\begin{cases} qSort \cdot \mathsf{nil} = \mathsf{nil} \\ qSort \cdot \mathsf{cons} = f_2 \cdot (id \times (qSort \times qSort)) \cdot g_2 \end{cases}$$

$\equiv$ { fusão-+, absorção-+, eq-+ etc }

$$qSort \cdot \mathsf{in} = [\mathsf{nil}, f_2] \cdot (id + id \times qSort^2) \cdot (id + g_2)$$

$\equiv$ { isomorfismo in / out }

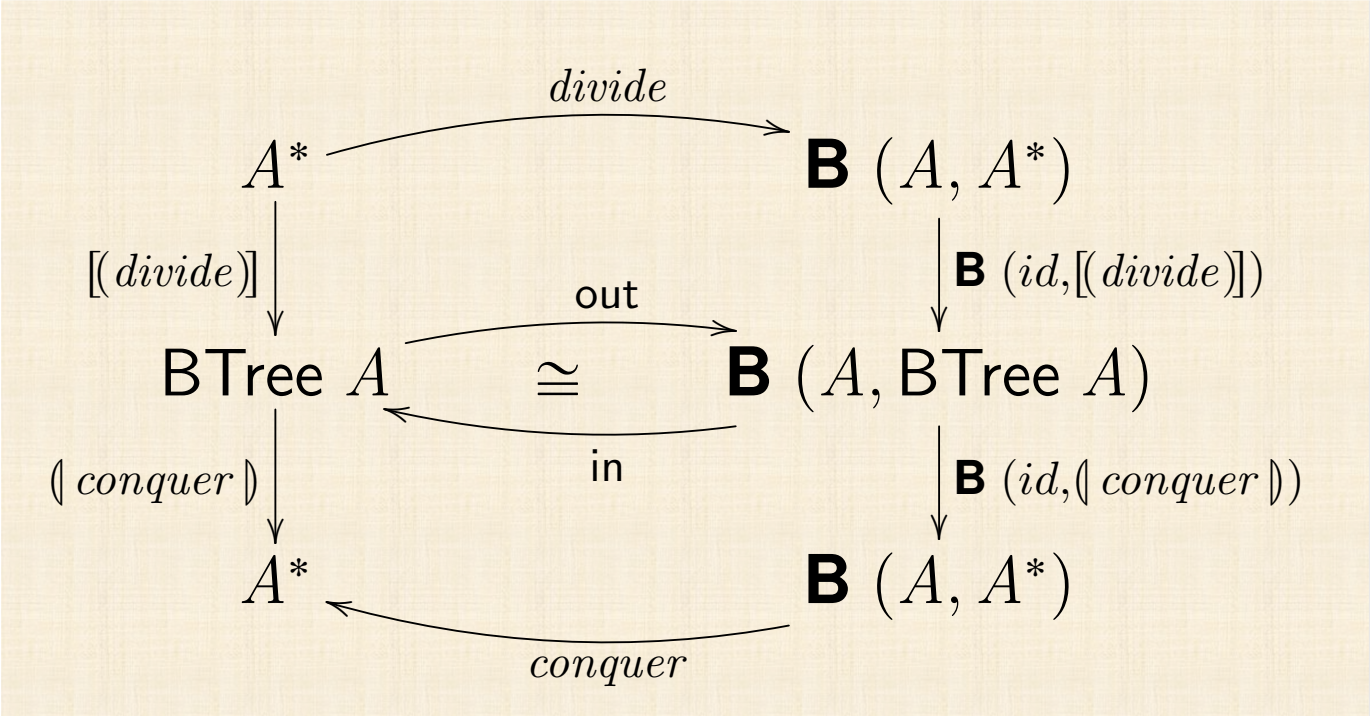
$$qSort = \underbrace{[\mathsf{nil}, f_2]}_{conquer} \cdot (\underbrace{id + id \times qSort^2}_{\mathbf{B}\ (id, qSort)} \cdot \underbrace{(id + g_2) \cdot \mathsf{out}}_{divide}$$




```
data BTree a = Empty | Node (a, (BTree a, BTree a))
```

QUICKSORT

```haskell
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (h:x) = qSort x1 ++ [h] ++ qSort x2
    where
      x1 = [ a | a <- x , a < h ]
      x2 = [ a | a <- x , a >= h ]
```

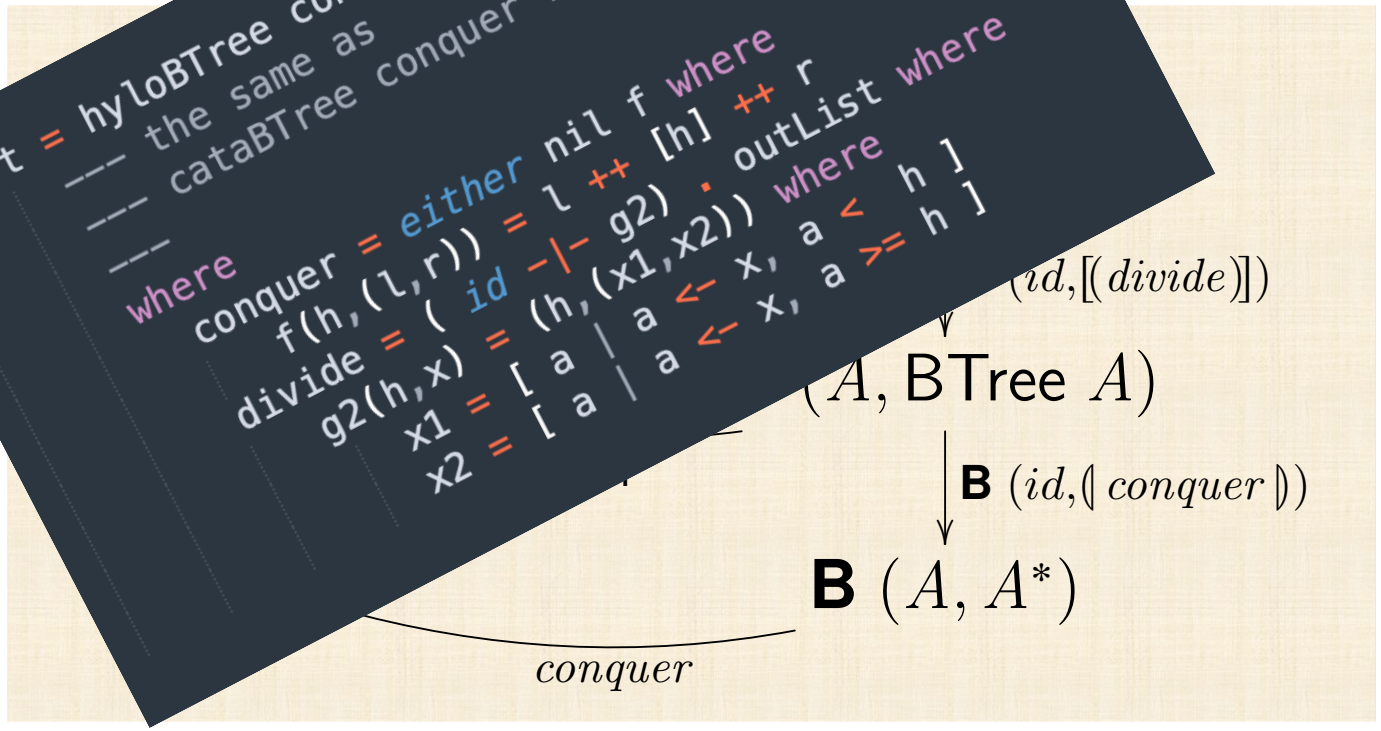


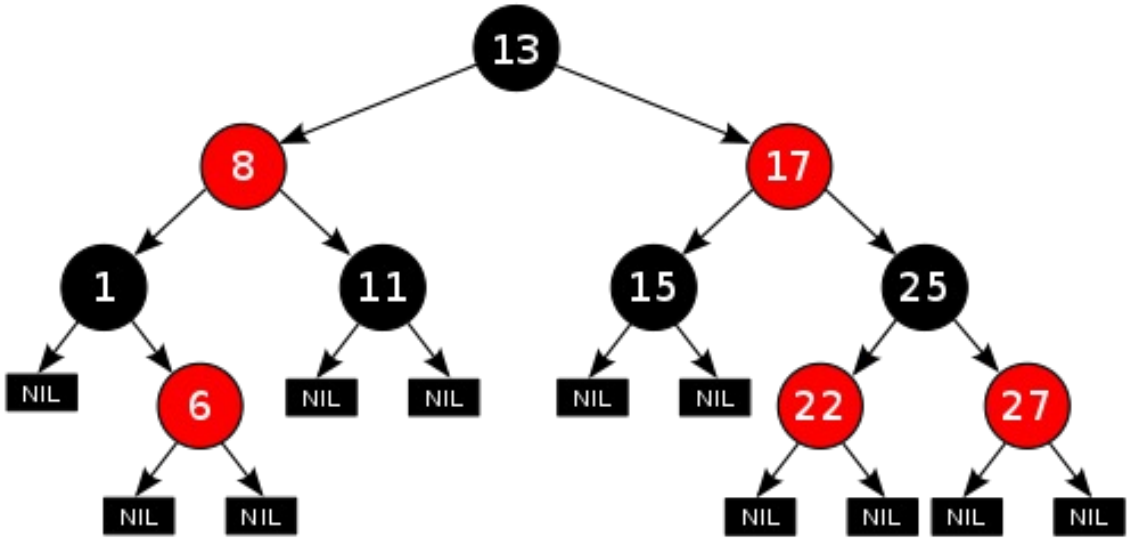
QUICKSORT

```haskell
qSort = hyloBTree conquer divide
--- the same as
--- cataBTree conquer . anaBTree divide
---
where
conquer = either nil f where
  f(h,(l,r)) = l ++ [h] ++ r
divide = ( id -|- g2) . outList where
g2(h,x) = (h,(x1,x2)) where
  x1 = [ a | a <- x, a <  h ]
  x2 = [ a | a <- x, a >= h ]
```

a.hs

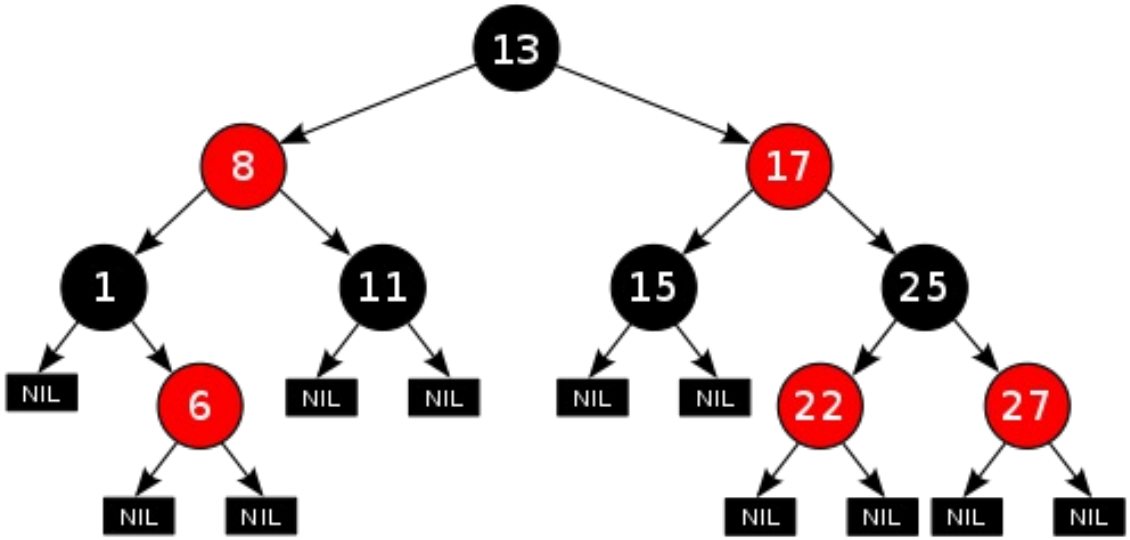
```haskell
qSort :: Ord a => [a] -> [a]
qSort [] = []
qSort (h:x) = qSort x1 ++ [h] ++ qSort x2
    where
      x1 = [ a | a <- x , a < h ]
      x2 = [ a | a <- x , a >= h ]
```



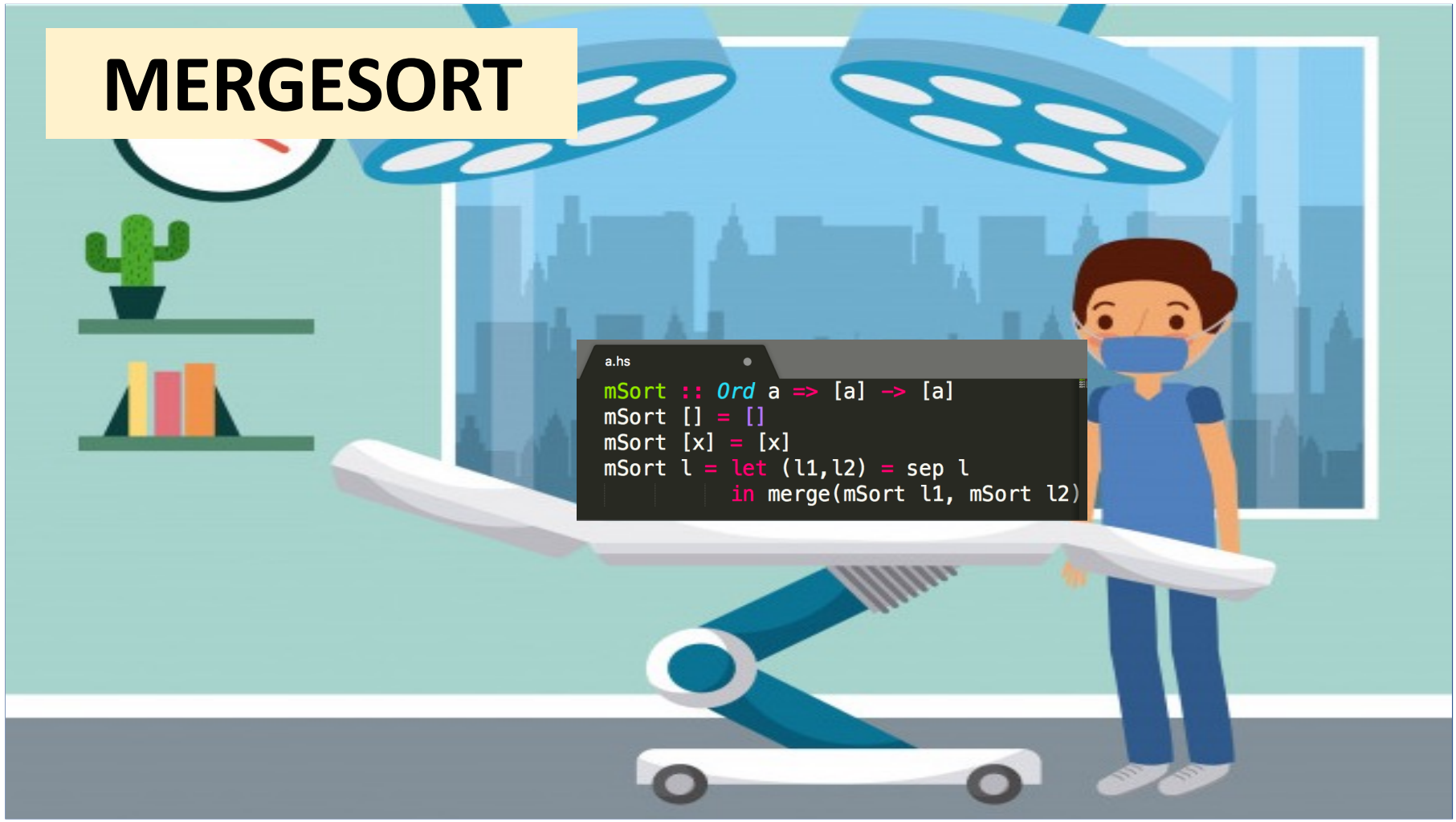



```
t = anaB divide [13,8,17,1,6,11,25,15,27,22]
```

```
data BTree a = Empty | Node (a, (BTree a, BTree a))
```



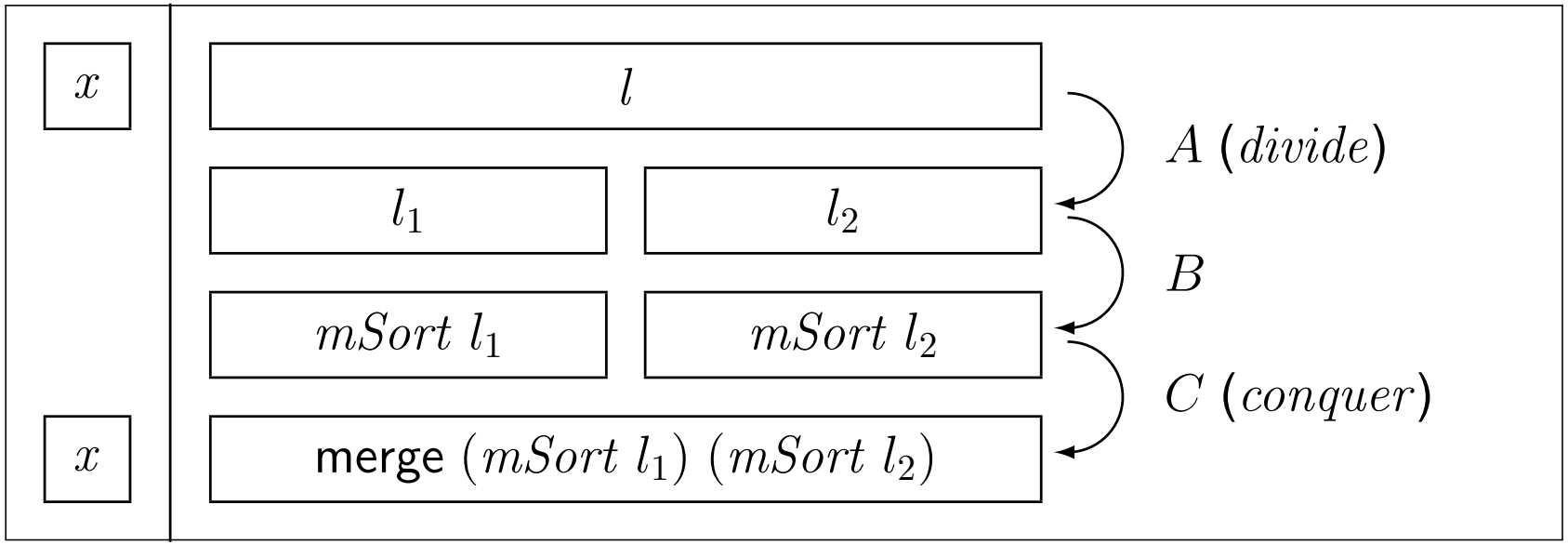

```
*Cp> anaB divide [13,8,17,1,6,11,25,15,27,22]
Node (13,(Node (8,(Node (1,(Empty,Node (6,(Empty,Empty)))),Node
(11,(Empty,Empty)))),Node (17,(Node (15,(Empty,Empty)),Node (25,
(Node (22,(Empty,Empty)),Node (27,(Empty,Empty)))))))
```


MERGESORT
a.hs
mSort :: Ord a => [a] -> [a]
mSort [] = []
mSort [x] = [x]
mSort l = let (l1,l2) = sep l
in merge(mSort l1, mSort l2)

MERGESORT

a.hs

```
mSort :: Ord a => [a] -> [a]
mSort [] = []
mSort [x] = [x]
mSort l = let (l1,l2) = sep l
          in merge(mSort l1, mSort l2)
```

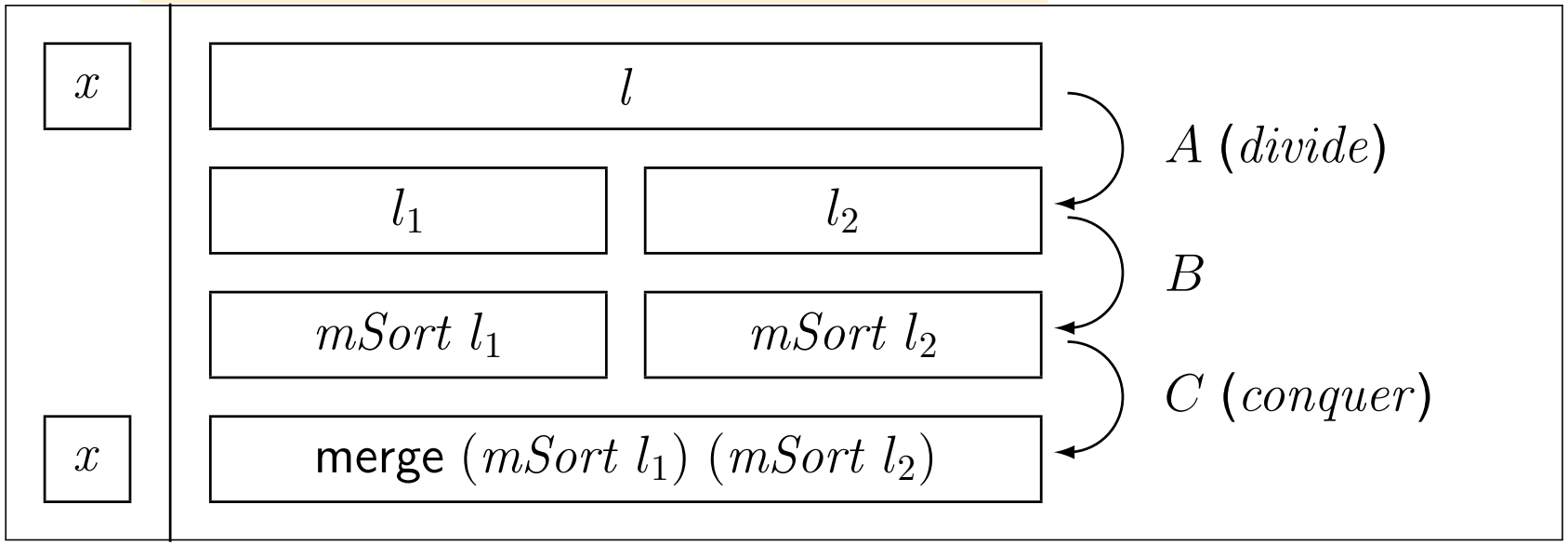


MERGESORT

a.hs

```
mSort :: Ord a => [a] -> [a]


l1,l2) = sep l
rge(mSort l1, mSort l2)
```

$$divide : A^* \rightarrow A + (A^* \times A^*)$$

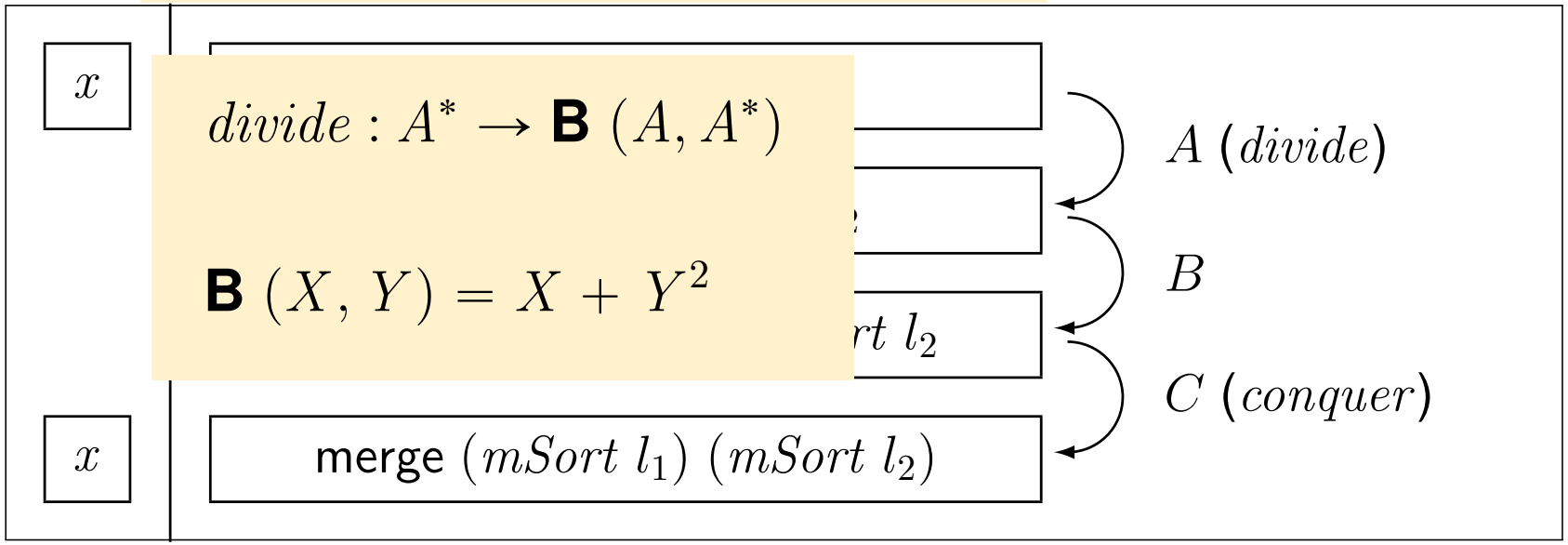


MERGESORT

a.hs

```
mSort :: Ord a => [a] -> [a]


l1,l2) = sep l
rge(mSort l1, mSort l2)
```

$$divide : A^* \rightarrow A + (A^* \times A^*)$$



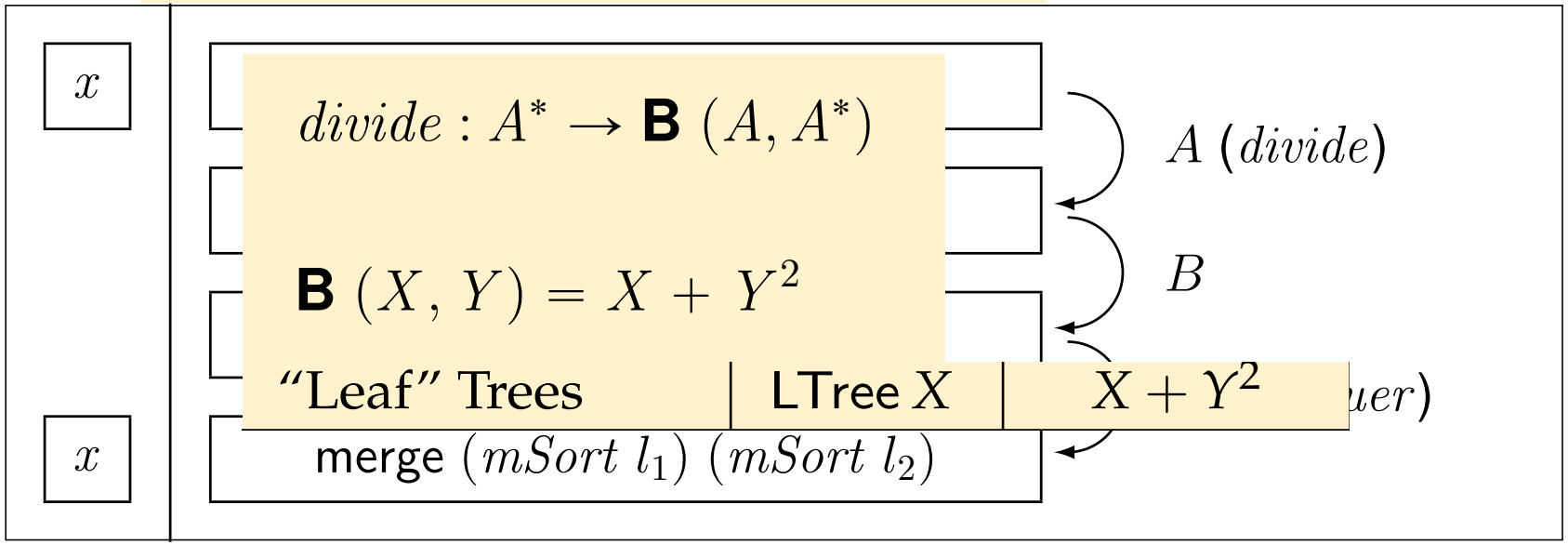
$$divide : A^* \rightarrow \mathbf{B}\ (A, A^*)$$

$$\mathbf{B}\ (X, Y) = X + Y^2$$

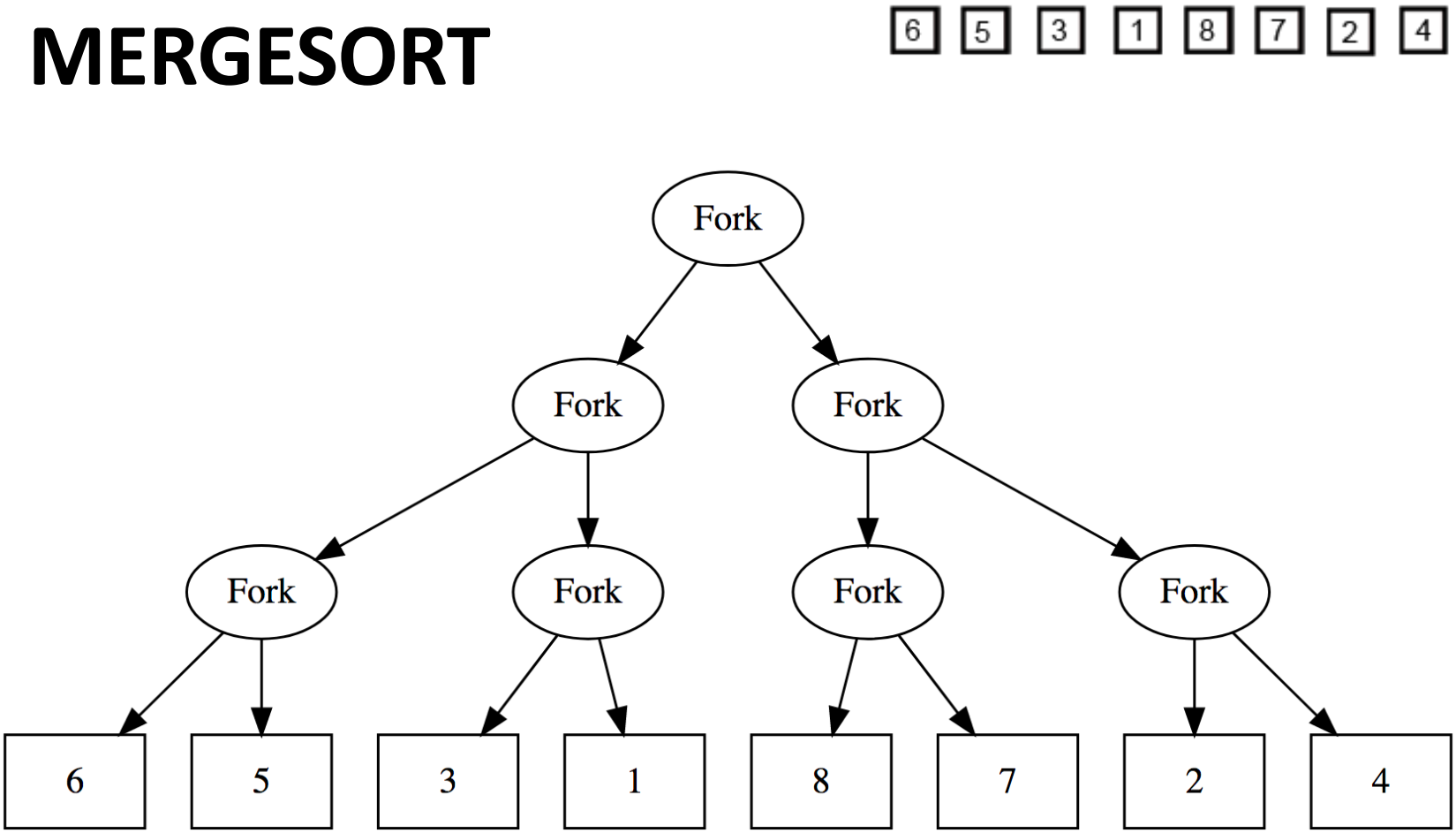
MERGESORT

```
a.hs
mSort :: Ord a => [a] -> [a]


l1,l2) = sep l
rge(mSort l1, mSort l2)
```

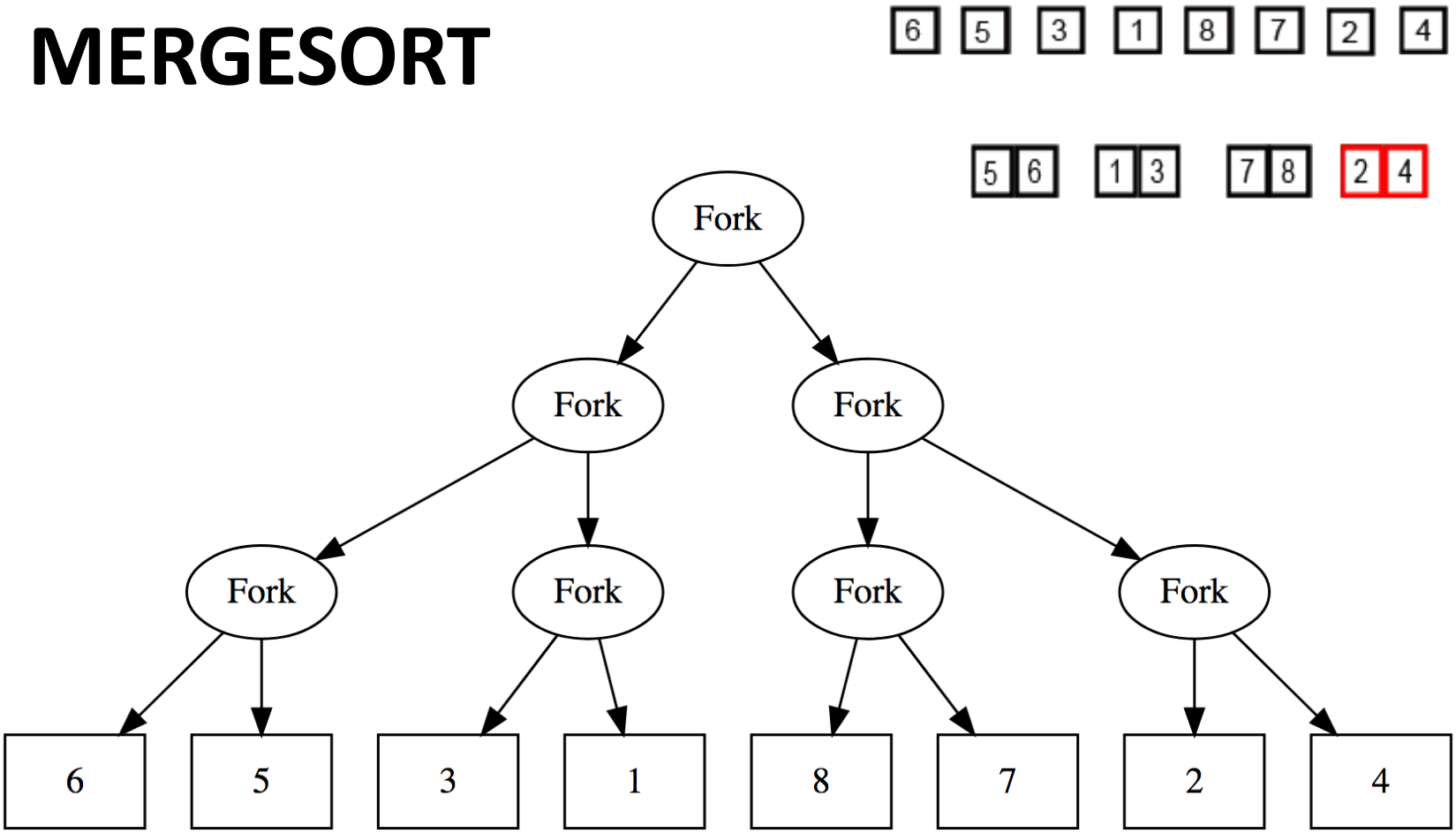
$$divide : A^* \to A + (A^* \times A^*)$$



MERGESORT



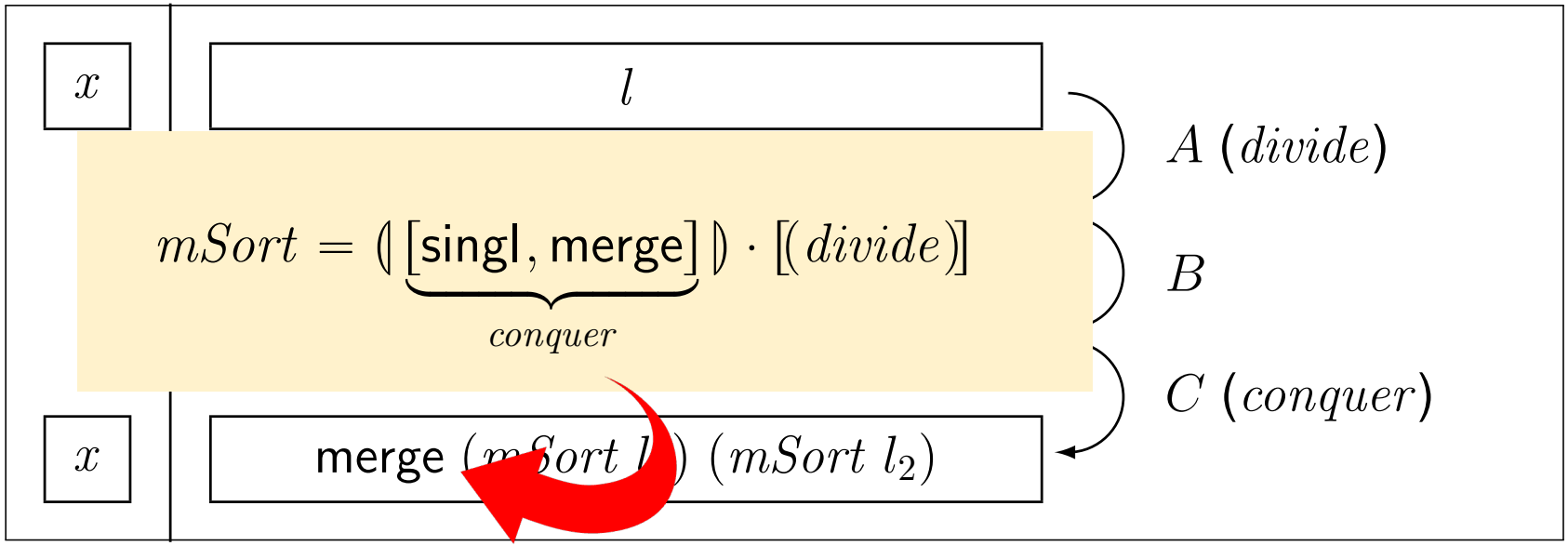
MERGESORT


6 5 3 1 8 7 2 4
5 6 1 3 7 8 2 4
Fork
Fork
Fork
Fork
Fork
Fork
Fork
6
5
3
1
8
7
2
4

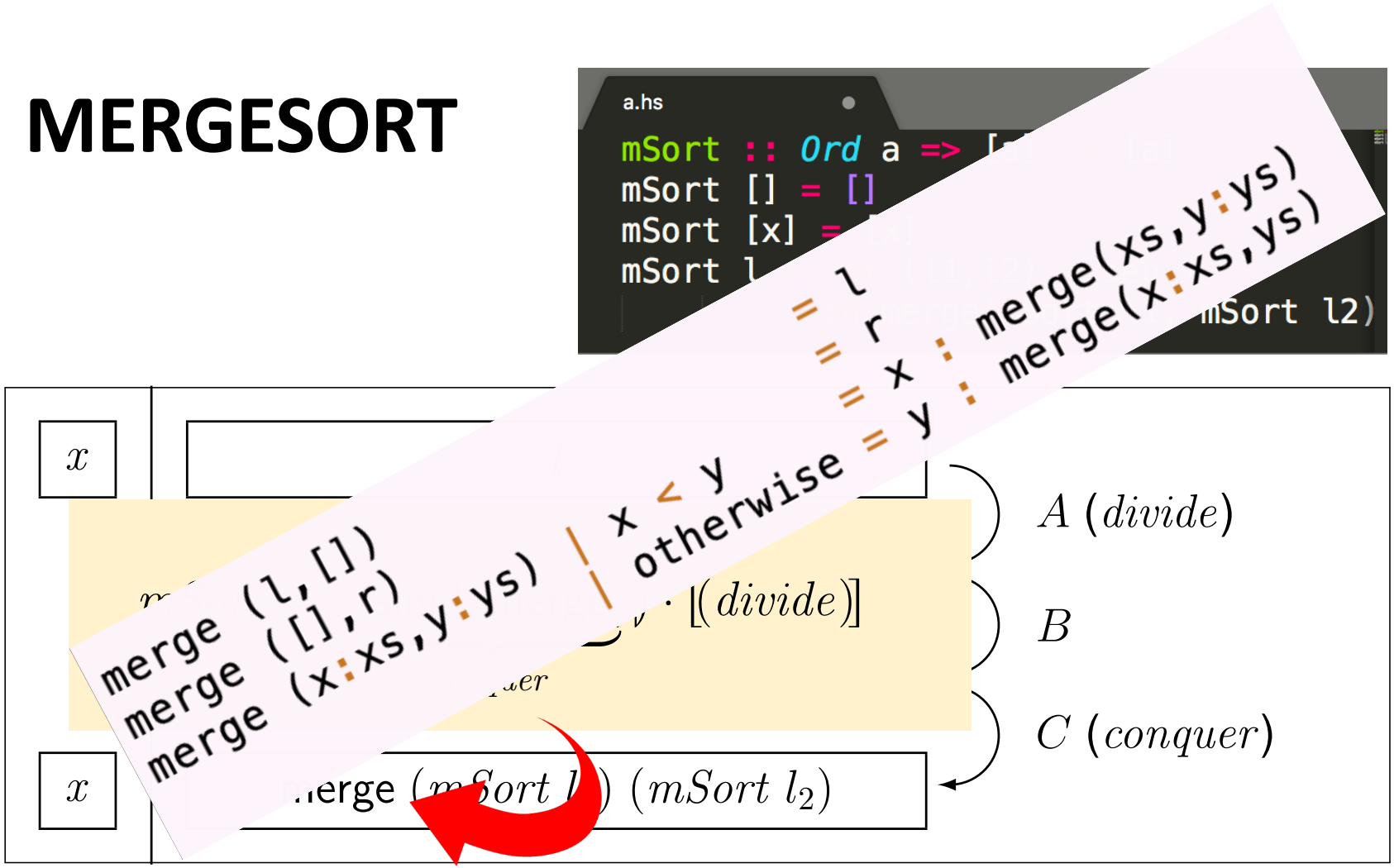
MERGESORT

a.hs

```haskell
mSort :: Ord a => [a] -> [a]
mSort [] = []
mSort [x] = [x]
mSort l = let (l1,l2) = sep l
          in merge(mSort l1, mSort l2)
```



MERGESORT

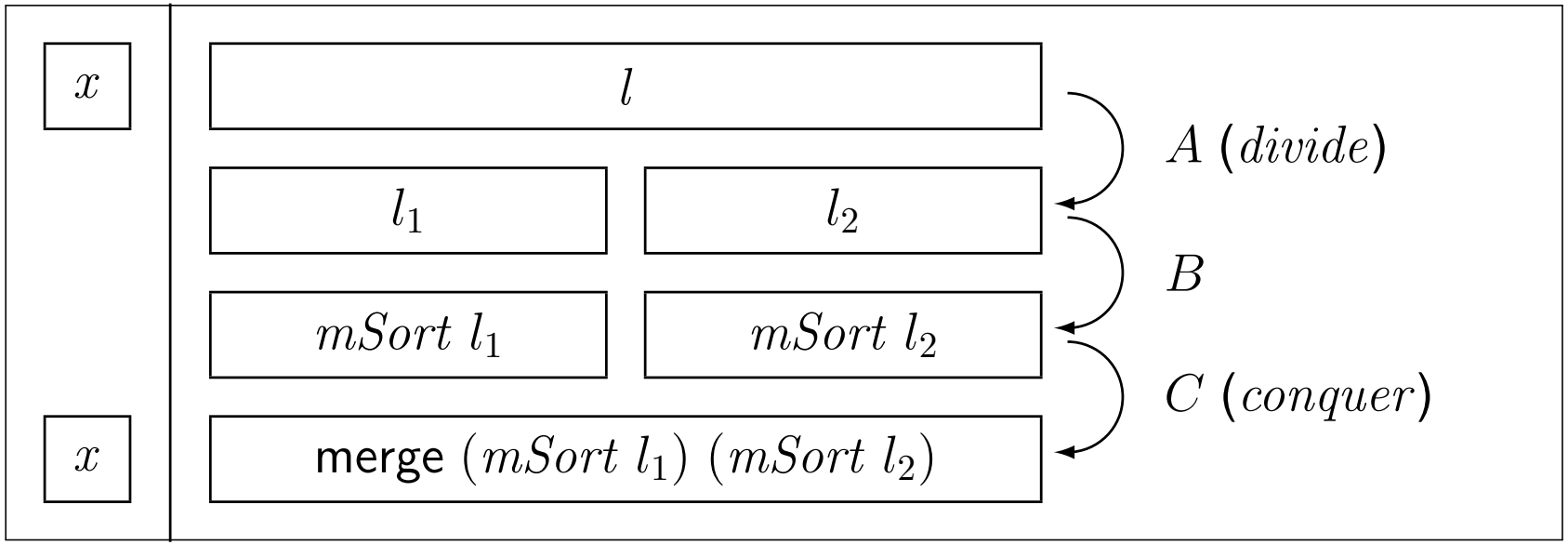

a.hs
mSort :: Ord a => [
mSort [] = []
mSort [x] =
mSort l
= l
= r
= x : merge(xs,y:ys)
= y : merge(x:xs,ys)
mSort l2)
merge (l,[])
merge ([],r)
merge (x:xs,y:ys) | x < y
| otherwise
x
x
A (*divide*)
B
C (*conquer*)
$(mSort\ l_2)$

Cálculo de Programas
T11

MERGESORT

a.hs

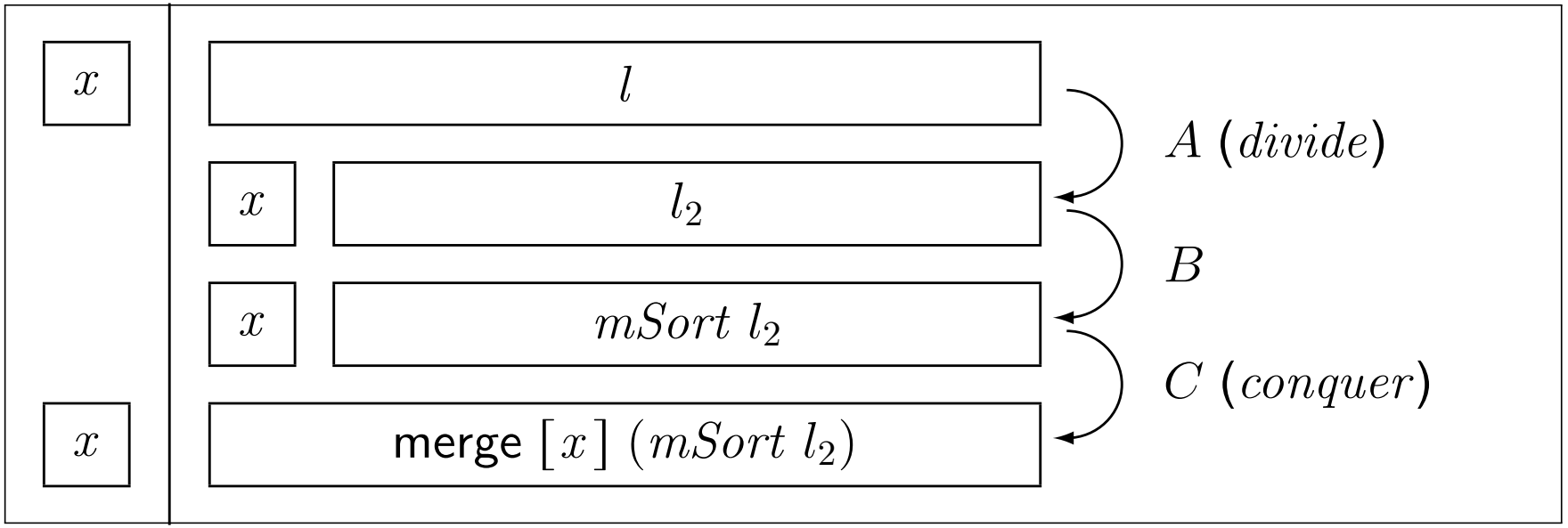
```
mSort :: Ord a => [a] -> [a]
mSort [] = []
mSort [x] = [x]
mSort l = let (l1,l2) = sep l
          in merge(mSort l1, mSort l2)
```



MERGESORT

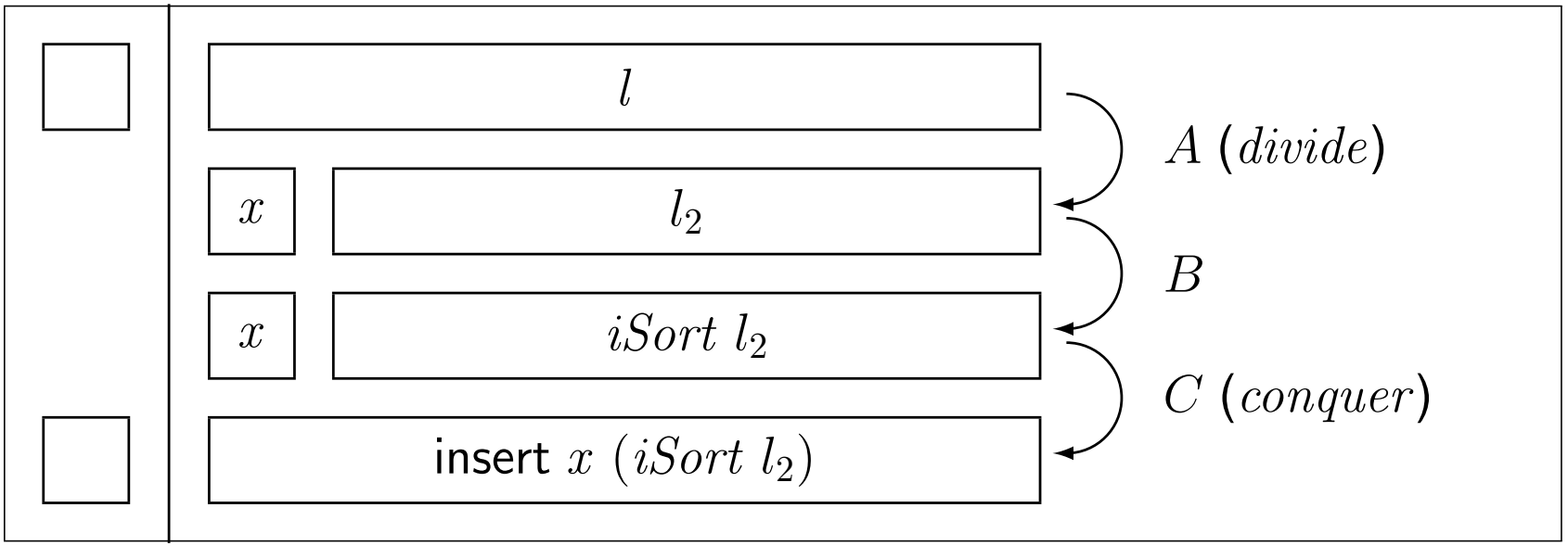
Particular case:

```
merge :: Ord a => ([a], [a]) -> [a]
merge ([x],[])                  = [x]
merge ([],r)                    = r
merge ([x],y:ys) | x < y        = x : merge([],y:ys)
                 ) | otherwise = y : merge(x:[],ys
```



INSERTION SORT

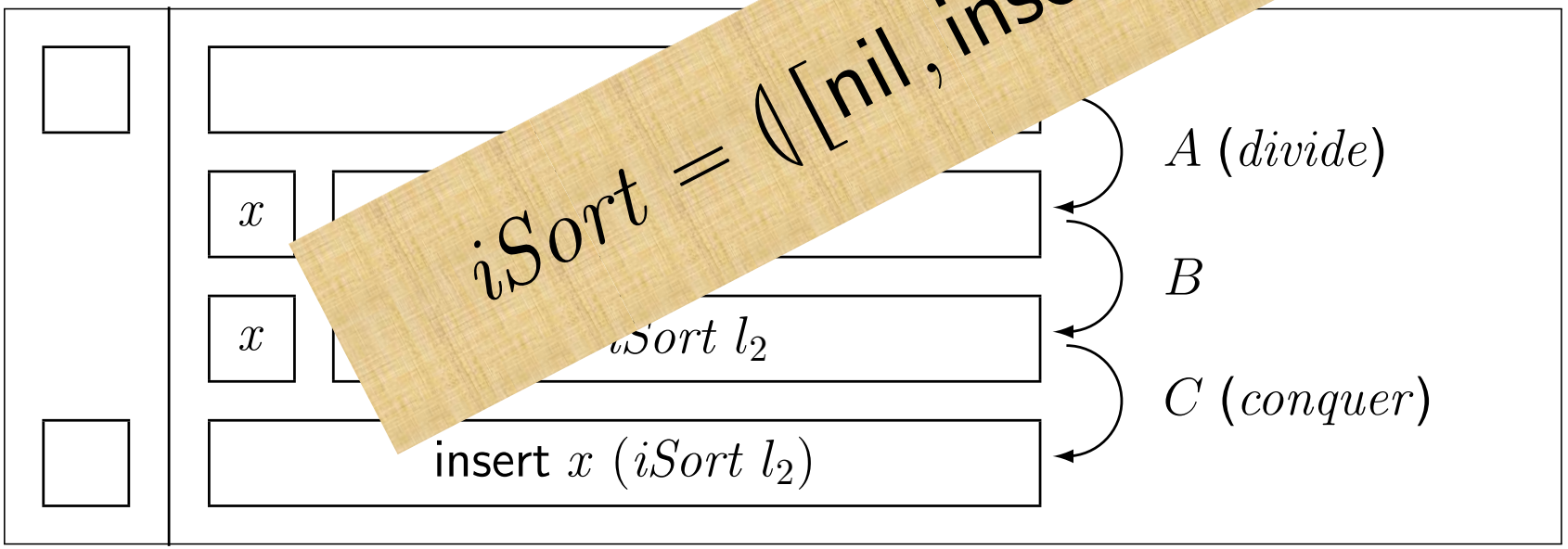
```haskell
insert :: Ord t => t -> [t] -> [t]
insert x []                 = [x]
insert x (y:ys) | x < y     = x:y:ys
                | otherwise = y:insert x ys
```



INSERTION SORT

```
insert :: Ord t => t -> [t] -> [
insert x []
insert x (y:ys) | x < y
                | ot
                                  x ys
```

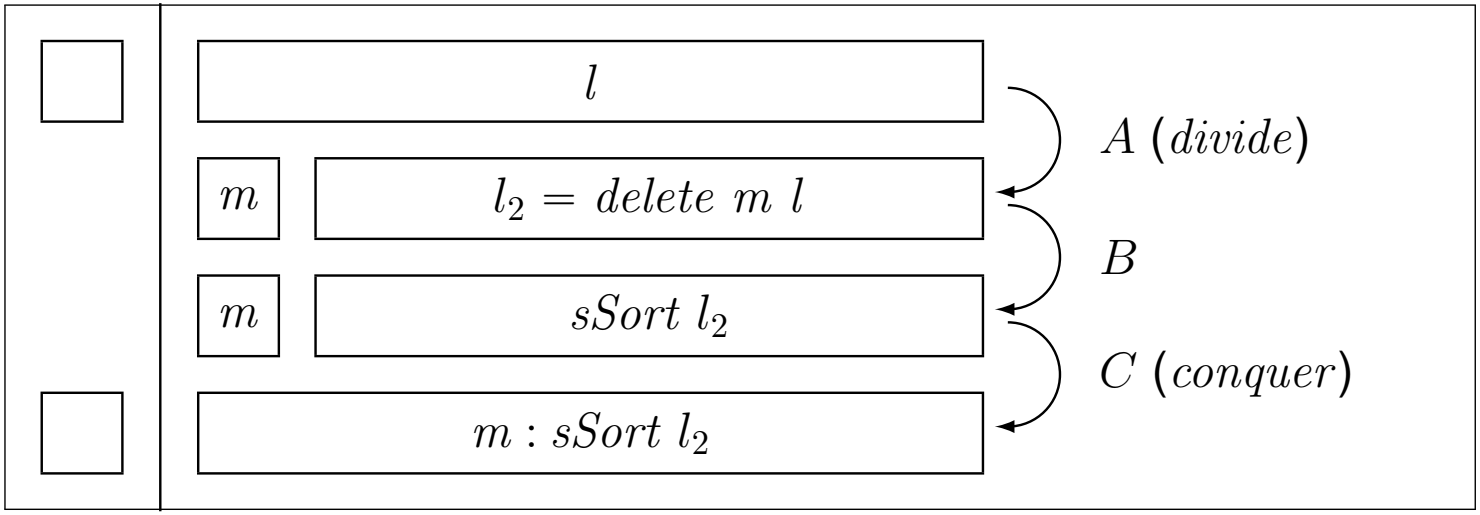
$$iSort = (\!|[nil, \widehat{insert}]|\!)$$



SELECTION SORT

```haskell
sSort :: Ord a => [a] -> [a]
sSort = anaL divide where
   divide [] = i1()
   divide (xs) = let m = minimum xs
                 in  i2(m, delete m xs)
```

Selection sort ($m = minimum\ l$):



SELECTION SORT

```haskell
1
2  sSort :: Ord a => [a] -> [a]
3  sSort = anaL divide where
4     divide [] = i1()
5     divide (xs) = let m = minimum xs
6                   in  i2(m, delete m xs)
```

Selection sort ($m = minimum\ l$):

ANA, CATA & HYLO

$$C \xleftarrow{(\!|f|\!)} \mathsf{T} \xleftarrow{[\!(g)\!]} A$$

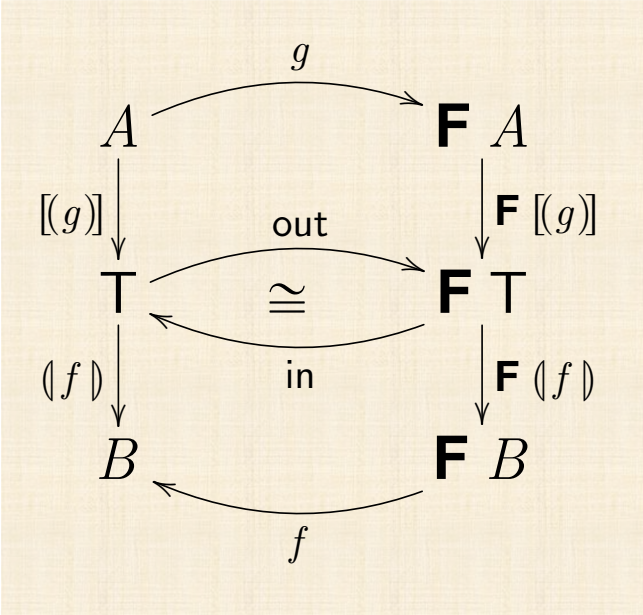
$$[\![f, g]\!] = (\!|f|\!) \cdot [\!(g)\!]$$

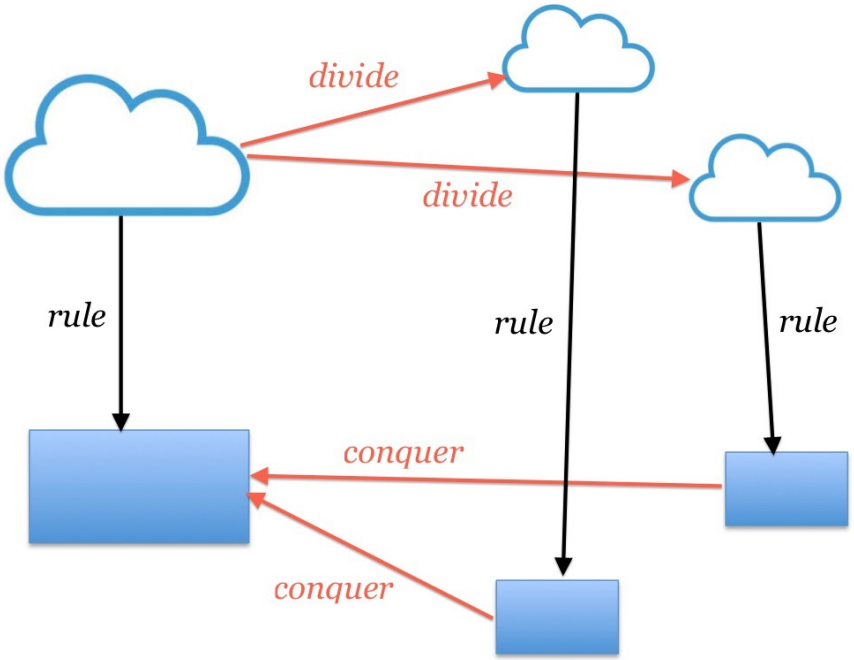
ALGORITHM CLASSIFICATION

	Singleton	Equal-size
Easy Split/Hard Join	Insertion Sort	Merge Sort
Hard Split/Easy Join	Selection Sort	Quick Sort

NB:
'Split' = *divide*
'Join' = *conquer*

More on ‘DIVIDE & CONQUER’



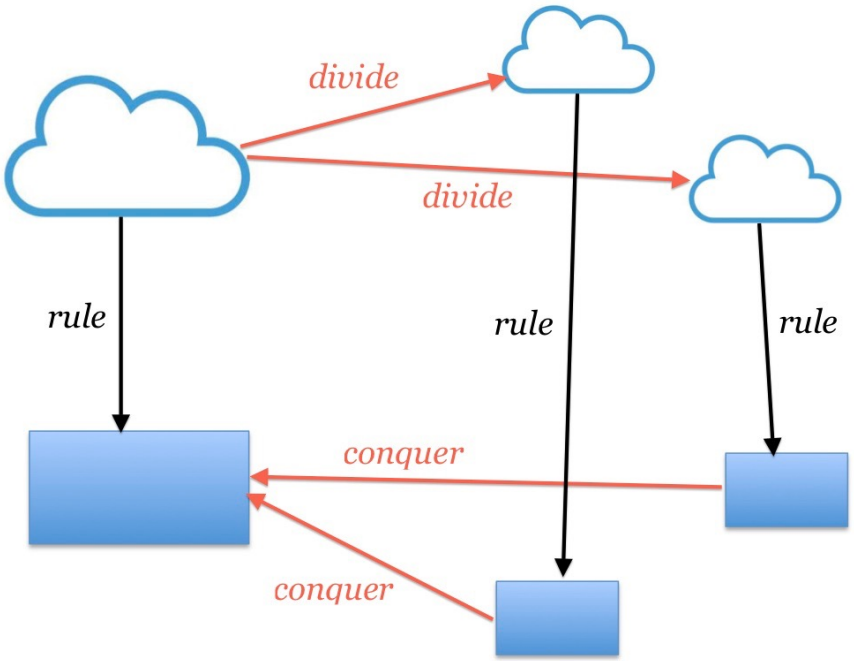


More on ‘DIVIDE & CONQUER’

$$C \xleftarrow{(\!|\, conquer \,|\!)} \mathsf{T} \xleftarrow{[\!(divide)\!]} A$$

More on ‘DIVIDE & CONQUER’

QUESTION:
In algorithm analysis, how do we find divide and conquer?



Example (FIBONACCI)

$$
\begin{aligned}
&\mathit{fib}\ 0 = 1 \\
&\mathit{fib}\ 1 = 1 \\
&\mathit{fib}\ (n+2) = \mathit{fib}\ (n+1) + \mathit{fib}\ n
\end{aligned}
$$

$$\mathsf{in} = [\underline{0}, \mathsf{succ}]$$

FIBONACCI

(practical rule)

$$
\begin{aligned}
&\mathit{fib}\ 0 = 1\\
&\mathit{fib}\ 1 = 1\\
&\mathit{fib}\ (n+2) = \mathit{fib}\ (n+1) + \mathit{fib}\ n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{divide}\ 0 = 1\\
&\mathit{divide}\ 1 = 1\\
&\mathit{divide}\ (n+2) = \ldots (n+1) \ldots n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{divide}\ 0 = i_1\ 1\\
&\mathit{divide}\ 1 = i_1\ 1\\
&\mathit{divide}\ (n+2) = i_2\ (n+1, n)
\end{aligned}
$$

FIBONACCI

$$
\begin{aligned}
&\mathit{fib}\ 0 = 1 \\
&\mathit{fib}\ 1 = 1 \\
&\mathit{fib}\ (n+2) = \mathit{fib}\ (n+1) + \mathit{fib}\ n
\end{aligned}
$$

$$\mathit{divide} : \mathbb{N}_0 \to \mathbb{N}_0 + {\mathbb{N}_0}^2$$

$$
\begin{aligned}
&\mathit{divide}\ 0 = i_1\ 1 \\
&\mathit{divide}\ 1 = i_1\ 1 \\
&\mathit{divide}\ (n+2) = i_2\ (n+1, n)
\end{aligned}
$$

FIBONACCI

$$fib\ 0 = 1$$
$$fib\ 1 = 1$$
$$fib\ (n+2) = fib\ (n+1) + fib\ n$$

$$divide : \mathbb{N}_0 \to \mathbb{N}_0 + \mathbb{N}_0{}^2$$

$$divide\ (\quad$$
$$divide\quad$$
$$divide\ (n+2) = \imath_2\ (n+1, n)$$

$$\mathbf{B}\ (X, Y) = X + Y^2$$

FIBONACCI

$$fib\ (n+1) + fib\ n$$

$$\mathsf{T} = \mathsf{LTree}\ \mathbb{N}_0$$

$$divide : \mathbb{N}_0 + \mathbb{N}_0{}^2$$

$$\mathbf{B}\ (X, Y) = X + Y^2$$

$$divide\ (n+2) = \imath_2\ (n+1, n)$$

FIBONACCI

$$
\begin{aligned}
&\mathit{fib}\ 0 = 1\\
&\mathit{fib}\ 1 = 1\\
&\mathit{fib}\ (n+2) = \mathit{fib}\ (n+1) + \mathit{fib}\ n
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{divide}\ 0 = i_1\ 1\\
&\mathit{divide}\ 1 = i_1\ 1\\
&\mathit{divide}\ (n+2) = i_2\ (n+1, n)
\end{aligned}
$$

$$\mathsf{T} = \mathsf{LTree}\ \mathbb{N}_0$$

$$
\begin{aligned}
&\mathit{conquer} : \mathbb{N}_0 + {\mathbb{N}_0}^2 \to \mathbb{N}_0\\
&\mathit{conquer} = [\mathit{id}, \mathsf{add}]
\end{aligned}
$$

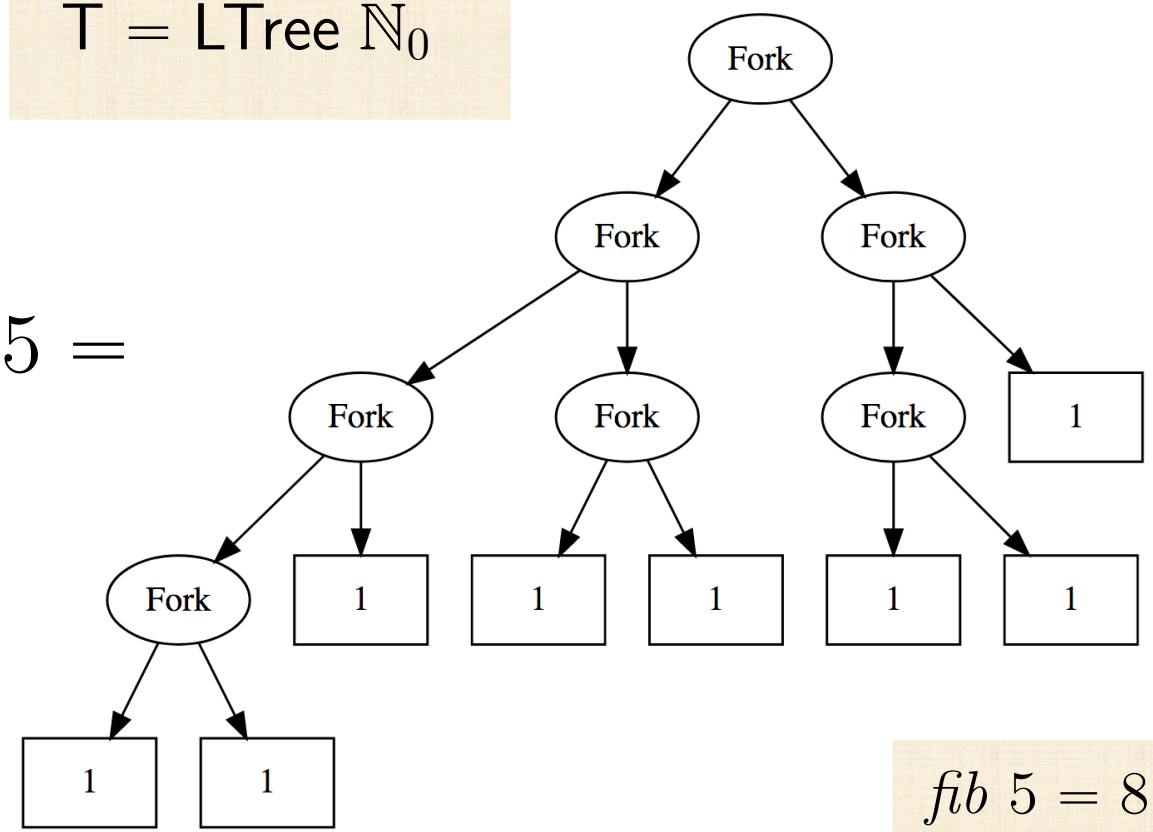
FIBONACCI

```haskell
fib :: (Integral c) => c -> c
fib = hyloL conquer divide
    where
        divide 0 = i1 1
        divide 1 = i1 1
        divide(n+2) = i2(n+1,n)
        conquer = either id add
```

FIBONACCI

$$\mathsf{T} = \mathsf{LTree}\ \mathbb{N}_0$$

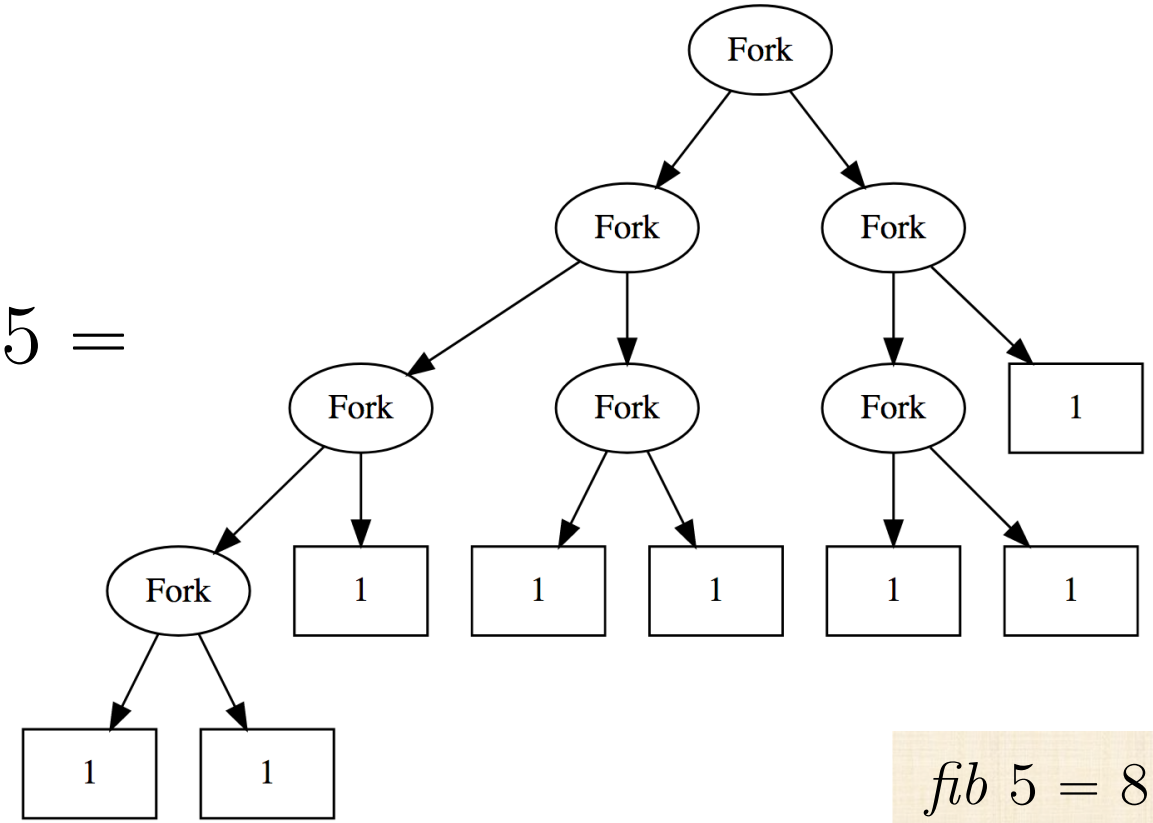
$$[\![(divide)]\!]\ 5 =$$



$$fib\ 5 = 8$$

FIBONACCI

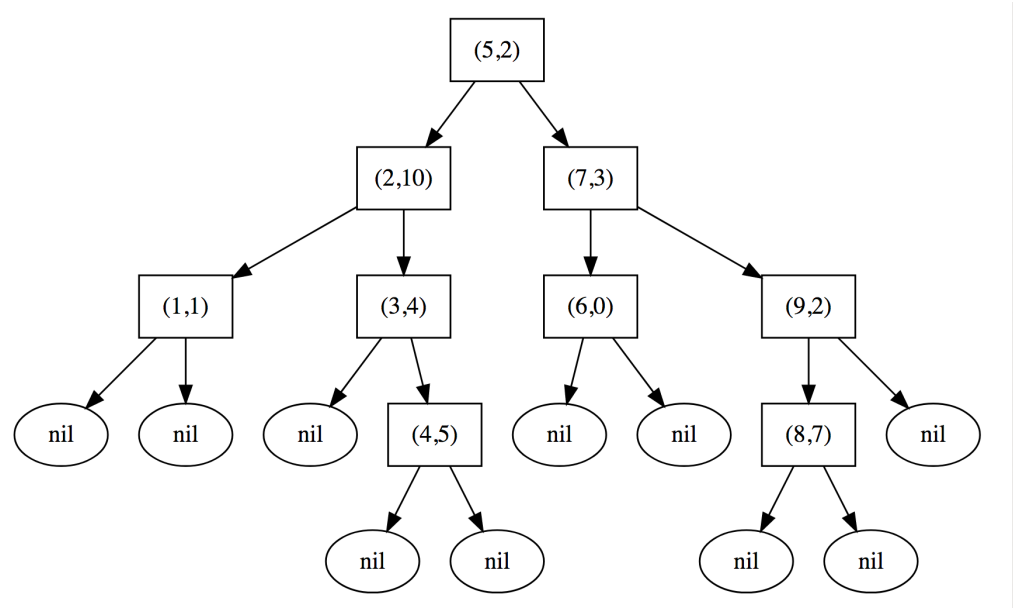
$$[\![(divide)]\!]\ 5 =$$



$$fib\ 5 = 8$$

Pesquisa binária

```haskell
lookBTree :: Ord a => a -> BTree (a,b) -> Maybe b
lookBTree a Empty = Nothing
lookBTree a (Node((a',b'),(l,r)))
    | a == a' = Just b'
    | a < a' =  lookBTree a l
    | a > a' =  lookBTree a r
```



Pesquisa binária

$$
\begin{array}{l}
try\ a\ ((a',b'),(l,r)) \\
\quad \mid a = a' = i_1\ (i_2\ b') \\
\quad \mid a < a' = i_2\ l \\
\quad \mid a > a' = i_2\ r
\end{array}
$$

```
lookBTree :: Ord a => a -> BTree (a,b) -> Maybe b
lookBTree a Empty = Nothing
lookBTree a (Node((a',b'),(l,r)))
    | a == a' = Just b'
    | a < a' =  lookBTree a l
    | a > a' =  lookBTree a r
```

$$
\begin{array}{ccccc}
\mathsf{T}\ (A \times B) & \xrightarrow{\text{out}} & 1 + (A \times B) \times (\mathsf{T}\ (A \times B))^2 & \xrightarrow{[i_1 \cdot i_1, try\ a]} & (1 + B) + \mathsf{T}\ (A \times B) \\
\downarrow{\scriptstyle look\ a} & & & & \downarrow{\scriptstyle id + look\ a} \\
1 + B & \xleftarrow{[id,id]} & & & (1 + B) + (1 + B)
\end{array}
$$

Pesquisa binária

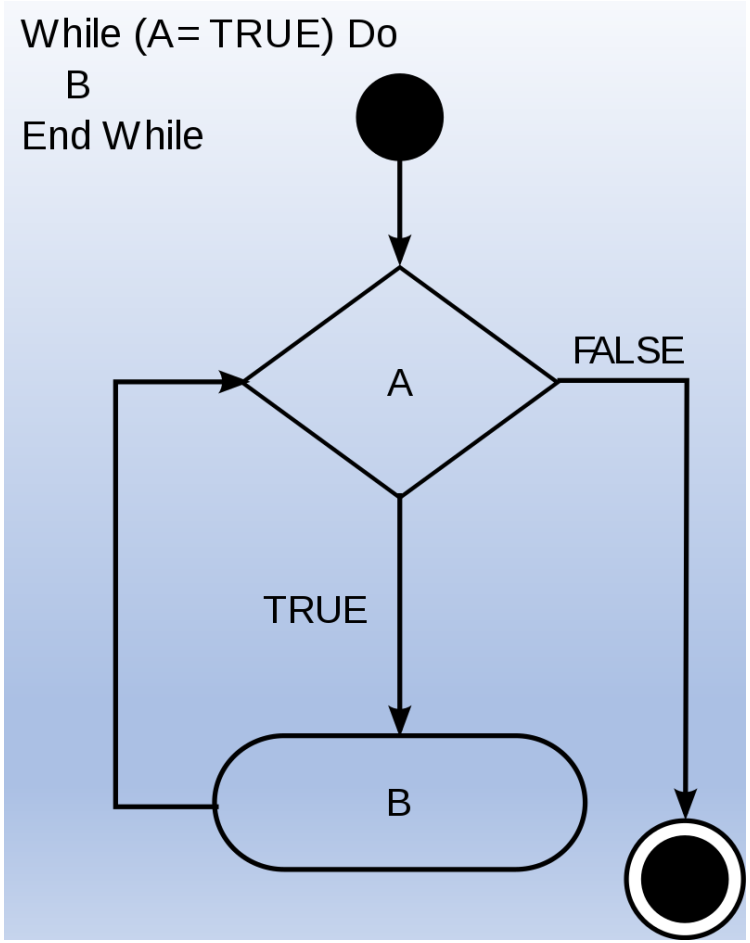
$$
\begin{aligned}
&try\ a\ ((a', b'), (l, r)) \\
&\quad \mid a = a' = i_1\ (i_2\ b') \\
&\quad \mid a < a' = i_2\ l \\
&\quad \mid a > a' = i_2\ r
\end{aligned}
$$

```
lookBTree :: Ord a =>                    -> Maybe b
lookBTree a Empty
lookBTree a (No
    | a == a'
    | a <
    | a
```

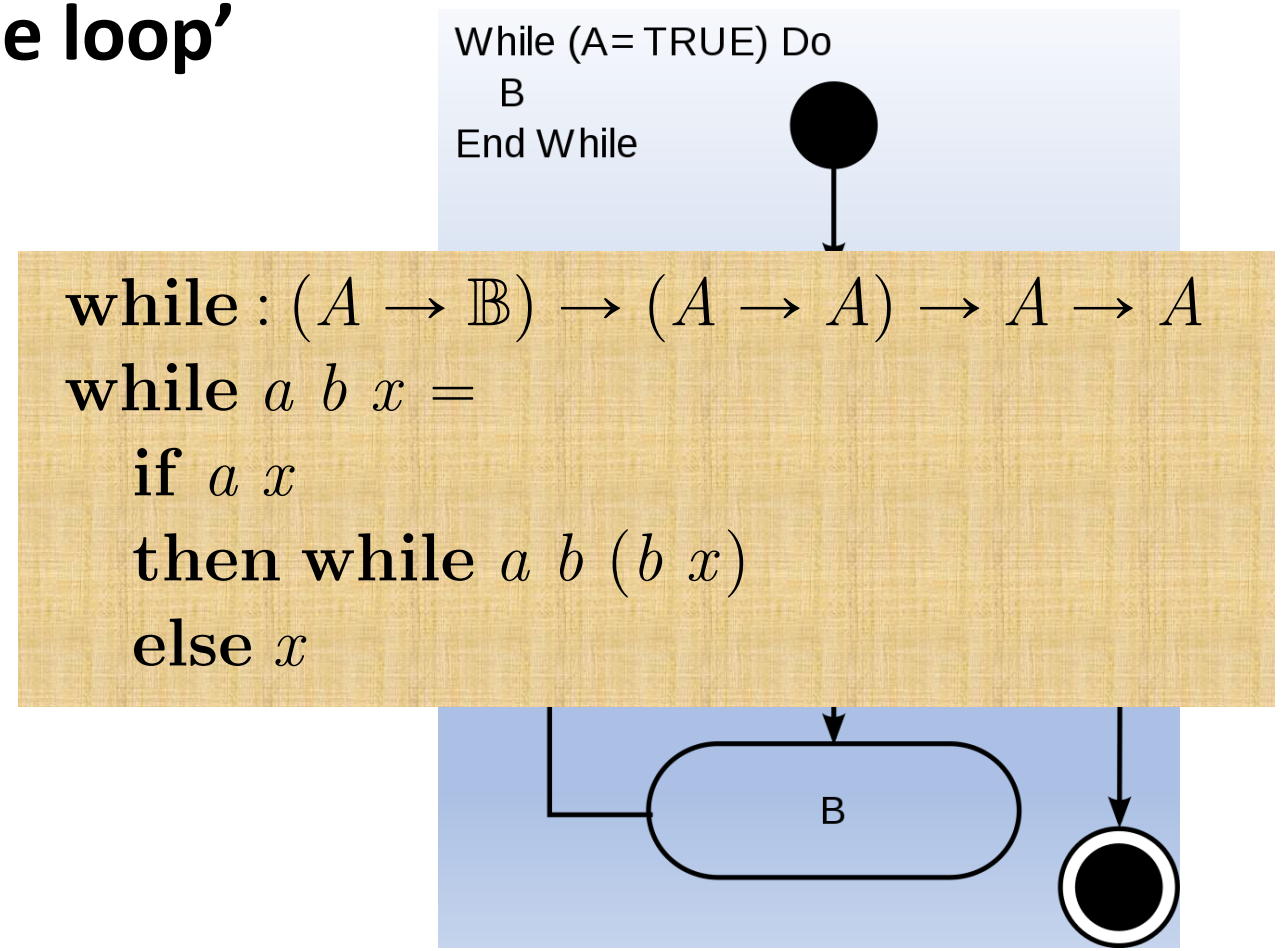
```
... while not(input == Empty)
    do { try a } ...
```

$$
\begin{array}{ccc}
\mathsf{T}\,(A \times B) & & (\mathsf{T}\,(A \times B))^2 \xrightarrow{[i_1 \cdot i_1,\, try\ a]} (1 + B) + \mathsf{T}\,(A \times B) \\
{\scriptstyle look\ a}\downarrow & & \downarrow{\scriptstyle id + look\ a} \\
1 + & \xrightarrow{\qquad [id, id] \qquad} & (1 + B) + (1 + B)
\end{array}
$$

‘While loop’

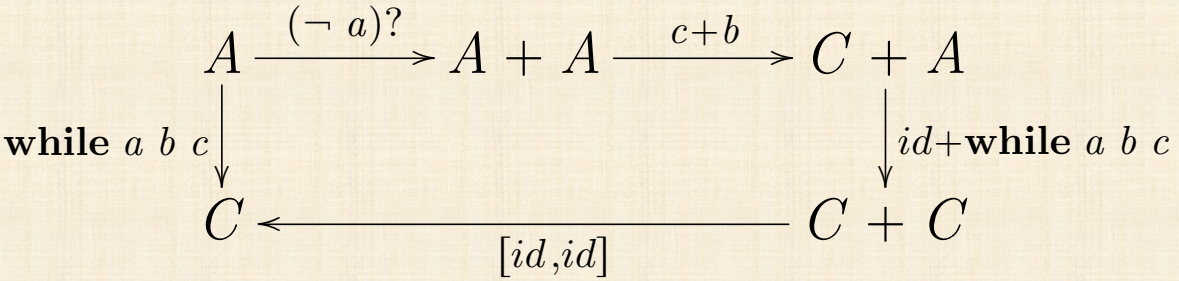


‘While loop’

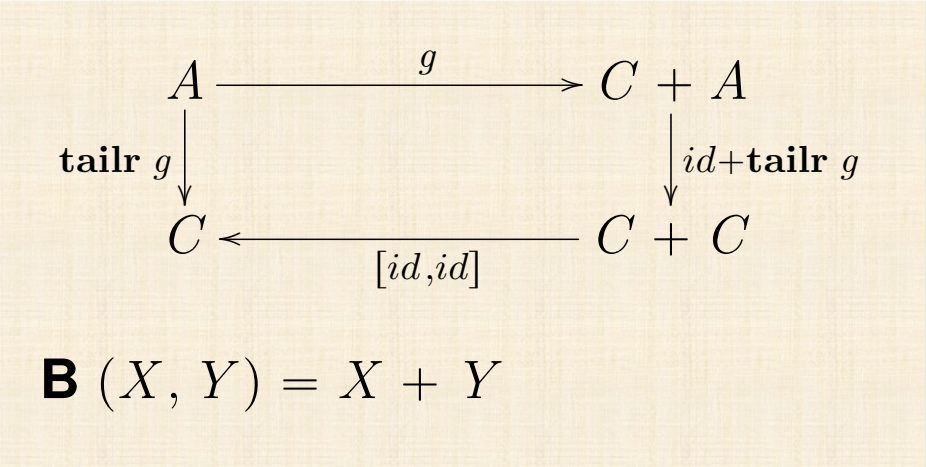

While (A= TRUE) Do
B
End While
$\mathbf{while} : (A \to \mathbb{B}) \to (A \to A) \to A \to A$
$\mathbf{while}\ a\ b\ x =$
$\mathbf{if}\ a\ x$
$\mathbf{then\ while}\ a\ b\ (b\ x)$
$\mathbf{else}\ x$
B

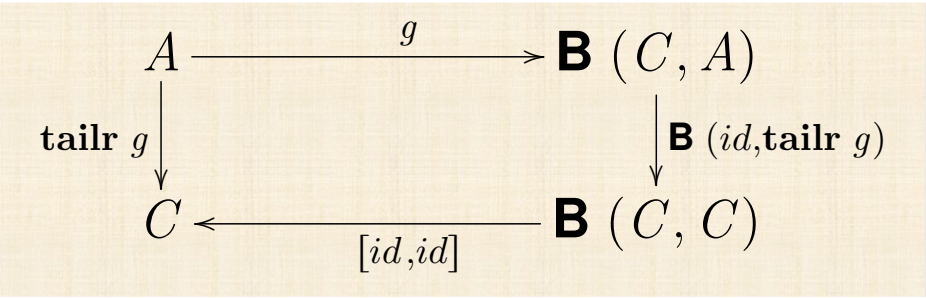
‘While loop’

$$
\begin{aligned}
&\mathbf{while} :: (A \to \mathbb{B}) \to (A \to A) \to (A \to C) \to A \to C \\
&\mathbf{while}\ a\ b\ c\ x = \\
&\quad \mathbf{if}\ a\ x \\
&\quad \mathbf{then\ while}\ a\ b\ c\ (b\ x) \\
&\quad \mathbf{else}\ c\ x
\end{aligned}
$$

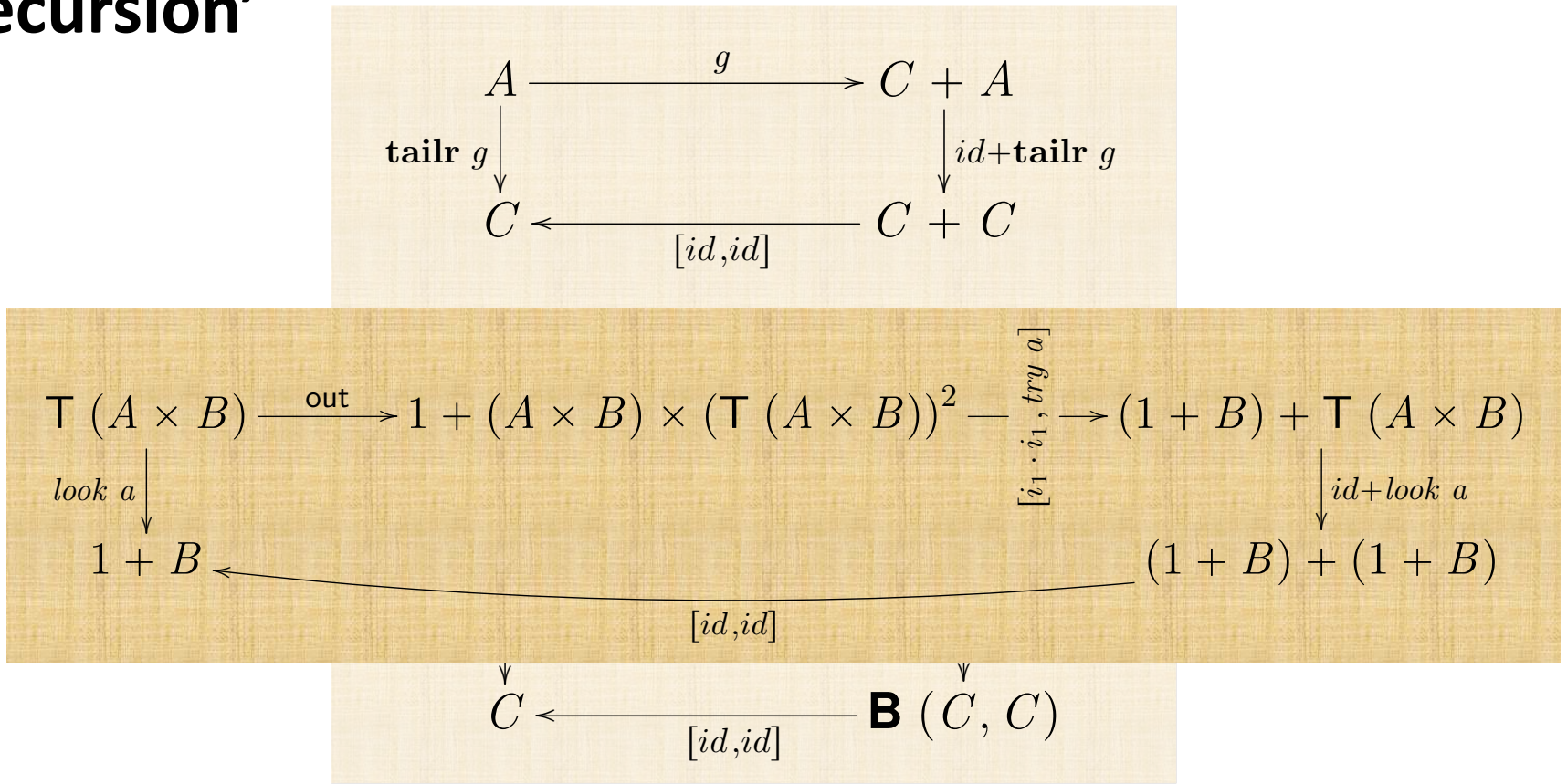


‘Tail recursion’

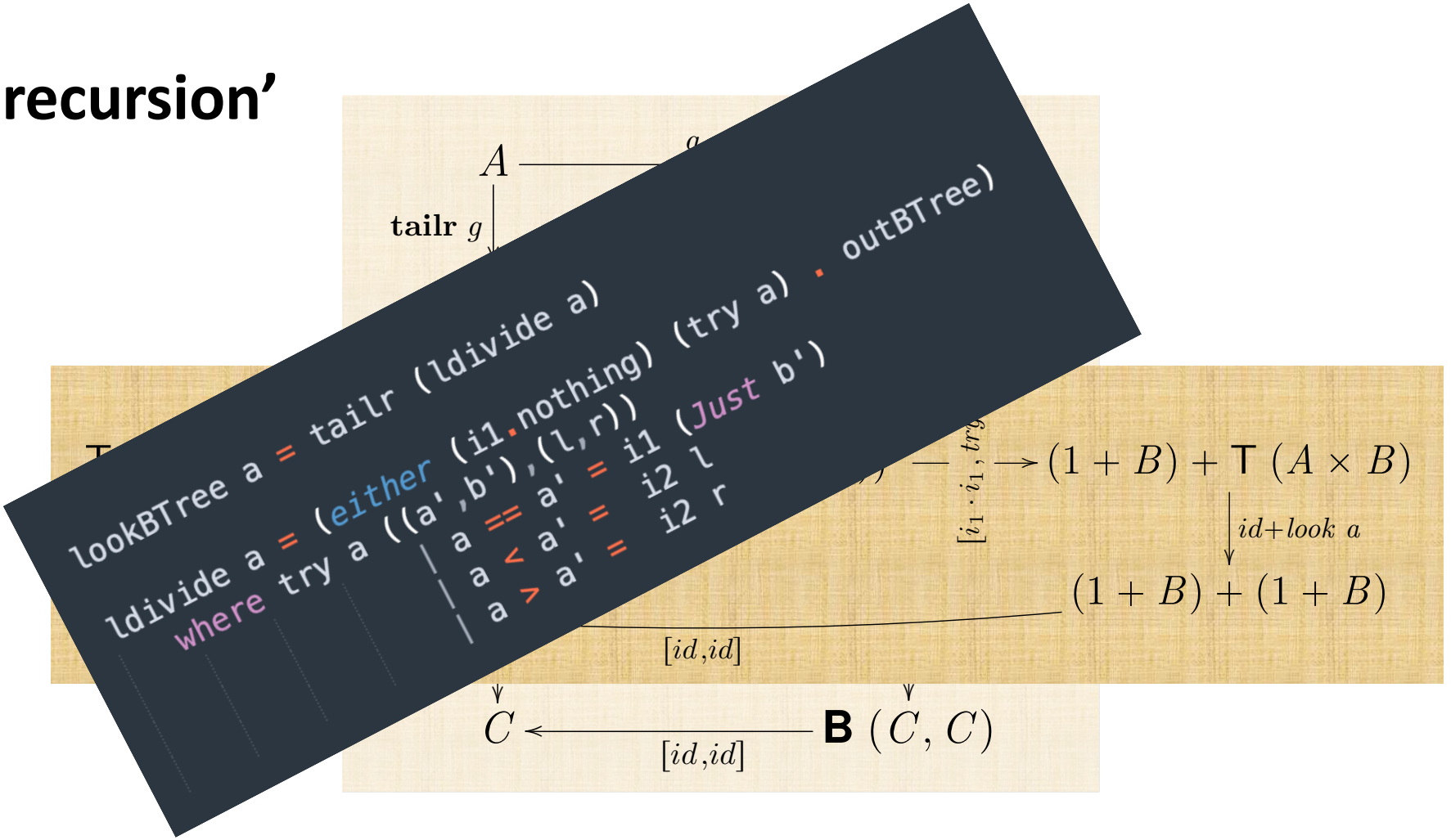




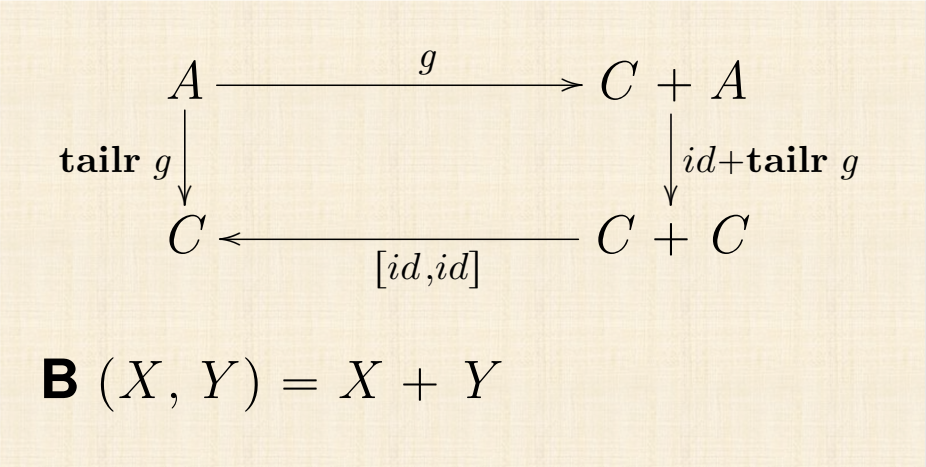
‘Tail recursion’

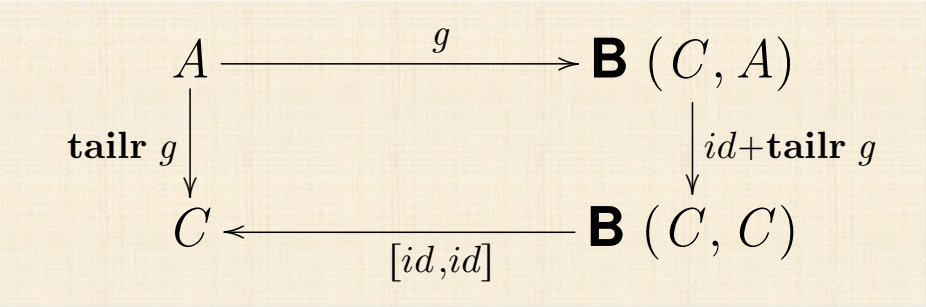


‘Tail recursion’

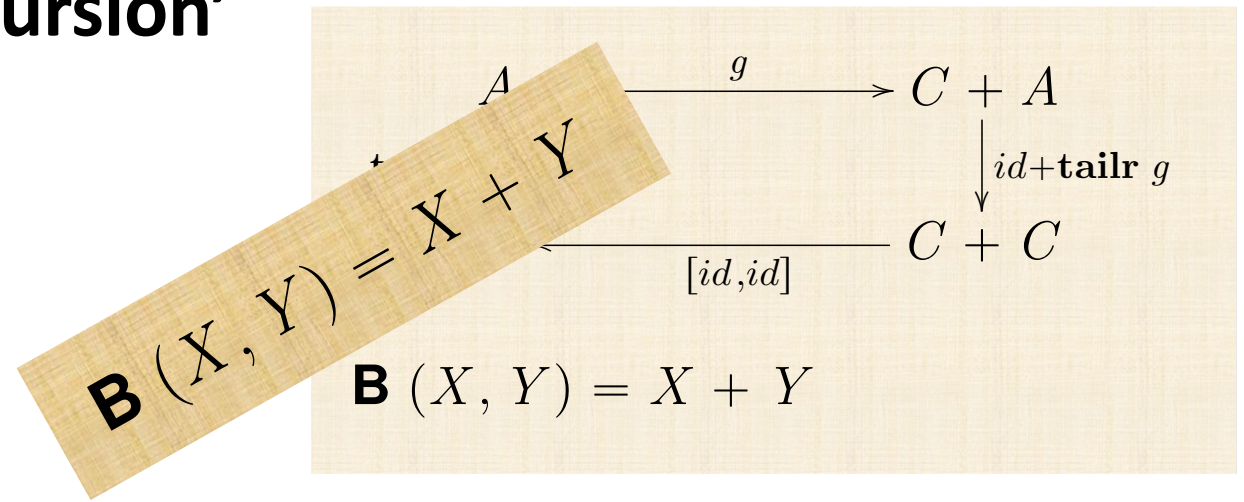


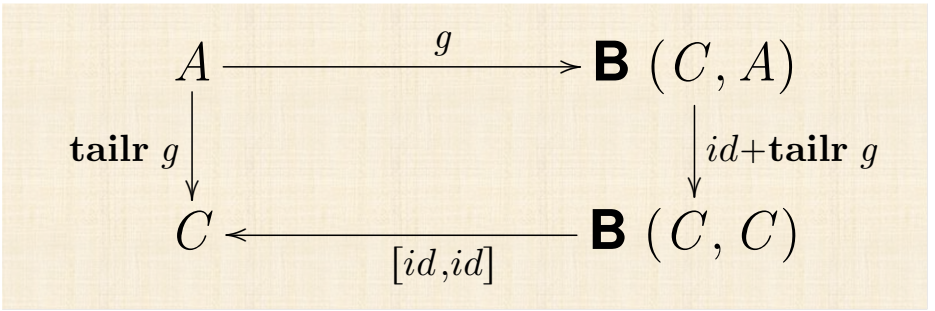
‘Tail recursion’





‘Tail recursion’

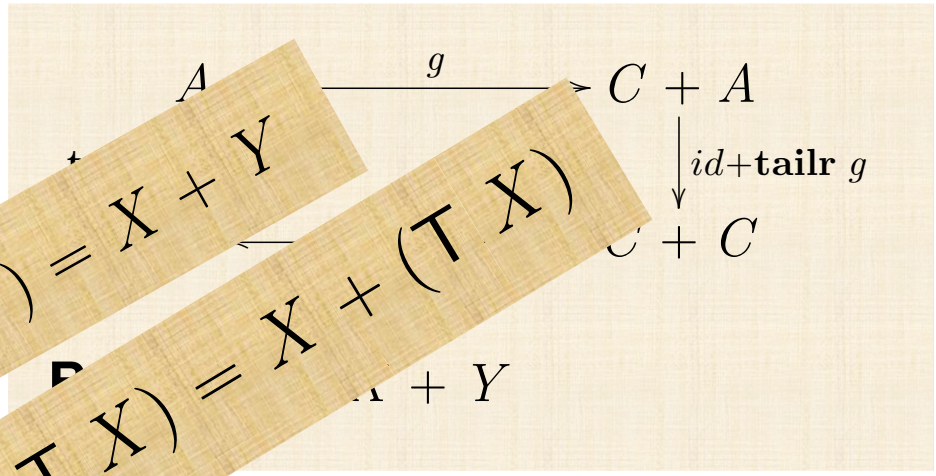


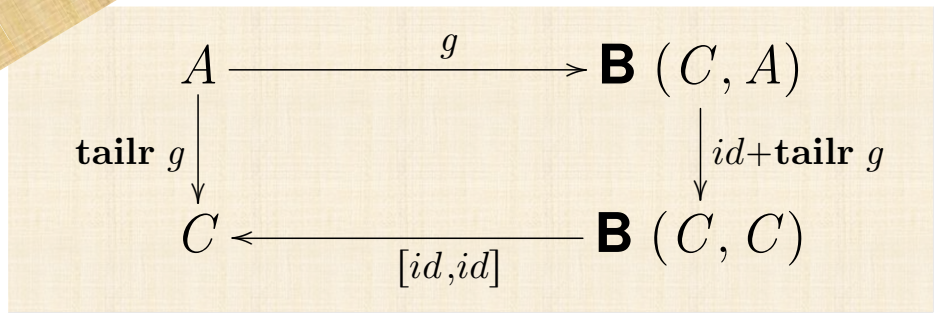


‘Tail recursion’

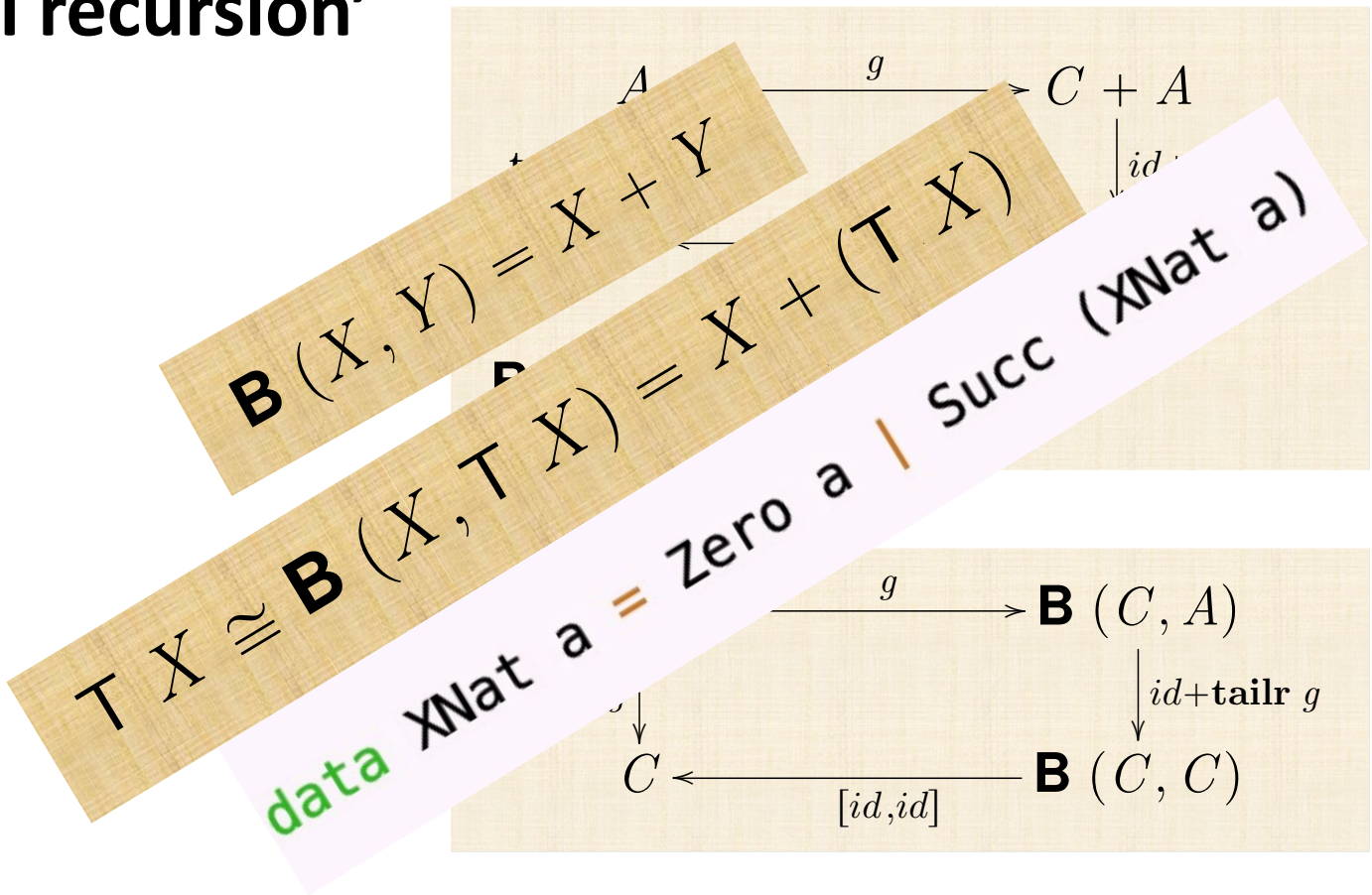
$$\mathbf{B}(X, Y) \cong X + Y$$

$$\mathsf{T}\,X \cong \mathbf{B}(X, \mathsf{T}\,X) \cong X + (\mathsf{T}\,X)$$

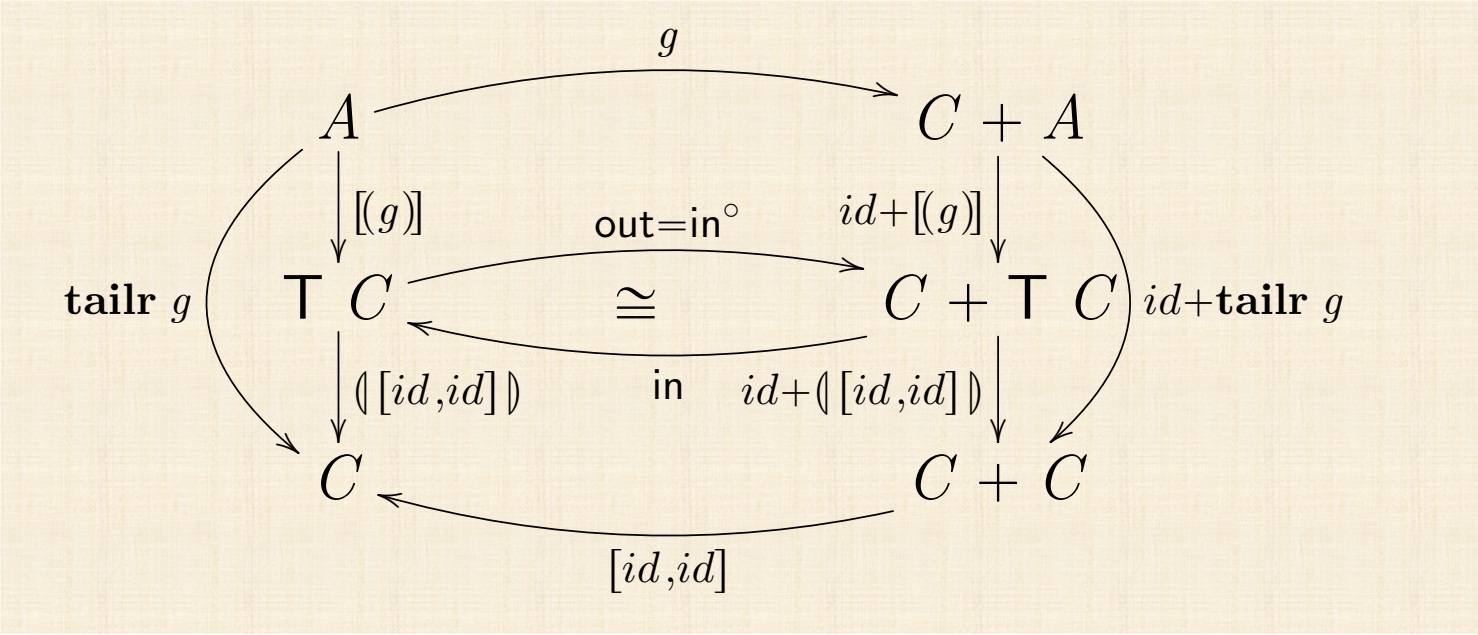




‘Tail recursion’



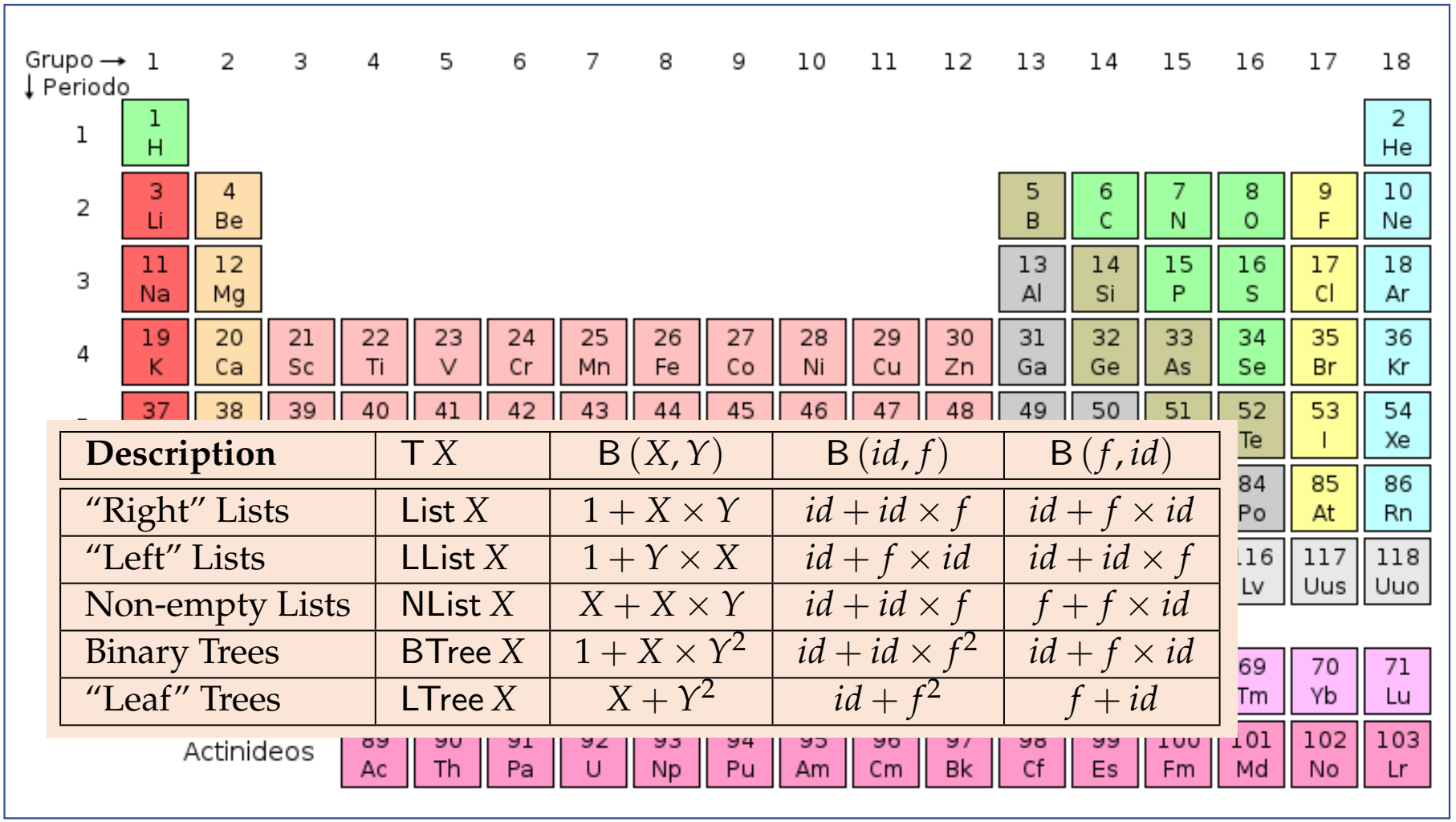
‘Tail recursion’ hylo



$$\mathsf{T}\ X \cong \mathbf{B}\ (X, \mathsf{T}\ X) = X + (\mathsf{T}\ X)$$

“Periodic table”

		B (X, Y)	$1 + Y$	$X + Y$	$1 + X \times Y$	$Z + X \times Y$	$X + Y^2$	$1 + X \times Y^2$
A	C	T X	$\mathbb{N}_0$	XNat X	X^*	SList X Z	LTree X	BTree X
$\mathbb{N}_0$	$\mathbb{N}_0$	Factorial			fac		$dfac$	
$\mathbb{N}_0$	$\mathbb{N}_0$	Misc. em $\mathbb{N}_0$	$(n*)$,$(n+)$, $\_^n$ etc		sq		dsq, fib	
$\mathbb{N}_0$	$\mathbb{N}_0$*	Séries			$odds$, $evens$			
$\mathbb{N}_0 \times X^*$	X^*	Selecção		$udrop$	$utake$			
$\mathbb{R}$	$\mathbb{R}$	Raiz quadrada		$\sqrt{\_}_\epsilon$				
X^*	X^*	Filtragem			filter p	filter p		
X^*	X^*	Ordenação	$bSort$		$iSort$, $sSort$		$mSort$	$qSort$
X^*	X^{**}	Grupos			$chunksOf\ n$			
$X^* \times X^*$	X^*	Junção				merge, $uconc$		
$X \times X^*$	X^*	Inserção				insert		
$\mathbb{B} \times \mathbb{N}_0$	$(\mathbb{N}_0 \times \mathbb{B})^*$	Puzzles						$hanoi$
BTree (X, Y)	$1 + Y$	Look-up		$lookup\ x$				
T (X, Y)	$1 + Y$	Look-up			$lookup\ x$			
T X	T X	Inversão			$reverse$		mirror	mirror
T X	$\mathbb{N}_0$	Cardinalidades			length		$count$	$count$
T X	$\mathbb{N}_0$	Profundidades					$depth$	$depth$
T X	X^*	Travessias					$tips$	$inordt$, $preordt$, $posordt$
T X	X^{**}	Caminhos			$prefixes$, $sufixes$			$traces$
T (T X)	T X	'Multiplicação'			μ		μ	



Description	T X	B (X, Y)	B (id, f)	B (f, id)
"Right" Lists	List X	$1 + X \times Y$	$id + id \times f$	$id + f \times id$
"Left" Lists	LList X	$1 + Y \times X$	$id + f \times id$	$id + id \times f$
Non-empty Lists	NList X	$X + X \times Y$	$id + id \times f$	$f + f \times id$
Binary Trees	BTree X	$1 + X \times Y^2$	$id + id \times f^2$	$id + f \times id$
"Leaf" Trees	LTree X	$X + Y^2$	$id + f^2$	$f + id$

Lecture: The Google MapReduce

http://research.google.com/archive/mapreduce.html

10/03/2014

Romain Jacotin

romain.jacotin@orange.fr

Monads

4

WHY MONADS MATTER

In this chapter we present a powerful device in state-of-the-art functional programming, that of a *monad*. The monad concept is nowadays of primary importance in computing science because it makes it possible to describe computational effects as disparate as input/output, comprehension notation, state variable updating, probabilistic behaviour, context dependence, partial behaviour *etc.* in an elegant and uniform way.

Our motivation to this concept will start from a well-known problem in functional programming (and computing as a whole) — that of coping with undefined computations.

4

WHY MONADS MATTER

In this chapter we present a powerful device in tional programming, that of a *monad*. The monad of primary importance in computing science bec sible to describe computational effects as dispar comprehension notation, state variable updating, probabilistic behaviour, context dependence, partial behaviour *etc.* in an elegant and uniform way.

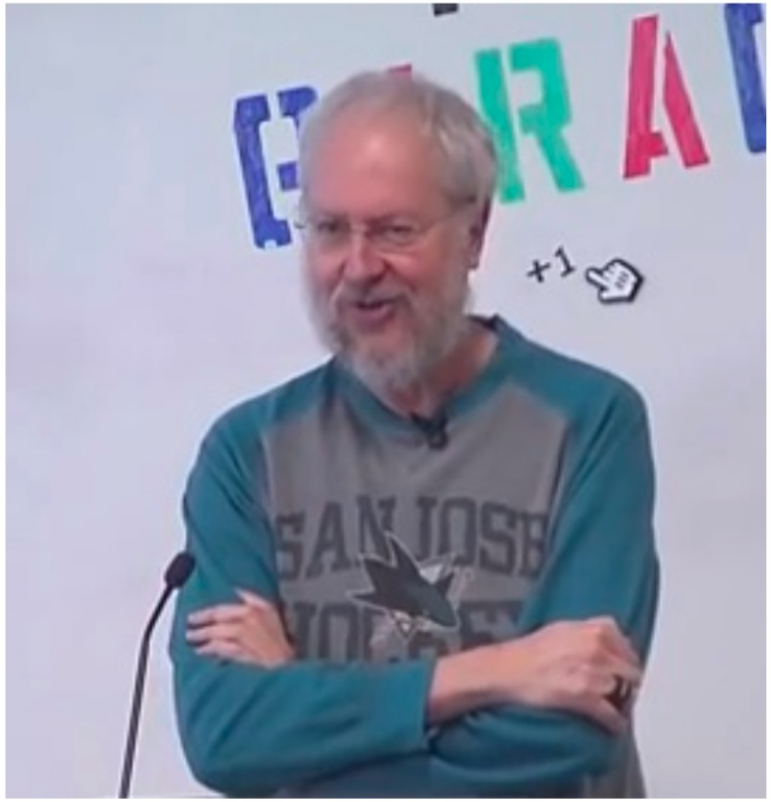
Our motivation to this concept will start from a well-known problem in functional programming (and computing as a whole) — that of coping with undefined computations.


CHANCE

Probability of the sum

"Monads [...] come with a curse. The monadic curse is that once someone learns what monads are and how to use them, they lose the ability to explain it to other people"

(Douglas Crockford: *Google Tech Talk on how to express monads in JavaScript* YouTube 2013)



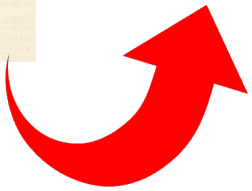
Douglas Crockford (2013)

Partial functions

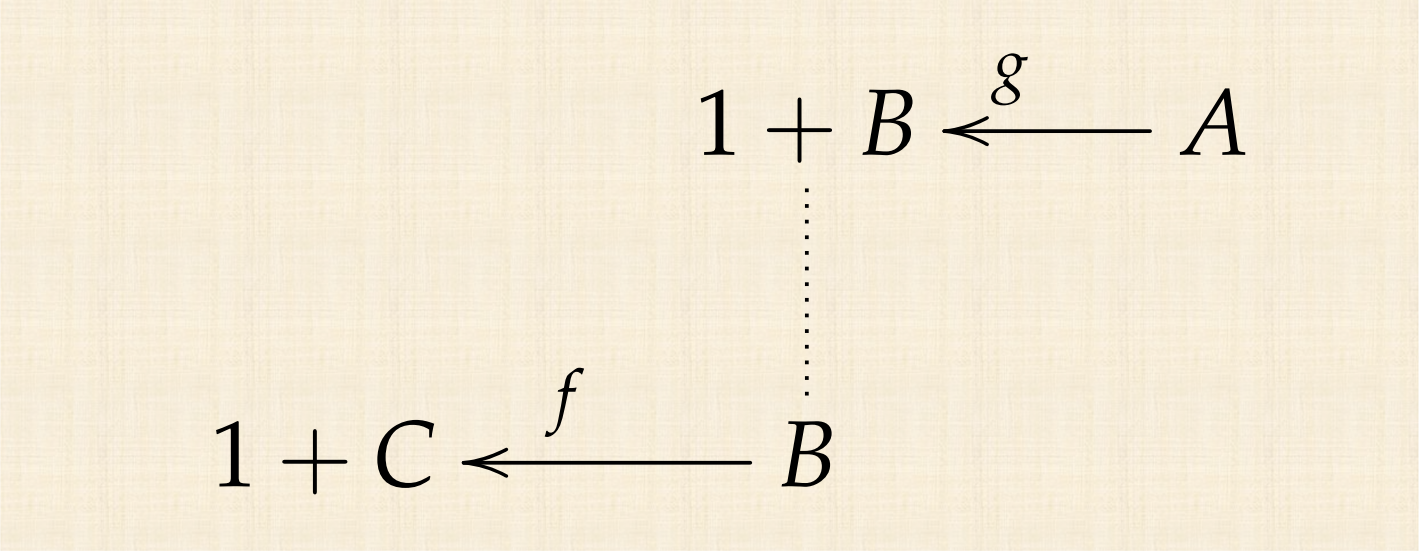
$$1 + B \xleftarrow{g} A$$

$$B \xleftarrow{f} A$$

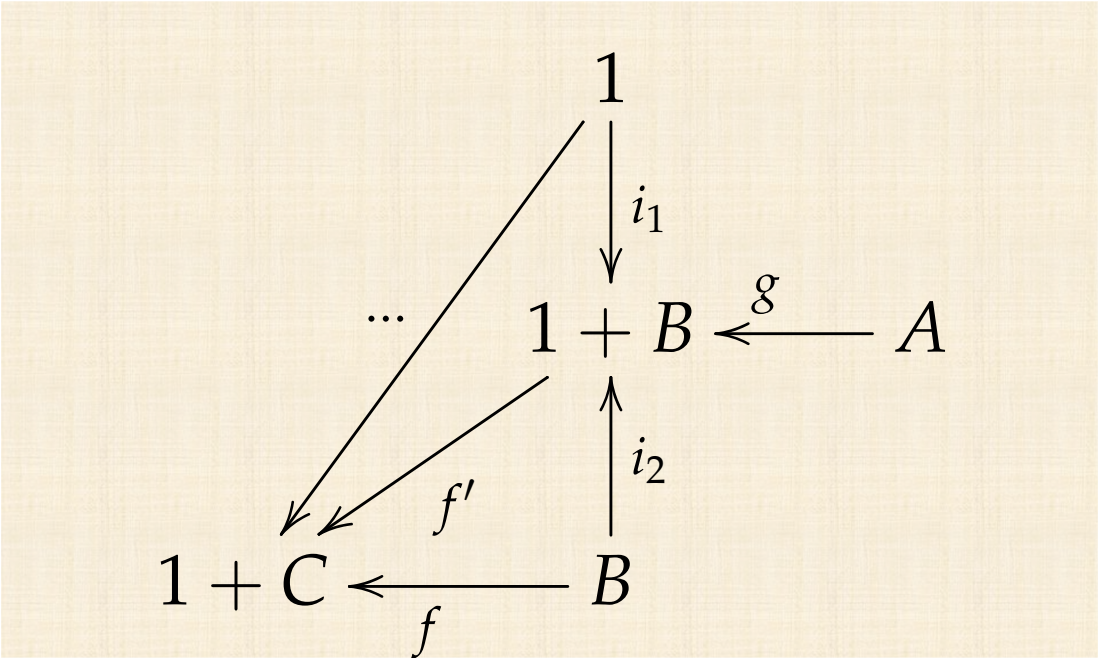
$$1 + B \xleftarrow{i_2 \cdot f} A$$



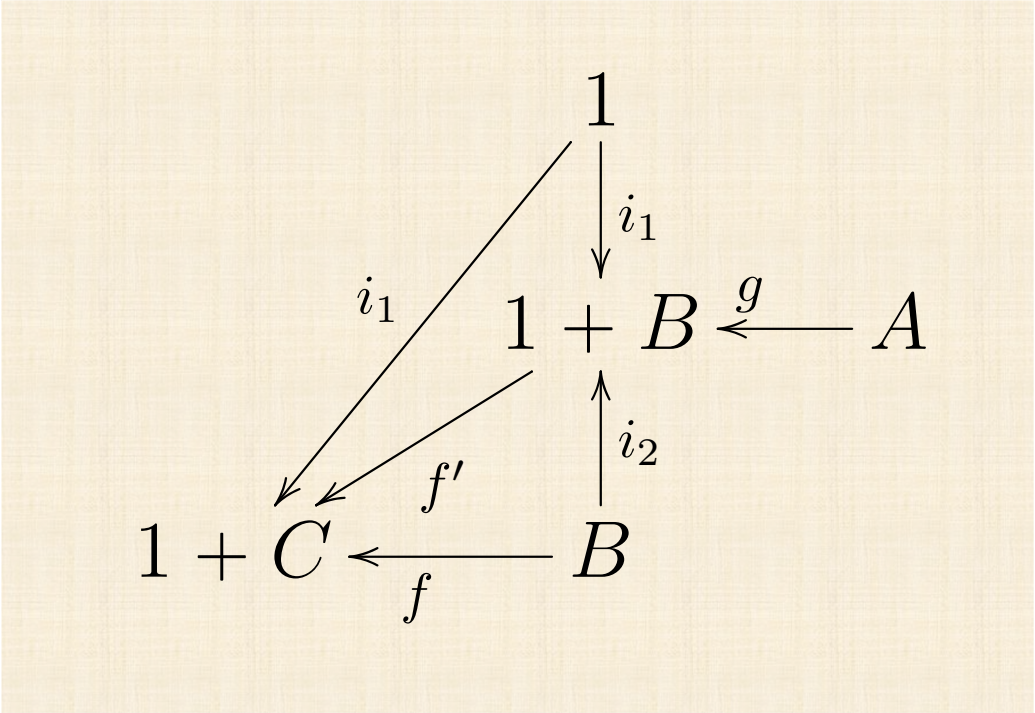
Partial functions



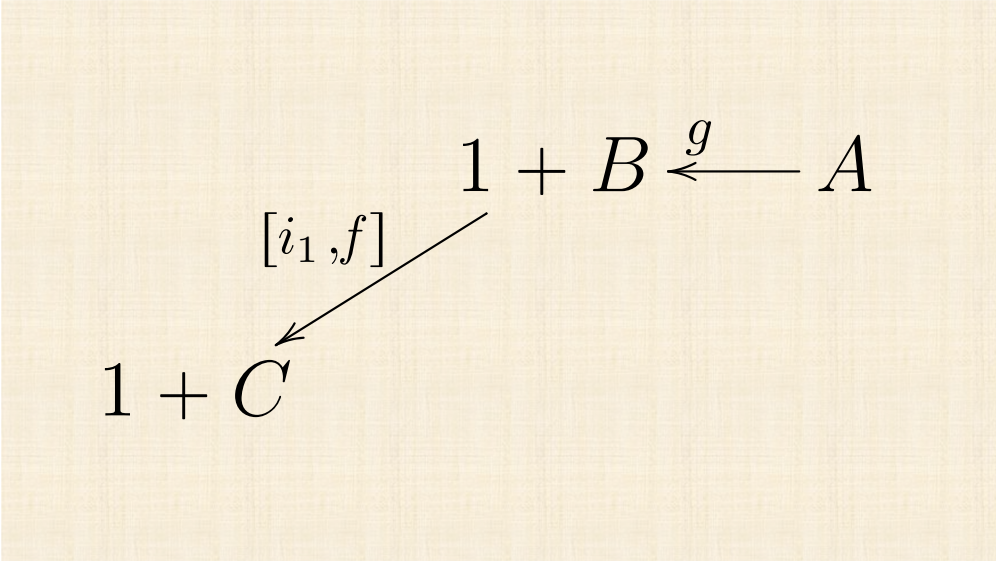
Partial functions



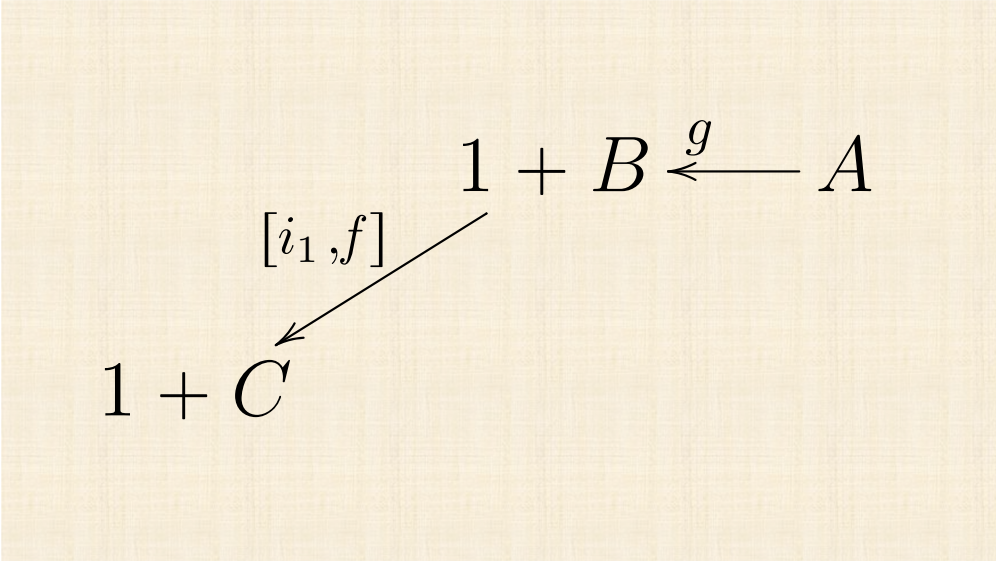
Partial functions



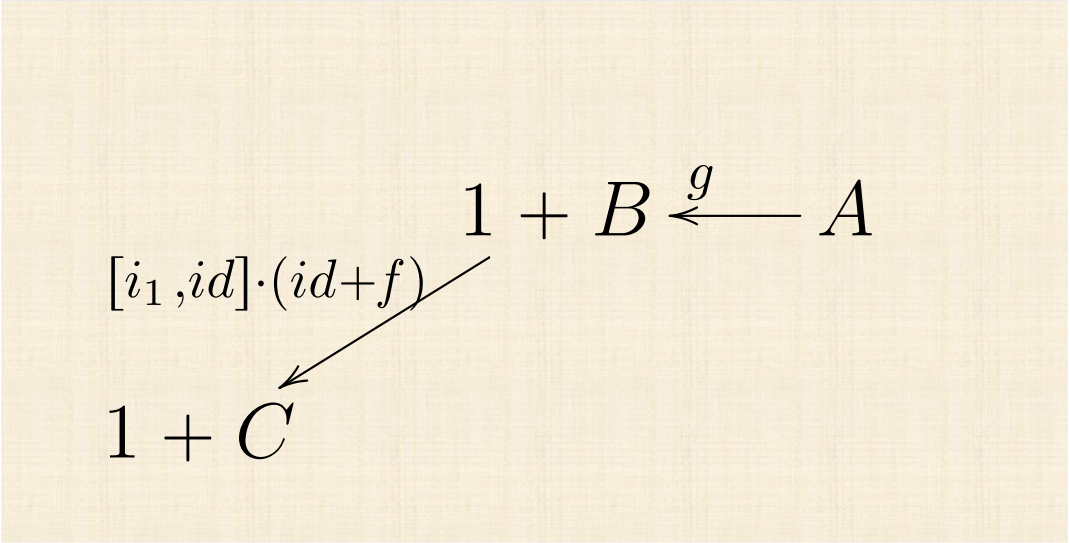
Partial functions



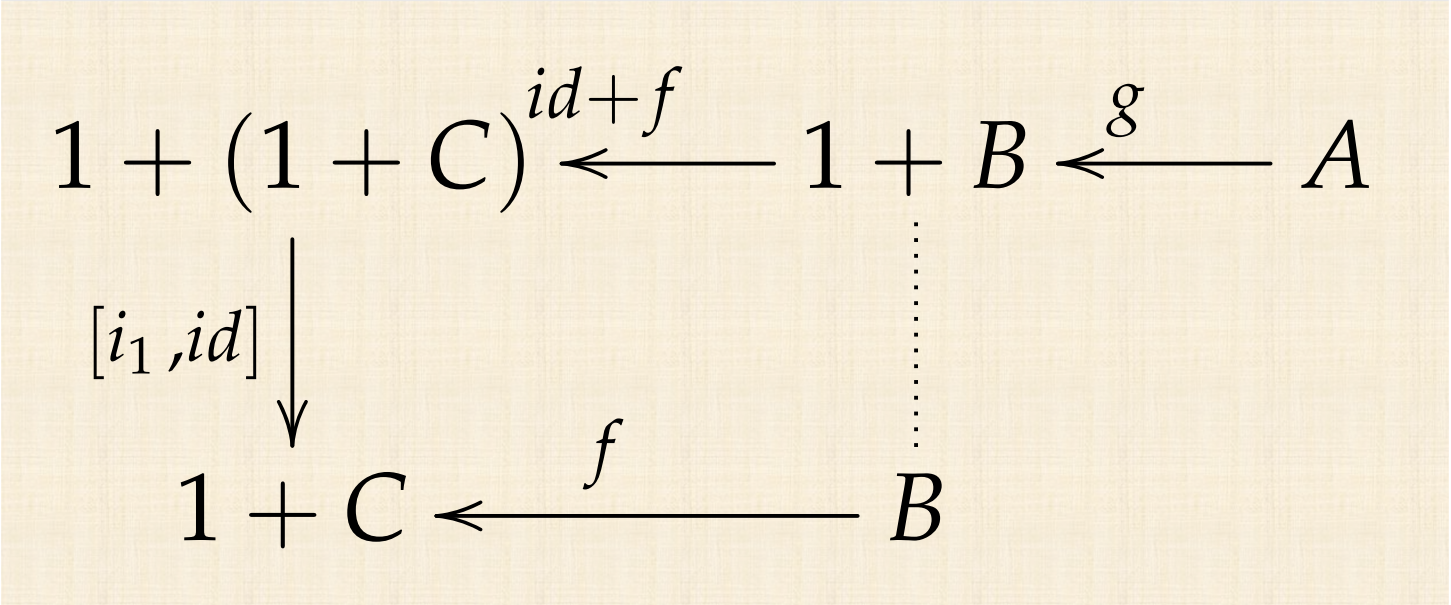
Partial functions



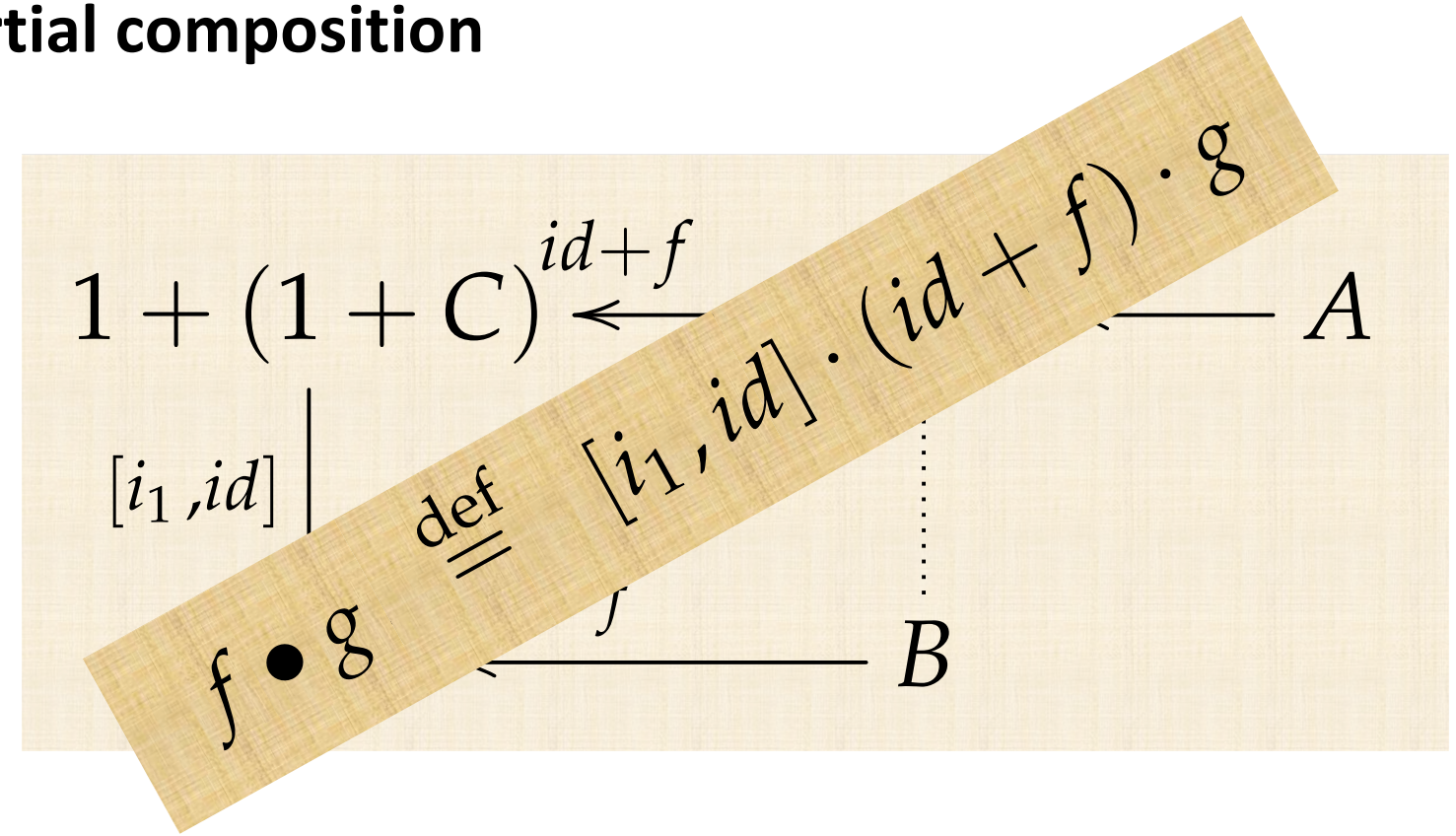
Partial functions



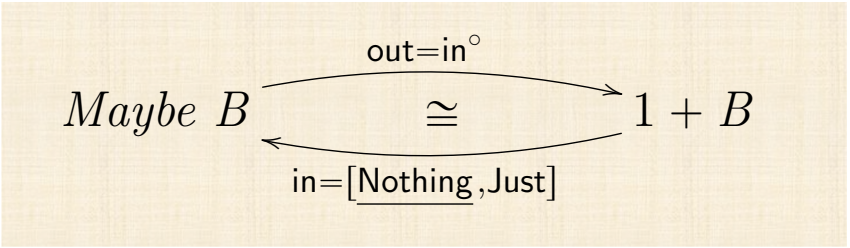
Partial composition

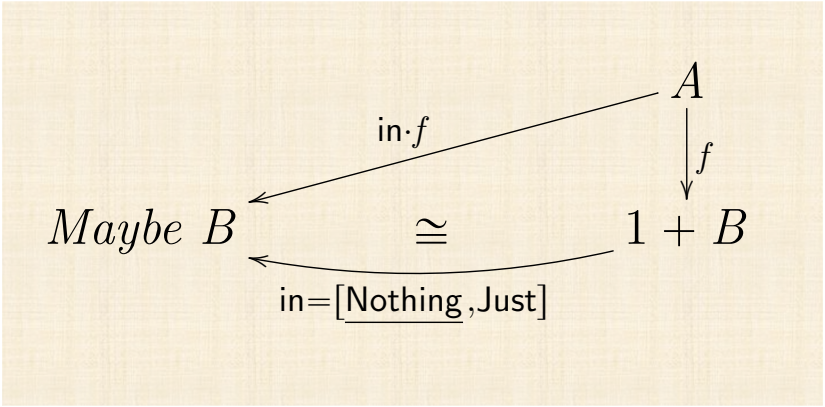


Partial composition



‘Maybe functions’

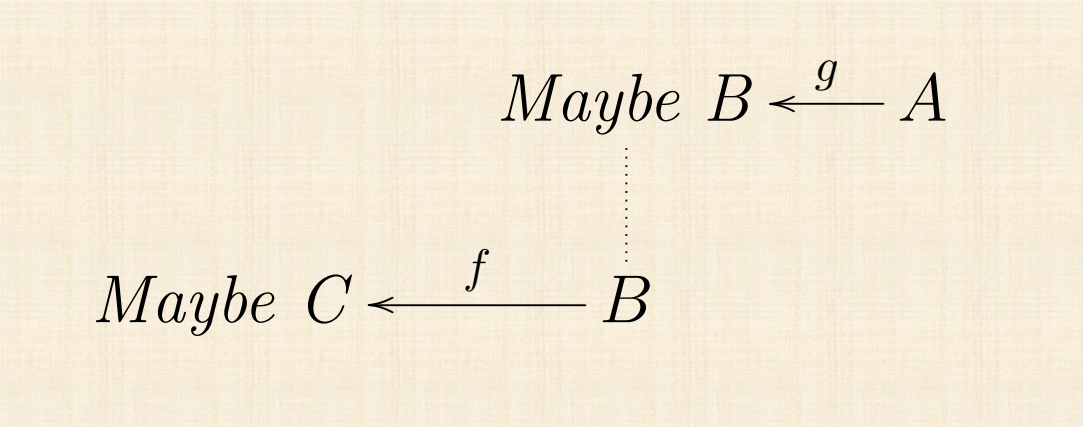




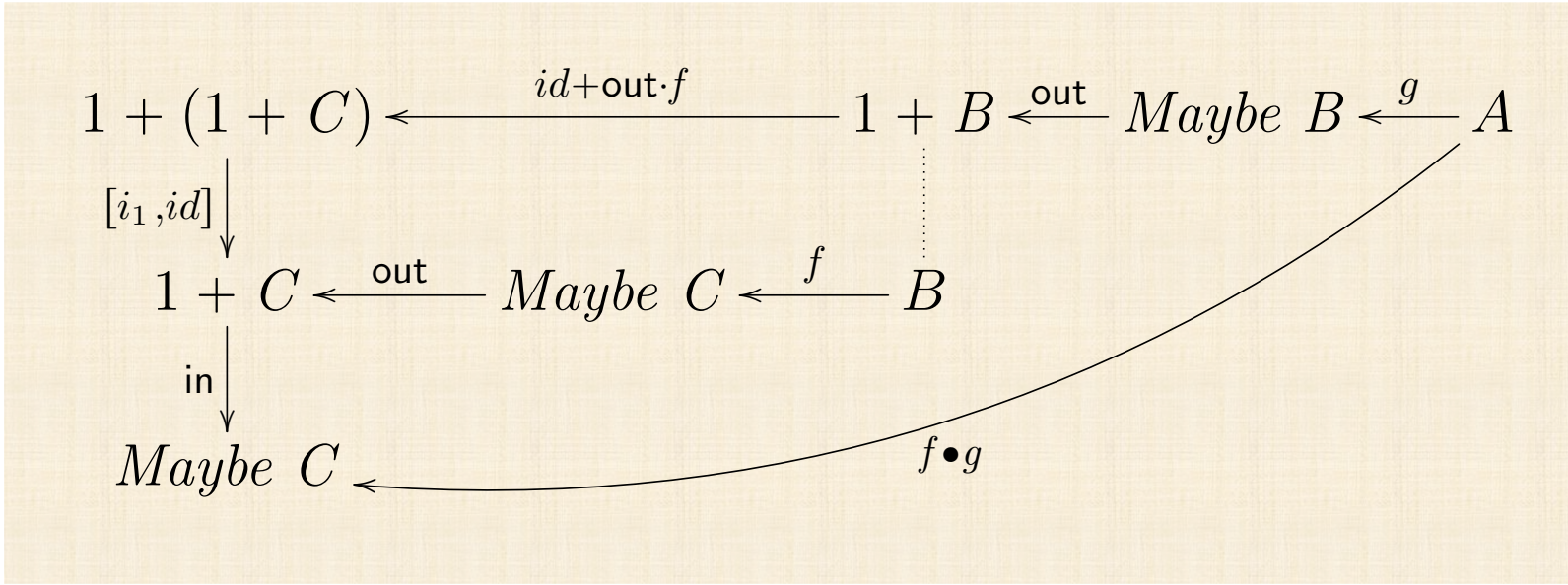
$$\text{out Nothing} = i_1\ ()$$
$$\text{out (Just } a) = i_2\ a$$

```
data Maybe a = Nothing | Just a
```

Composing ‘Maybe functions’



Composing ‘Maybe functions’



Composing ‘Maybe functions’

$$
\begin{aligned}
& f \bullet g = \mathsf{in} \cdot [i_1, \mathsf{out} \cdot f] \cdot \mathsf{out} \cdot g \\
\equiv & \quad \{ \text{ fusão-+ e } \mathsf{in} \cdot \mathsf{out} = id \ \} \\
& f \bullet g = [\mathsf{in} \cdot i_1, f] \cdot \mathsf{out} \cdot g \\
\equiv & \quad \{ \text{ introdução da variável } a \ \} \\
& (f \bullet g)\ a = [\mathsf{in} \cdot i_1, f]\ (\mathsf{out}\ (g\ a)) \\
\equiv & \quad \{ \text{ definição de out} \} \\
& (f \bullet g)\ a = \mathbf{if}\ g\ a = \mathsf{Nothing}\ \mathbf{then}\ [\mathsf{in} \cdot i_1, f]\ (i_1\ ())\ \mathbf{else}\ [\mathsf{in} \cdot i_1, f]\ (i_2\ (g\ a)) \\
\equiv & \quad \{ \text{ cancelamento-+ e simplificação} \} \\
& (f \bullet g)\ a = \mathbf{if}\ g\ a = \mathsf{Nothing}\ \mathbf{then}\ \mathsf{Nothing}\ \mathbf{else}\ f\ (g\ a)
\end{aligned}
$$

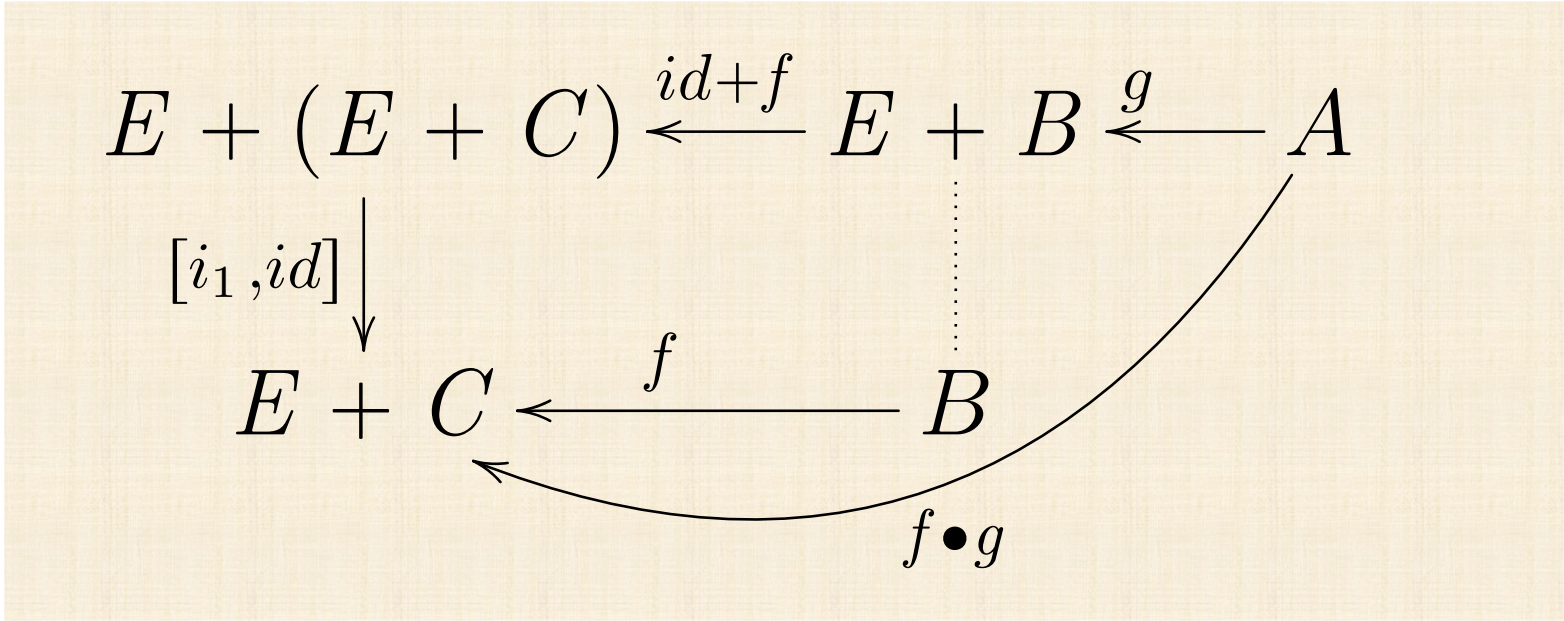
Error messages

$$E + B \xleftarrow{g} A$$

$$B \xleftarrow{f} A$$

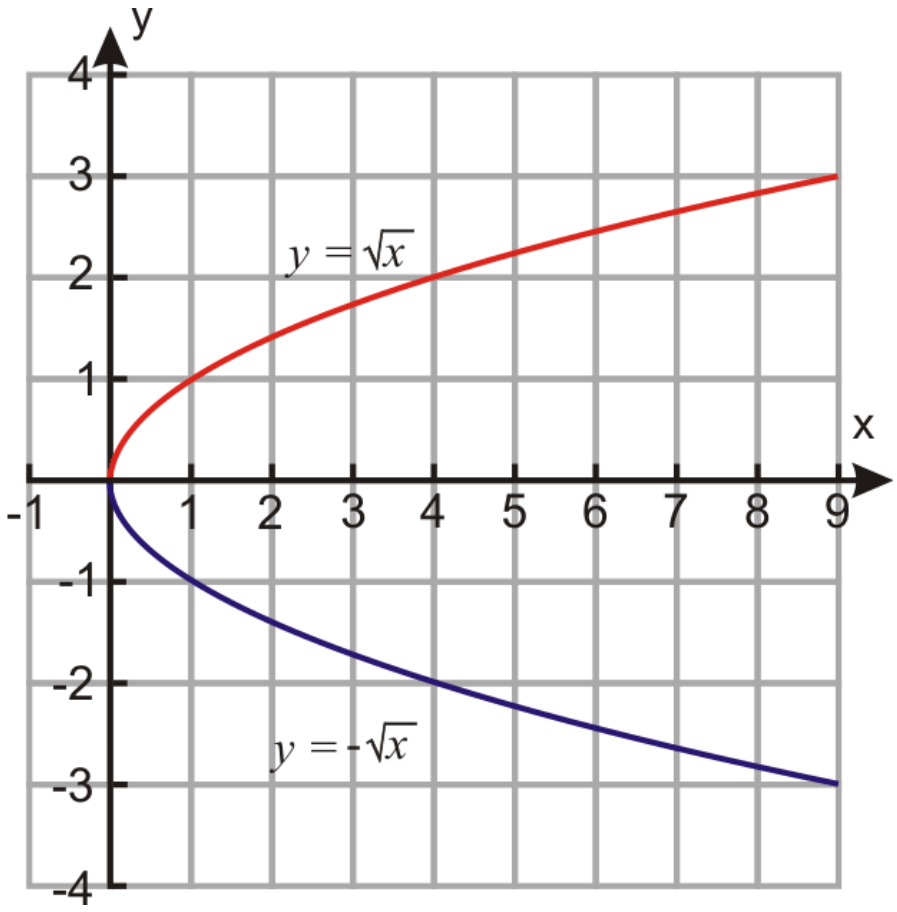
$$E + B \xleftarrow{i_2 \cdot f} A$$

Handling error messages




Unchecked Exception
Checked Exception
Error
Exception Handling

Square root “function”

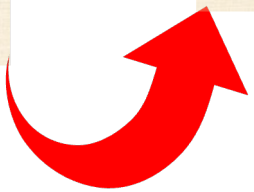


'Undecided' ("nondeterministic") functions

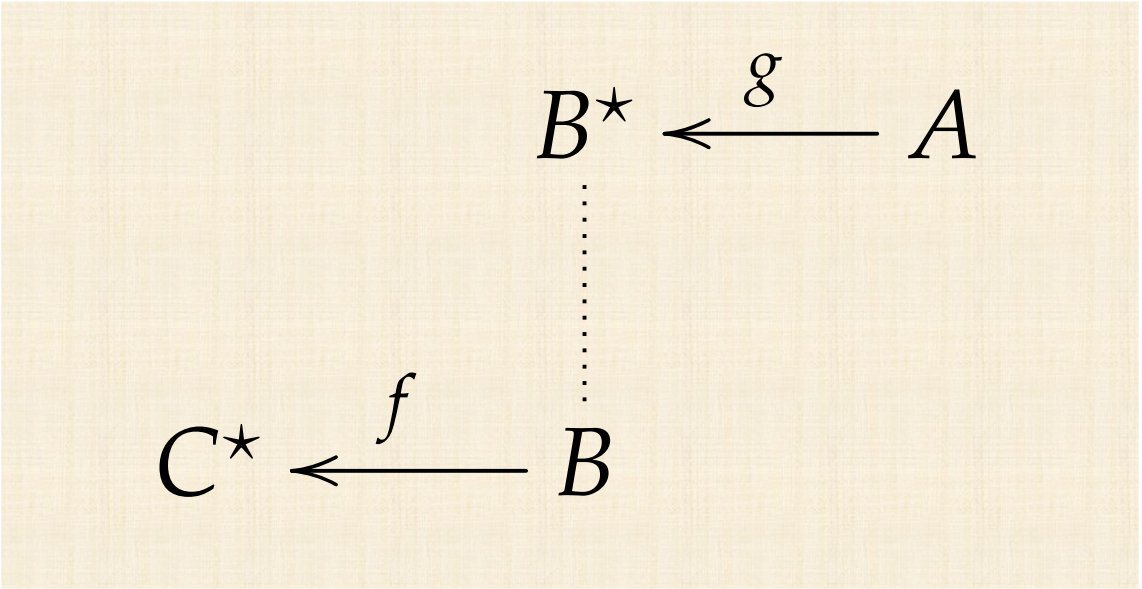
$$B^{\star} \xleftarrow{g} A$$

$$B \xleftarrow{f} A$$

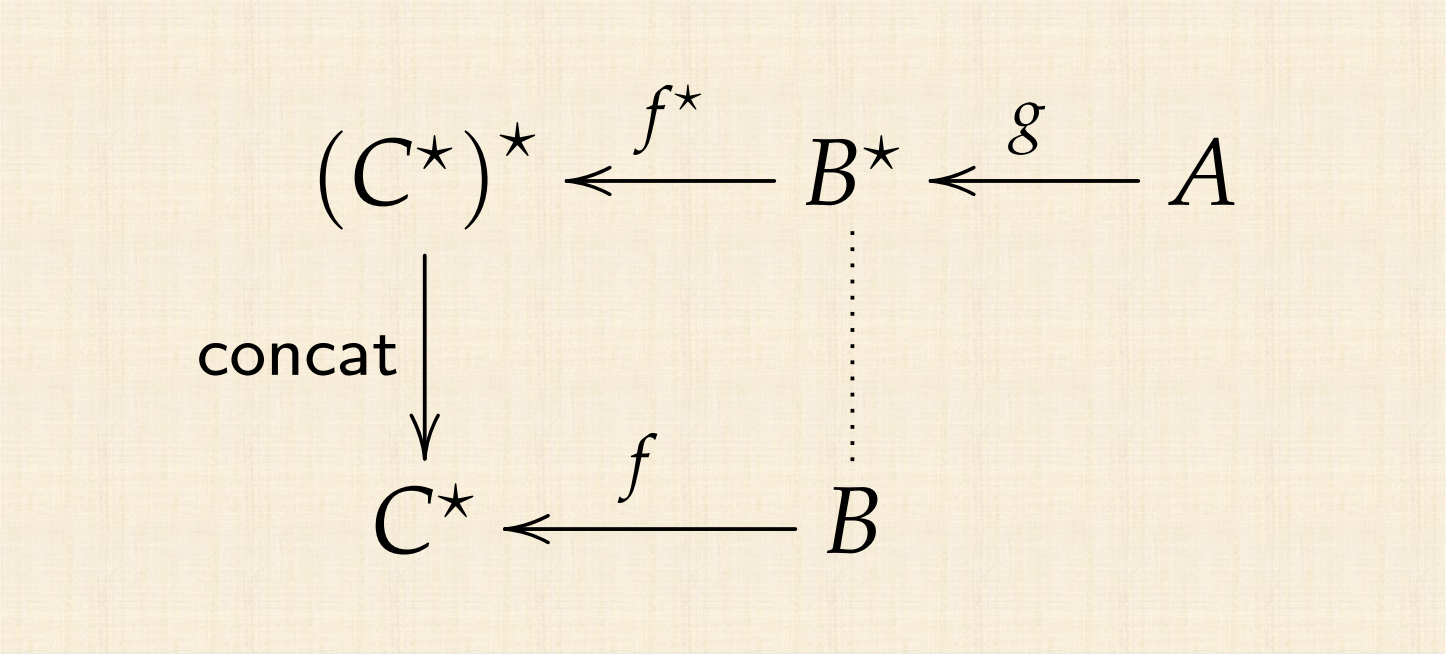
$$B^{*} \xleftarrow{\text{singl}\cdot f} A$$



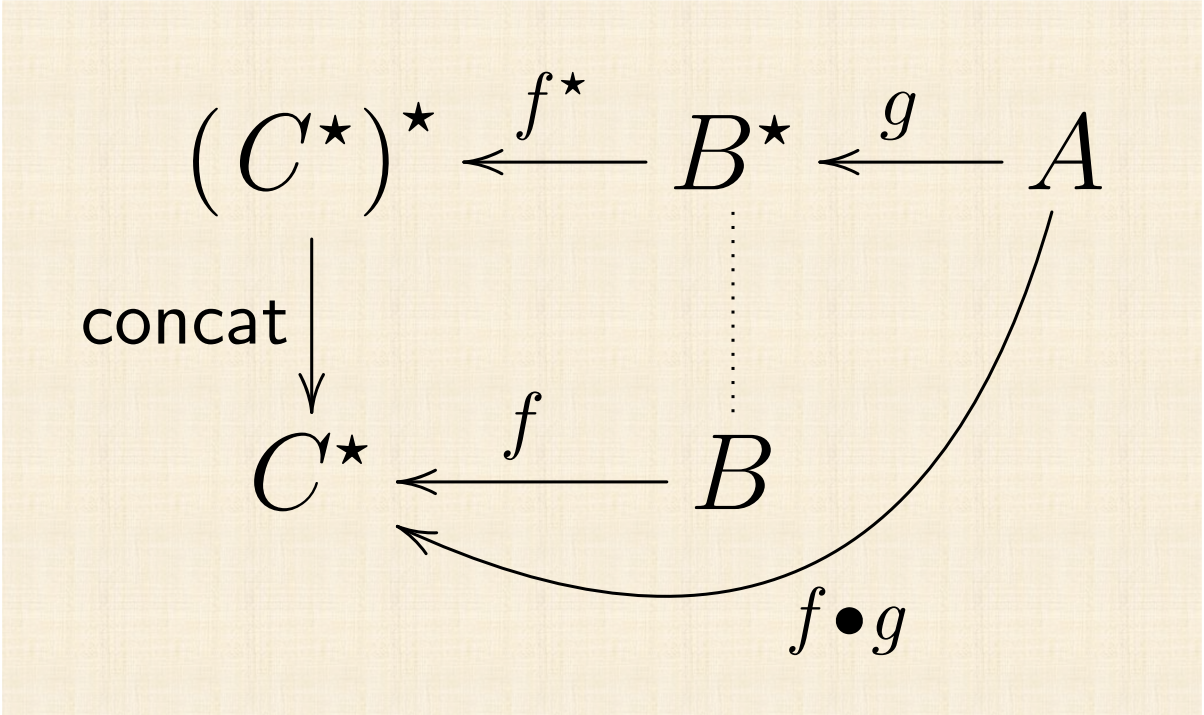
Composing 'undecided' functions



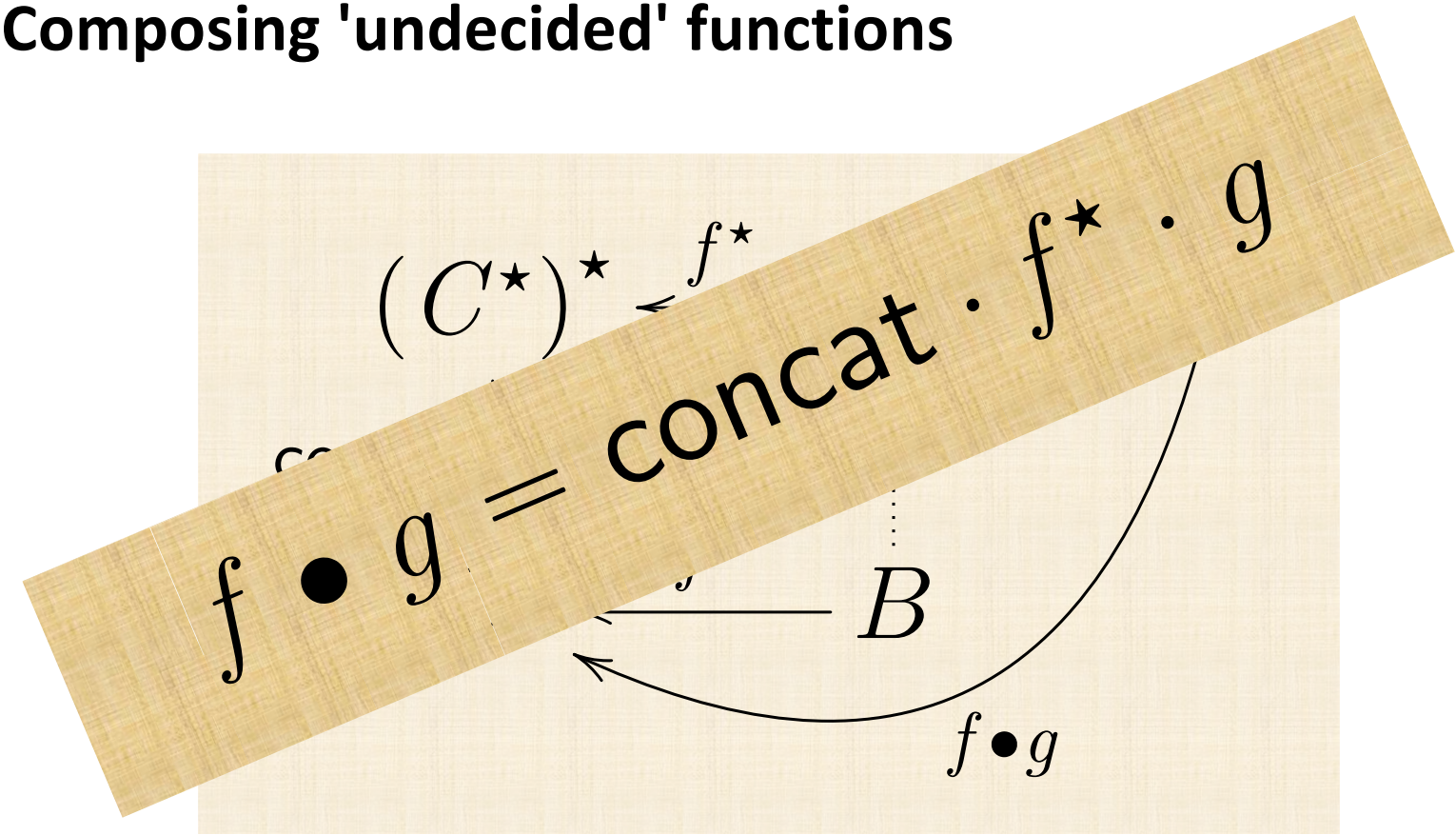
Composing 'undecided' functions



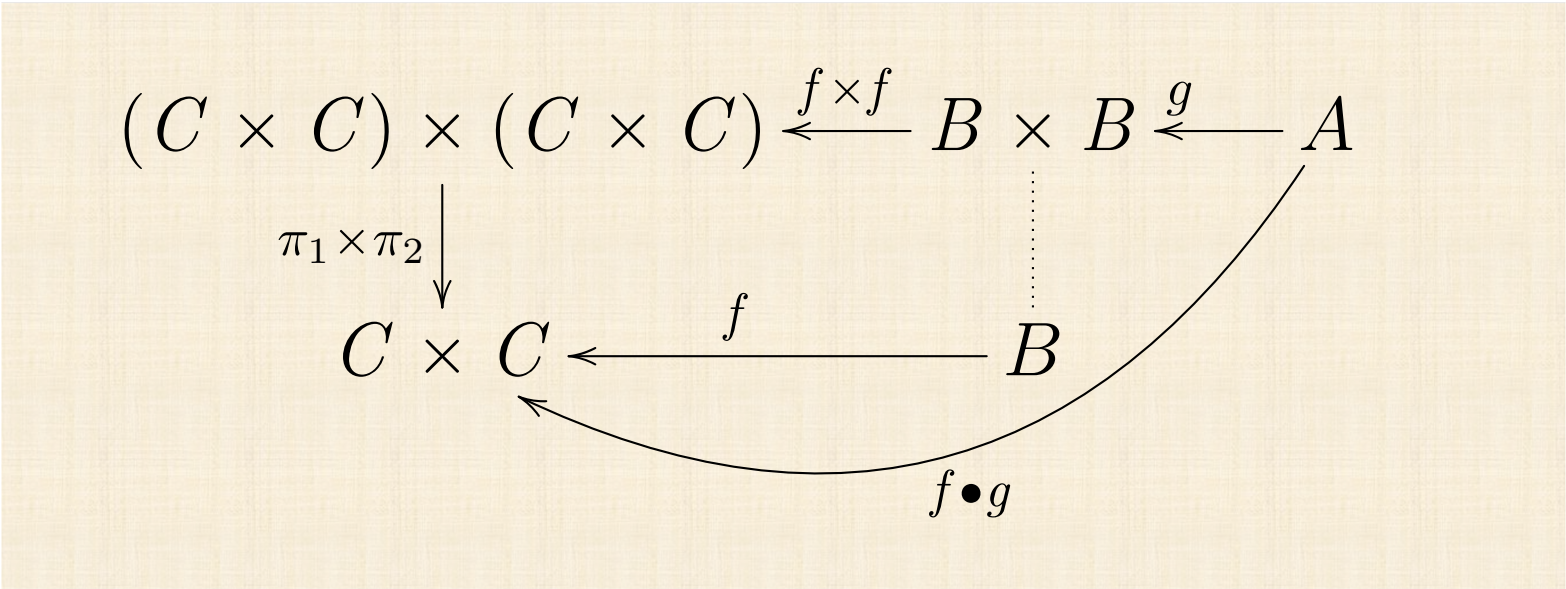
Composing 'undecided' functions



Composing 'undecided' functions



Composing functions that yield pairs



FUNCTORS so far

$$\mathbf{T}\ X = 1 + X$$

$$\mathbf{T}\ X = Maybe\ X$$

$$\mathbf{T}\ X = E + X$$

$$\mathbf{T}\ X = X^*$$

$$\mathbf{T}\ X = X \times X$$

Similar structure

$$X \xrightarrow{i_2} 1 + X \xleftarrow{[i_1, id]} 1 + (1 + X)$$

$$X \xrightarrow{i_2} E + X \xleftarrow{[i_1, id]} E + (E + X)$$

$$X \xrightarrow{\mathsf{Just}} Maybe\ X \xleftarrow{\mu} Maybe\ (Maybe\ X)$$

$$X \xrightarrow{singl} X^* \xleftarrow{\mathsf{concat}} (X^*)^*$$

$$X \xrightarrow{\langle id, id \rangle} X \times X \xleftarrow{\pi_1 \times \pi_2} (X \times X) \times (X \times X)$$

MONAD

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}\ (\mathbf{T}\ X)$$

MONAD

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}\ (\mathbf{T}\ X)$$

Unit Multiplication

MONAD

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}\ (\mathbf{T}\ X)$$

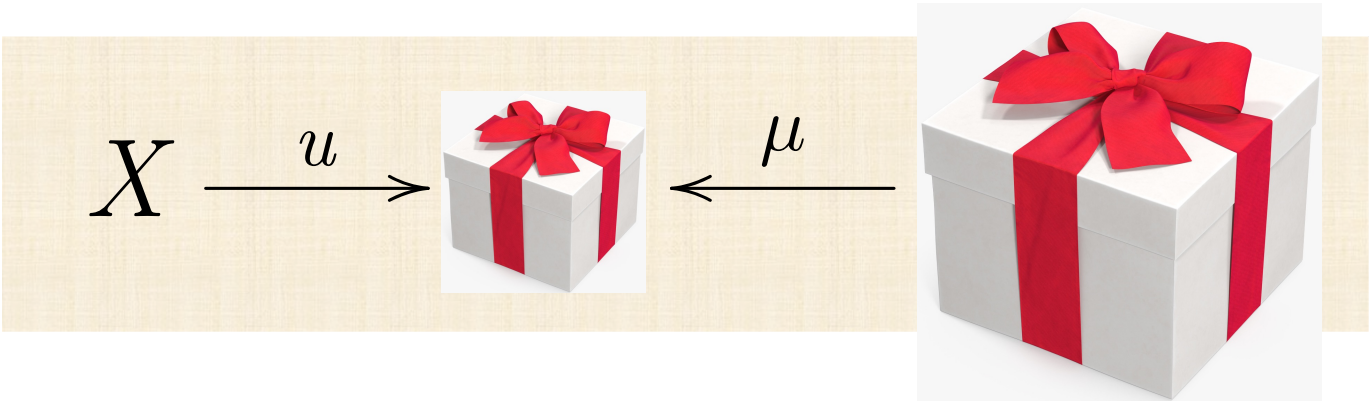
Monad = **Functor** + unit + multiplication

Cálculo de Programas
T12

Monad = “racing” functor

$$X \xrightarrow{u} \mathbf{T}\,X \xleftarrow{\mu} \mathbf{T}\,(\mathbf{T}\,X)$$

MONAD



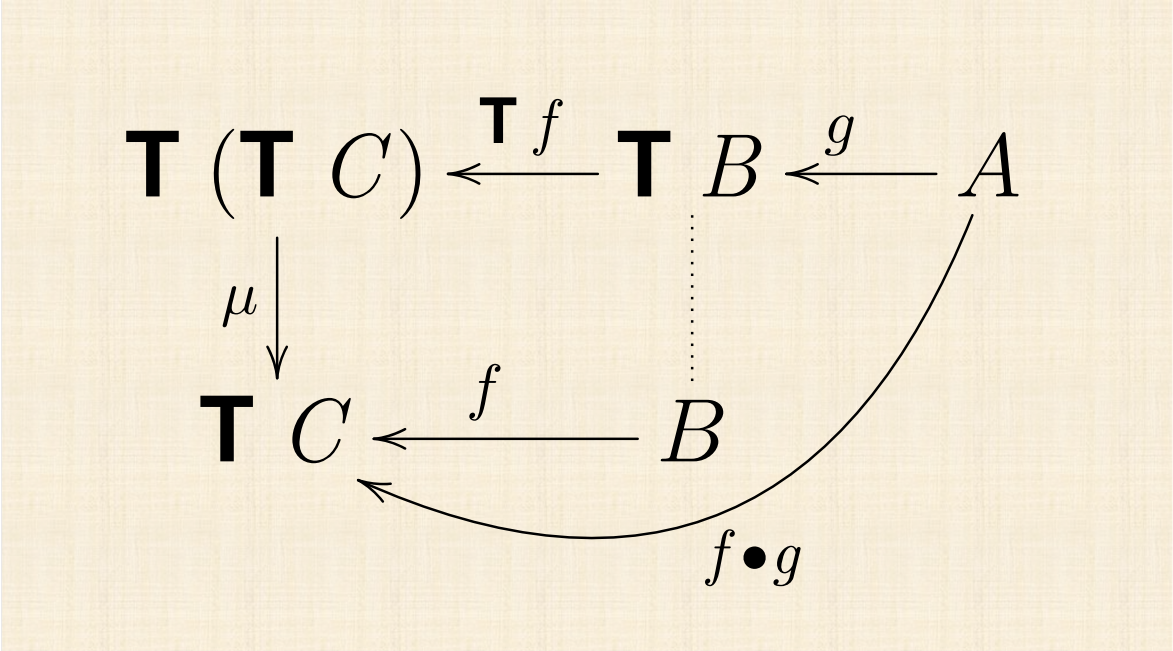
Monad = **Functor** + unit + multiplication

MONAD

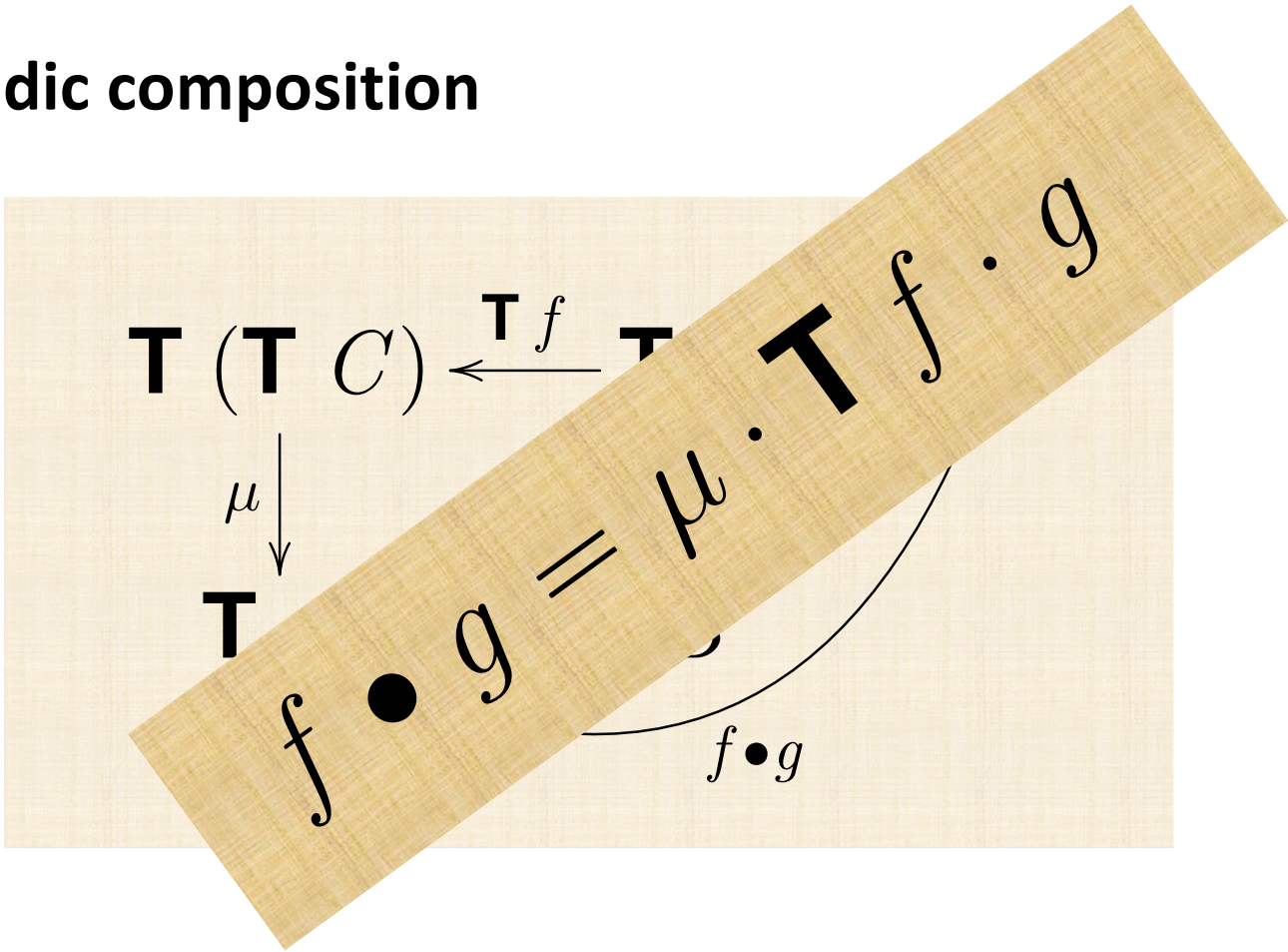
$=$

- pointers
- exceptions
- pairs
- distributions
- I/O operations
- ...

Monadic composition



Monadic composition



Heinrich Kleisli

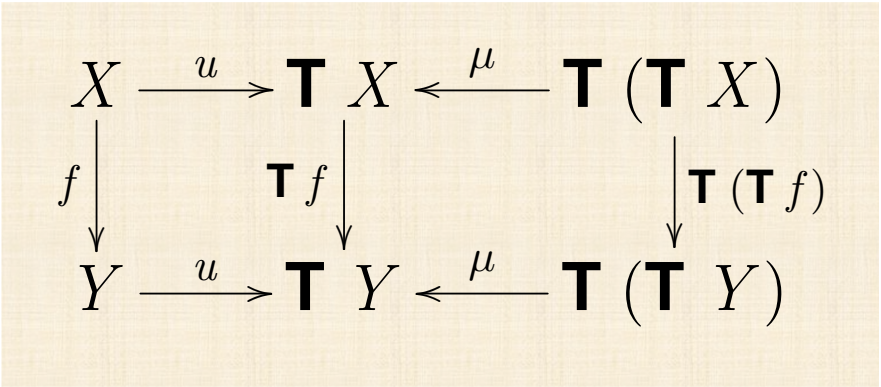
From Wikipedia, the free encyclopedia

Heinrich Kleisli (/ˈklaɪsli/; October 19, 1930 – April 5, 2011) was a Swiss mathematician. He is the namesake of several constructions in category theory, including the Kleisli category and Kleisli triples. He is also the namesake of the Kleisli Query System, a tool for integration of heterogeneous databases developed at the University of Pennsylvania.

Heinrich Kleisli in 1987

Kleisli earned his Ph.D. at ETH Zurich in 1960, having been supervised by Beno Eckmann and Ernst Specker. His dissertation was on homotopy and abelian categories. He served as an associate professor at the University of Ottawa before relocating to the University of Fribourg in 1966. He became a full professor at Fribourg in 1967.

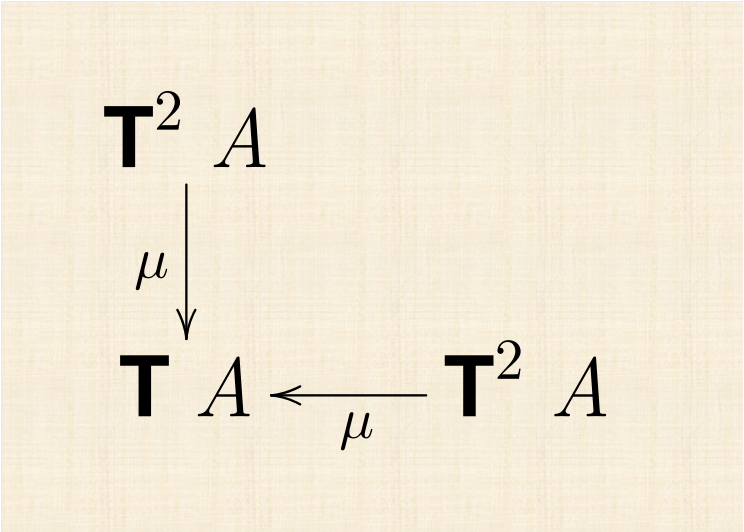
Monad – natural properties



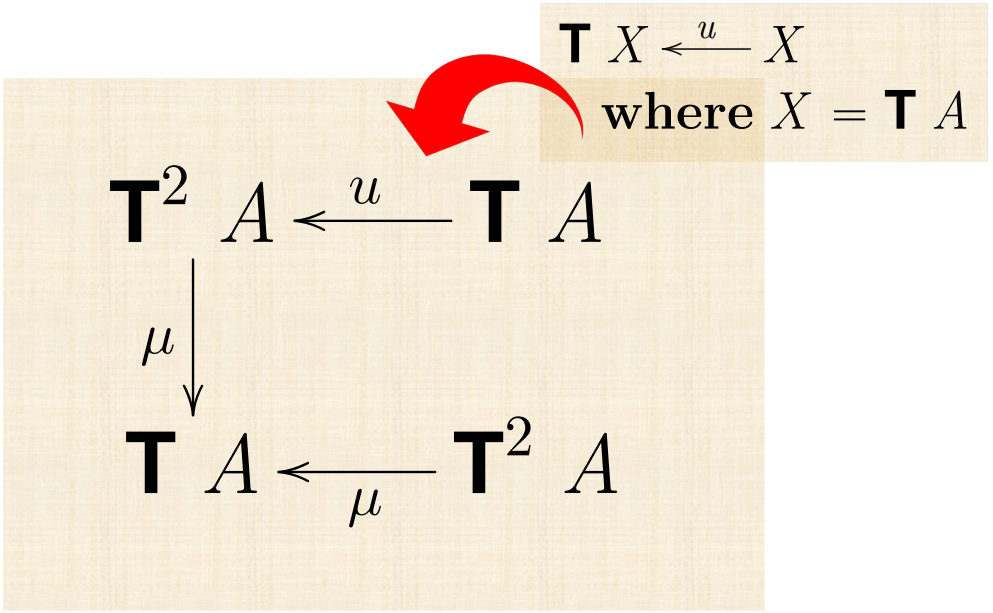
$$\mathbf{T}\, f \cdot u = u \cdot f$$

$$\mathbf{T}\, f \cdot \mu = \mu \cdot \mathbf{T}^2\, f$$

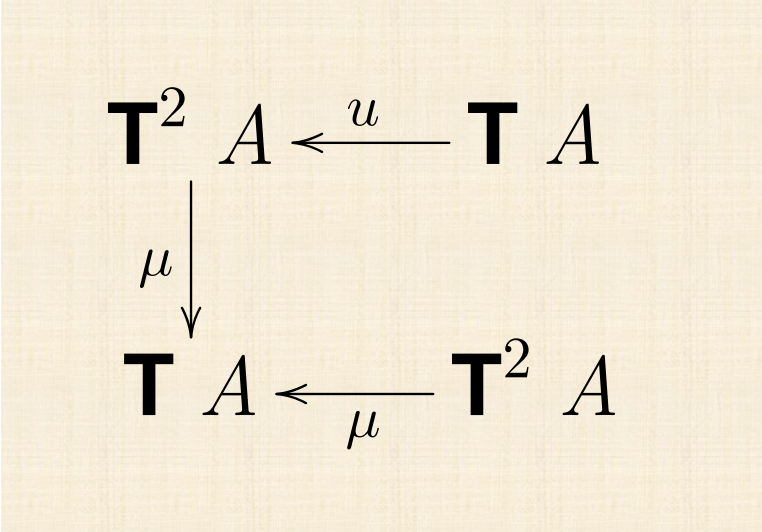
Monad – multiplication…



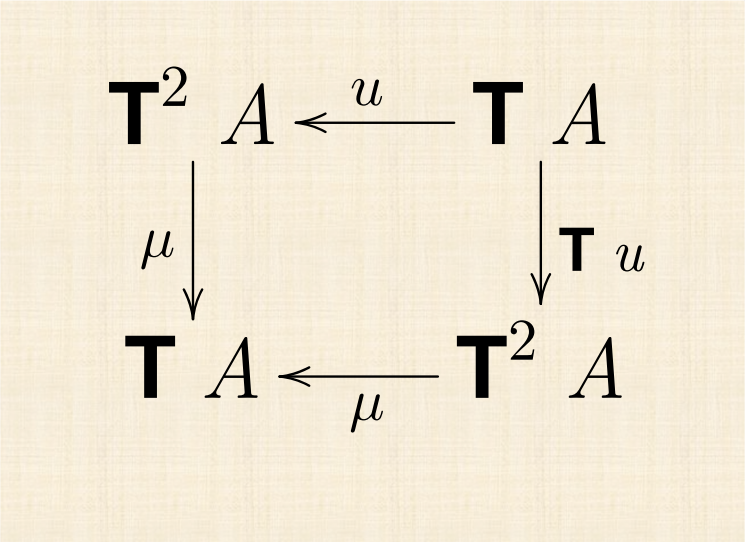
Monad – multiplication *versus* unit



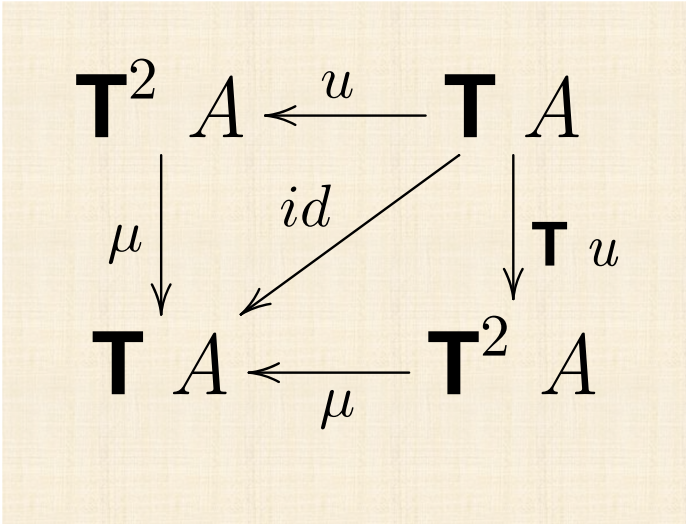
Monad – multiplication *versus* unit



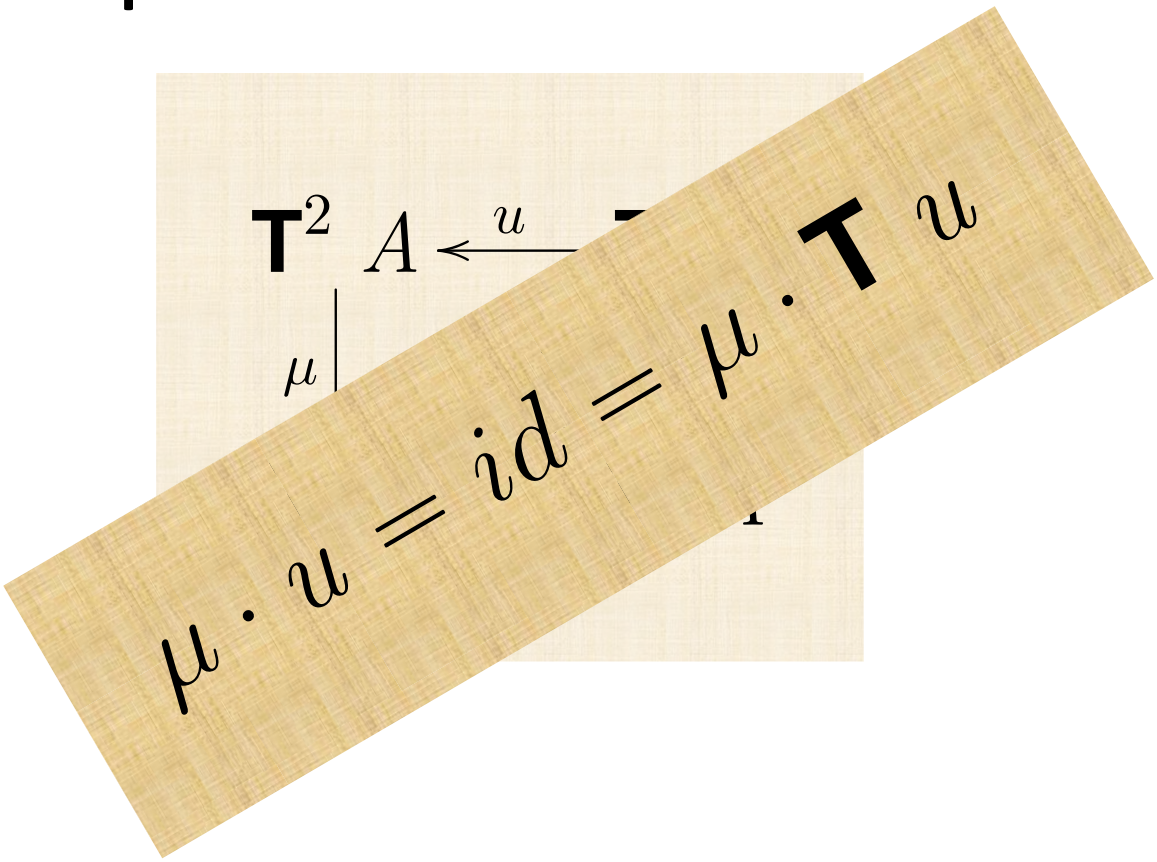
Monad – multiplication *versus* unit



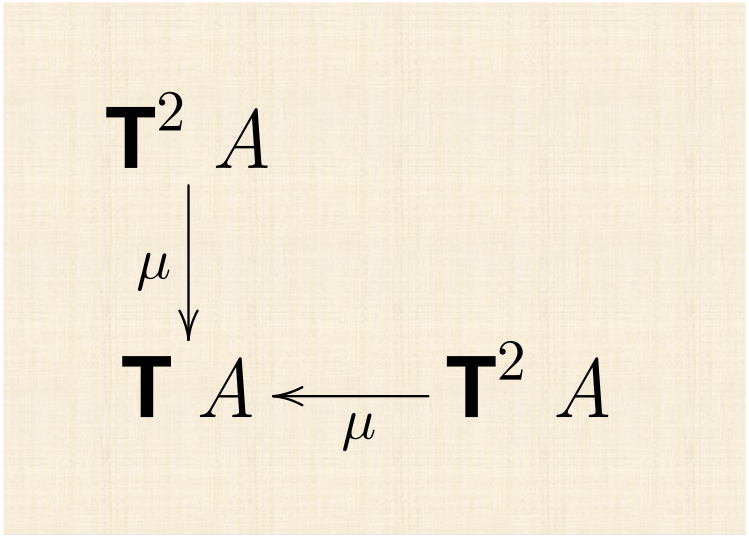
Monad – multiplication *versus* unit



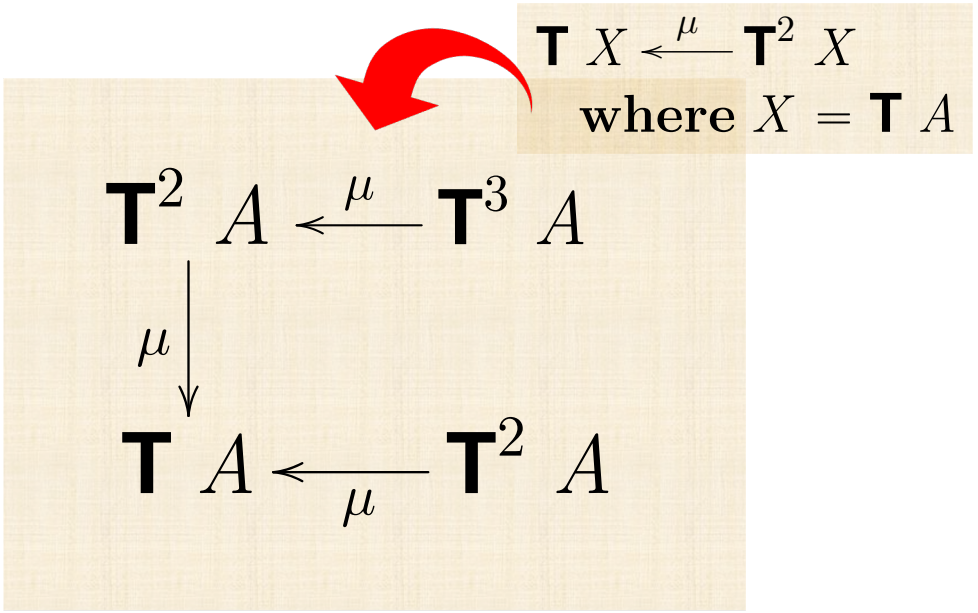
Monad – multiplication *versus* unit



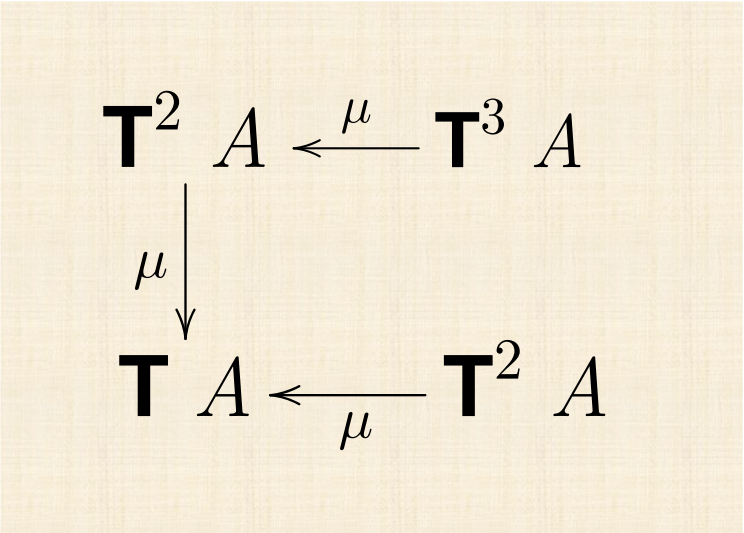
Monad – multiplication *versus* multiplication



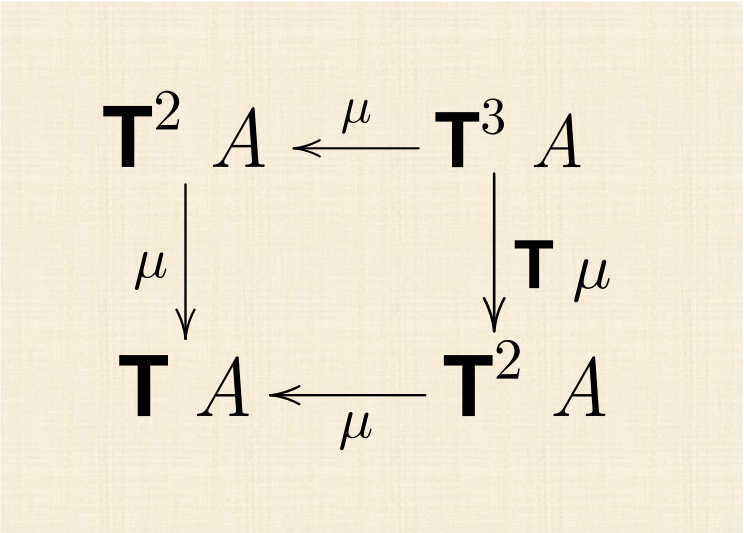
Monad – multiplication *versus* multiplication



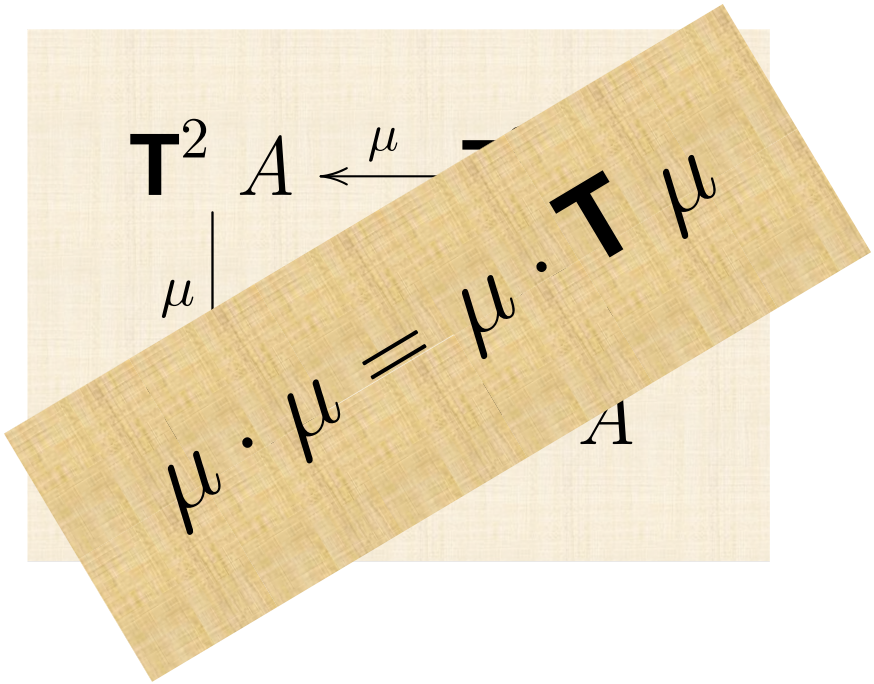
Monad – multiplication *versus* multiplication



Monad – multiplication *versus* multiplication



Monad – multiplication *versus* multiplication



Monad – laws of unit and multiplication

$$\mu \cdot u = id = \mu \cdot \mathbf{T}\, u$$

$$\mu \cdot \mu = \mu \cdot \mathbf{T}\, \mu$$

$$X \xrightarrow{u} \mathbf{T}\, X \xleftarrow{\mu} \mathbf{T}\, (\mathbf{T}\, X)$$

Monad – unit of composition

$$
\begin{aligned}
& f \bullet u = f = u \bullet f \\
\equiv & \quad \{ \text{ definition of } f \bullet g, \text{ twice } \} \\
& \mu \cdot \mathbf{T}\, f \cdot u = f = \mu \cdot \mathbf{T}\, u \cdot f \\
\equiv & \quad \{ \text{ natural-}u \text{ and } \mu \cdot \mathbf{T}\, u = id \ \} \\
& \mu \cdot u \cdot f = f = id \cdot f \\
\equiv & \quad \{ \ \mu \cdot u = id \ \} \\
& f = f
\end{aligned}
$$

$$\mathbf{T}\, f \cdot u = u \cdot f$$
$$\mathbf{T}\, f \cdot \mu = \mu \cdot \mathbf{T}^2\, f$$

$$\mu \cdot u = id = \mu \cdot \mathbf{T}\, u$$

Monad – unit of composition

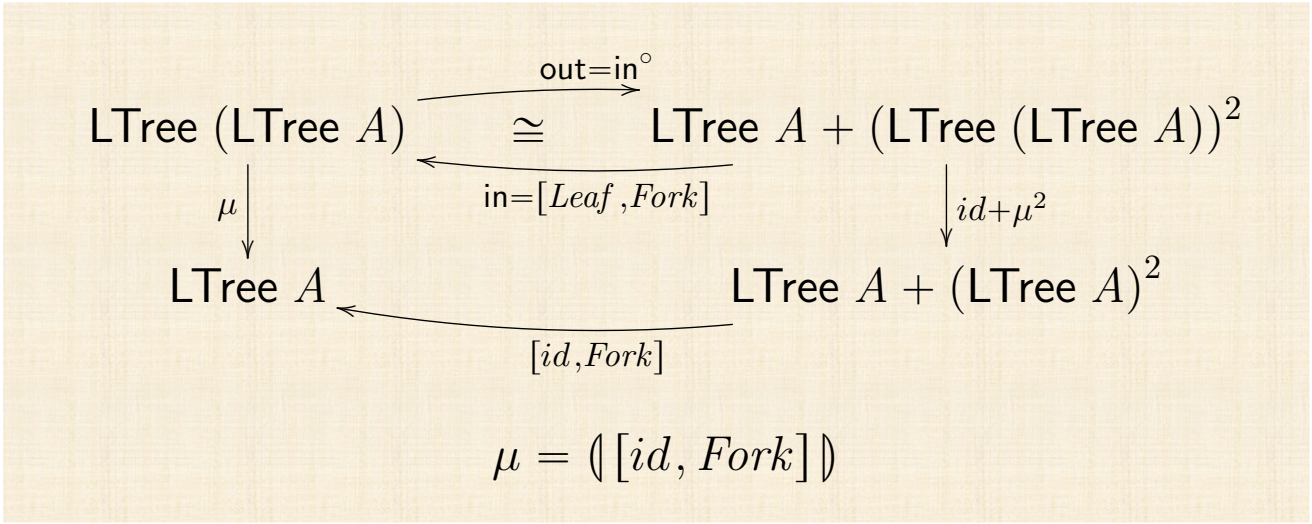
Multiplicação	$\mu \cdot \mu = \mu \cdot \mathsf{T}\,\mu$	(62)
Unidade	$\mu \cdot u = \mu \cdot \mathsf{T}\,u = id$	(63)
Natural-u	$u \cdot f \quad = \quad \mathsf{T}\,f \cdot u$	(64)
Natural-μ	$\mu \cdot \mathsf{T}\,(\mathsf{T}\,f) \quad = \quad \mathsf{T}\,f \cdot \mu$	(65)

Monad – laws

Multiplicação	$\mu \cdot \mu = \mu \cdot \mathsf{T}\,\mu$	(62)
Unidade	$\mu \cdot u = \mu \cdot \mathsf{T}\,u = id$	(63)
Natural-u	$u \cdot f \;=\; \mathsf{T}\,f \cdot u$	(64)
Natural-μ	$\mu \cdot \mathsf{T}\,(\mathsf{T}\,f) \;=\; \mathsf{T}\,f \cdot \mu$	(65)

Composição monádica	$f \bullet g \;=\; \mu \cdot \mathsf{T}\,f \cdot g$	(66)
Associatividade-$\bullet$	$f \bullet (g \bullet h) = (f \bullet g) \bullet h$	(67)
Identidade-$\bullet$	$u \bullet f = f = f \bullet u$	(68)
Associatividade-$\bullet/\cdot$	$(f \bullet g) \cdot h \;=\; f \bullet (g \cdot h)$	(69)
Associatividade-$\cdot/\bullet$	$(f \cdot g) \bullet h \;=\; f \bullet (\mathsf{T}\,g \cdot h)$	(70)
μ versus $\bullet$	$id \bullet id \;=\; \mu$	(71)

LTree monad




```haskell
data LTree a = Leaf a | Fork (LTree a, LTree a)
```

```haskell
mu :: LTree (LTree a) -> LTree a
mu = cataLTree (either id Fork)
```

LTree monad

$$X \xrightarrow{Leaf} \mathsf{LTree}\ X \xleftarrow{(\!|[id, Fork]|\!)} \mathsf{LTree}^2\ X$$

$$\mu \cdot u = id = \mu \cdot \mathsf{T}\ u$$

$$\mu \cdot \mu = \mu \cdot \mathsf{T}\ \mu$$

```
data LTree a = Leaf a | Fork (LTree a, LTree a)
```

```
mu  :: LTree (LTree a) -> LTree a
mu  =  cataLTree (either id Fork)
```

LTree monad

$$
\begin{array}{ll}
& \mu \cdot \mathsf{LTree}\ Leaf = id \\
\equiv & \quad \{\ \text{definition of } \mu\ \} \\
& (\!|\, [id, Fork] \,|\!) \cdot \mathsf{LTree}\ Leaf = id \\
\equiv & \quad \{\ \text{cata-absorption}\ \} \\
& (\!|\, [id, Fork] \cdot (Leaf + id) \,|\!) = id \\
\equiv & \quad \{\ +\text{-absorption}\ \} \\
& (\!|\, [Leaf\,, Fork] \,|\!) = id \\
\equiv & \quad \{\ \text{cata-reflection}\ \} \\
& true
\end{array}
$$

$$u = Leaf$$

$$\mu \cdot u = id = \mu \cdot \mathsf{T}\ u$$

$$X \xrightarrow{Leaf} \mathsf{LTree}\ X \xleftarrow{(\!|\, [id, Fork] \,|\!)} \mathsf{LTree}^2\ X$$

```
data LTree a = Leaf a | Fork (LTree a, LTree a)
```

```
mu :: LTree (LTree a) -> LTree a
mu = cataLTree (either id Fork)
```

Monad “binding”

$$A \xrightarrow{u} \mathbf{T}\ A \xleftarrow{\mu} \mathbf{T}^2\ A$$

$$f \bullet u = f$$

Monad "binding"

$$A \xrightarrow{u} \mathbf{T}\ A \xleftarrow{\mu} \mathbf{T}^2\ A$$

$$f \bullet id = ?$$

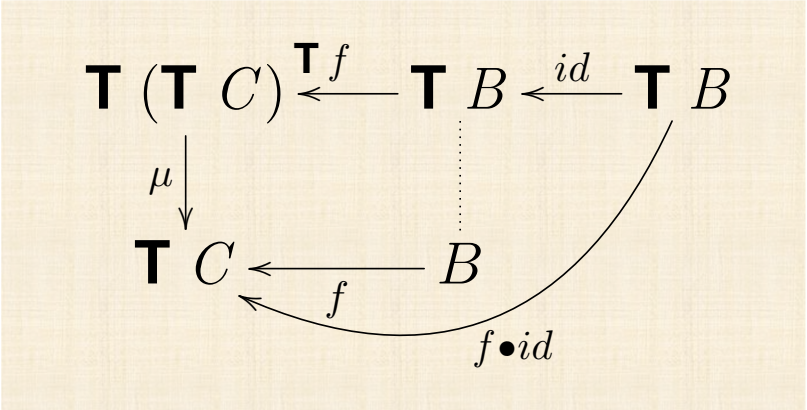
$$f \bullet u = f$$

Monad "binding"

$$A \xrightarrow{u} \mathbf{T}\ A \xleftarrow{\mu} \mathbf{T}^2\ A$$

$$f \bullet id = ?$$

$$f \bullet u = f$$

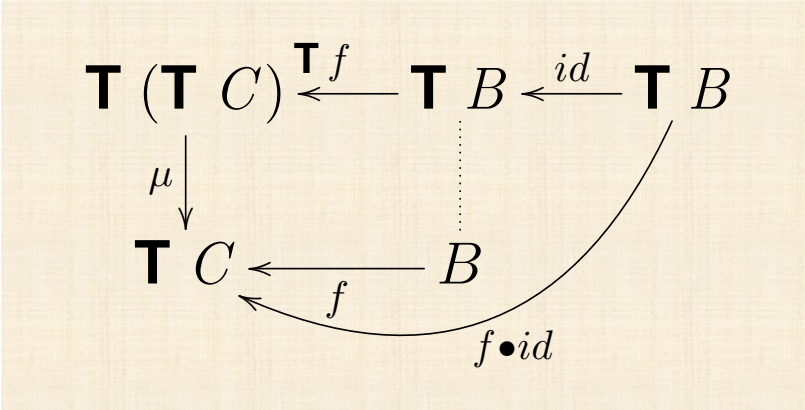


Monad "binding"

$$A \xrightarrow{u} \mathbf{T}\ A \xleftarrow{\mu} \mathbf{T}^2\ A$$

$$f \bullet id = \mu \cdot \mathbf{T}\ f$$

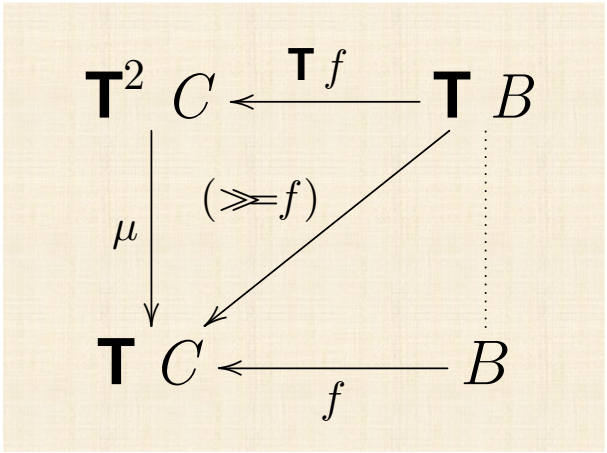
$$f \bullet u = f$$



Monad "binding"

$$A \xrightarrow{u} \mathbf{T}\ A \xleftarrow{\mu} \mathbf{T}^2\ A$$

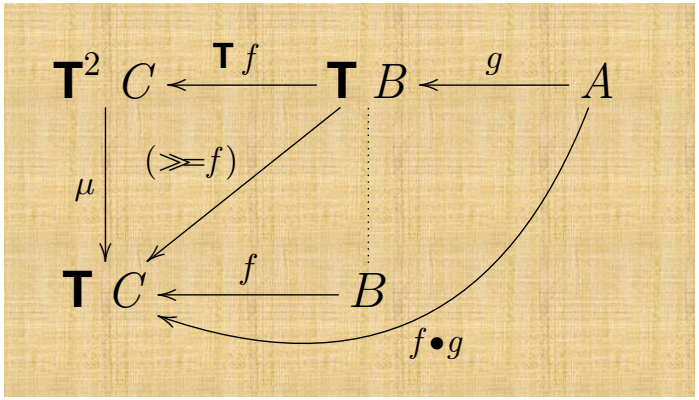
$$(\gg\!= f) = f \bullet id$$



$$x \gg\!= f \stackrel{\text{def}}{=} (\mu \cdot \mathbf{T}\ f)x$$

Monad "binding"

$$A \xrightarrow{u} \mathbf{T}\ A \xleftarrow{\mu} \mathbf{T}^2\ A$$



$$(\gg\!= f) = f \bullet id$$

$$(f \bullet g)\ x = (g\ x) \gg\!= f$$

Monad (Haskell)

```
-- (5) Monad ------------------------

instance Monad LTree where
    return  = Leaf
    t >>= g = (mu . fmap g) t

mu  :: LTree (LTree a) -> LTree a
mu  =  cataLTree (either id Fork)
```

Minimal complete definition

```
(>>=)
```

Methods

```
(>>=) :: forall a b. m a -> (a -> m b) -> m b   | infixl 1 | # Source
```

Sequentially compose two actions, passing any value produced by the first as an argument to the second.

```
(>>) :: forall a b. m a -> m b -> m b   | infixl 1 | # Source
```

Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.

```
return :: a -> m a   # Sourc
```

return $= u$

Monad (Haskell)

```
-- (7) Monads: ------------------------------------------------

-- (7.1) Kleisli monadic composition ------------------------

infix 4  .!

(.!) :: Monad a => (b -> a c) -> (d -> a b) -> d -> a c
(f .! g) a = (g a) >>= f

mult :: (Monad m) => m (m b) -> m b
mult = (>>= id)  -- also known as join
```

Minimal complete definition

`(>>=)`

Methods

(>>=) :: forall a b. m a -> (a -> m b) -> m b | infixl 1 | # Source

Sequentially compose two actions, passing any value produced by the first as an argument to the second.

$$(f \bullet g)\ x = (g\ x) \gg\!= f$$

(>>) :: forall a b. m a -> m b -> m b | infixl 1 | # Source

Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.

$$id \bullet id \quad = \quad \mu$$

return :: a -> m a # Source

$$\text{return} = u$$

Monad (Haskell)

```
-- (5) Monad ----------------------

instance Monad LTree where
    return  = Leaf
    t >>= g = (mu . fmap g) t

mu  :: LTree (LTree a) -> LTree a
mu  =  cataLTree (either id Fork)
```

Minimal complete definition

```
(>>=)
```

Methods

```
(>>=) :: forall a b. m a -> (a -> m b) -> m b    infixl 1    # Source
```

Sequentially compose two actions, passing any value produced by the first as an argument to the second.

```
(>>) :: forall a b. m a -> m b -> m b    infixl 1    # Source
```

Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.

```
return :: a -> m a    # Sourc
```

return $= u$

```haskell
---- Using the pairing monad (cf. teachers versus classes) ----

data P a = P(a,a) deriving (Show,Eq,Ord)

instance Monad P where
   x >>= f = (muP.fmap f)  x
   return a = P(a,a)

muP :: P (P a) -> P a
muP(P(P(a,b),P(c,d))) = P(a,d)

instance Functor P where
    fmap f (P(a,b)) = P(f a, f b)

instance Applicative P where
    pure = return
    (<*>) = aap
```

Identity monad

$$\mathbf{T}\ X = X$$
$$\mathbf{T}\ f = f$$

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}\ (\mathbf{T}\ X)$$

$$X \xrightarrow{u} X \xleftarrow{\mu} X$$

$$u = id$$
$$\mu = id$$
$$f \bullet g = \mu \cdot \mathbf{T}\ f \cdot g = id \cdot f \cdot g = f \cdot g$$

Monads – summary

$$\mu \cdot u = id = \mu \cdot \mathbf{T}\ u$$

$$f \bullet g = \mu \cdot \mathbf{T}\ f \cdot g$$

$$\mu \cdot \mu = \mu \cdot \mathbf{T}\ \mu$$

$$(\gg\!= f) = f \bullet id$$

MONADS: do-notation

$$(f \cdot g)\ a = f\ (g\ a) = \mathbf{let}\ b = g\ a\ \mathbf{in}\ f\ b$$

MONADS: do-notation

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}\ (\mathbf{T}\ X)$$

$$(f \cdot g)\ a = f\ (g\ a) = \mathbf{let}\ b = g\ a\ \mathbf{in}\ f\ b$$

$$(f \bullet g)\ a = \mathbf{do}\ \{b \leftarrow g\ a; f\ b\}$$

MONADS: do-notation

$$X \xrightarrow{u} \mathbf{T}\,X \xleftarrow{\mu} \mathbf{T}\,(\mathbf{T}\,X)$$

$$(f \cdot g)\ a = f\ (g\ a) = \mathbf{let}\ b = g\ a\ \mathbf{in}\ f\ b$$

$$(f \bullet g)\ a = \mathbf{do}\ \{ b \leftarrow g\ a;\ f\ b \}$$

MONADS: do-notation

$$(\gg\!= f) = f \bullet id$$

$$
\begin{aligned}
& x \gg\!= f \\
= \quad & \{ \text{ definition of } (\gg\!= f) \} \\
& (f \bullet id)\ x \\
= \quad & \{ \ \mathbf{do}\text{-notation} \ \} \\
& \mathbf{do}\ \{ b \leftarrow x; f\ b \}
\end{aligned}
$$

$$(f \bullet g)\ a = \mathbf{do}\ \{ b \leftarrow g\ a; f\ b \}$$

MONADS: do-notation

$$(\gg\!= f) = f \bullet id$$

$$\begin{aligned} & x \gg\!= f \\ = & \quad \{ \text{ definition of } (\gg\!= \\ & (f \bullet id)\ x \\ = & \quad \{ \\ & \mathbf{do}\ \{ \end{aligned}$$

$$x \gg\!= f = \mathbf{do}\ \{b \leftarrow x; f\ b\}$$

$$(f \bullet g)\ a = \mathbf{do}\ \{b \leftarrow g\ a; f\ b\}$$

MONADS: do-notation

$$(\gg\!= f) = f \bullet id$$

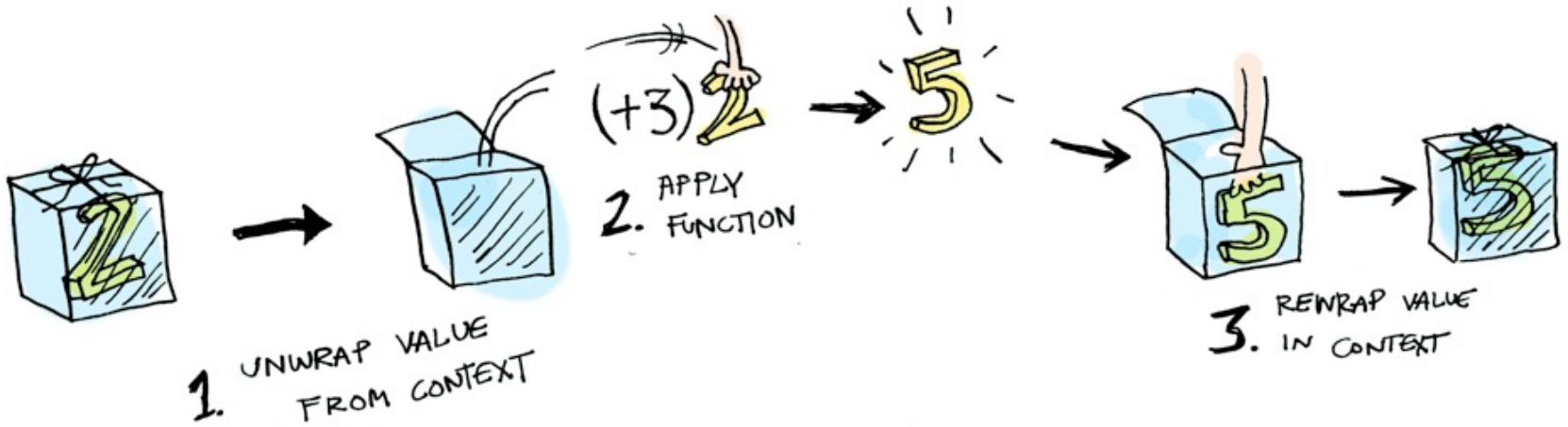
$$x \gg\!= f$$
$$= \quad \{ \text{ definition of } (\gg\!= \}$$
$$(f \bullet id)\ x$$
$$= \quad \{$$
$$\mathbf{do}\ \{$$

$$x \gg\!= f = \mathbf{do}\ \{b \leftarrow x;\ f\ b\}$$

$$(f \bullet g)\ a = \mathbf{do}\ \{b \leftarrow g\ a; f\ b\}$$

MONADS: do-notation

$$x \gg\!= f = \mathbf{do}\ \{ b \leftarrow x; f\ b \}$$



(**Credits**: `http://shorturl.at/buNPX`)

MONADS: do-notation

Recall law

$$(g \cdot f) \bullet h = g \bullet (\mathbf{T}\, f \cdot h)$$

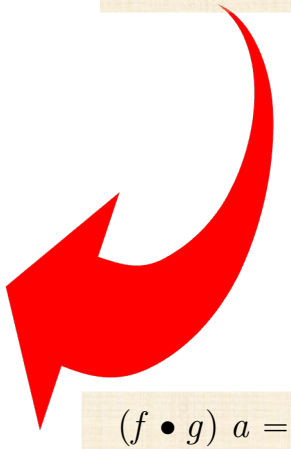
Particular case : $g, h := u, id$:

MONADS: do-notation

$$(g \cdot f) \bullet h = g \bullet (\mathbf{T}\ f \cdot h)$$

$$
\begin{aligned}
& (u \cdot f) \bullet id = u \bullet (\mathbf{T}\ f \cdot id) \\
\equiv & \quad \{\ \text{natural-}id\text{; unit } u\ \} \\
& (u \cdot f) \bullet id = \mathbf{T}\ f \\
\equiv & \quad \{\ \text{go pointwise on } x\ \} \\
& ((u \cdot f) \bullet id)\ x = \mathbf{T}\ f\ x \\
\equiv & \quad \{\ \text{introduce } \mathbf{do}\text{-notation}\ \} \\
& \mathbf{T}\ f\ x = \mathbf{do}\ \{b \leftarrow x; \text{return } (f\ b)\}
\end{aligned}
$$

Particular case : $g, h := u, id$:



$$(f \bullet g)\ a = \mathbf{do}\ \{b \leftarrow g\ a; f\ b\}$$

MONADS: do-notation

$$(g \cdot \quad) \qquad (\mathbf{T}\ f \cdot h)$$

$$(u \cdot f) \bullet id = u \bullet (\mathbf{T}\ f \cdot id)$$

$\equiv$ { natural-id; unit u }

$$(u \cdot f) \bullet id = \mathbf{T}\ f$$

$\equiv$ { go pointwise o

$$((u \cdot f) \bullet id)$$

$\equiv$ { tation }

$$\mathbf{T}\ f\ x = \{\ b \leftarrow x; \text{return}\ (f\ b)\ \}$$

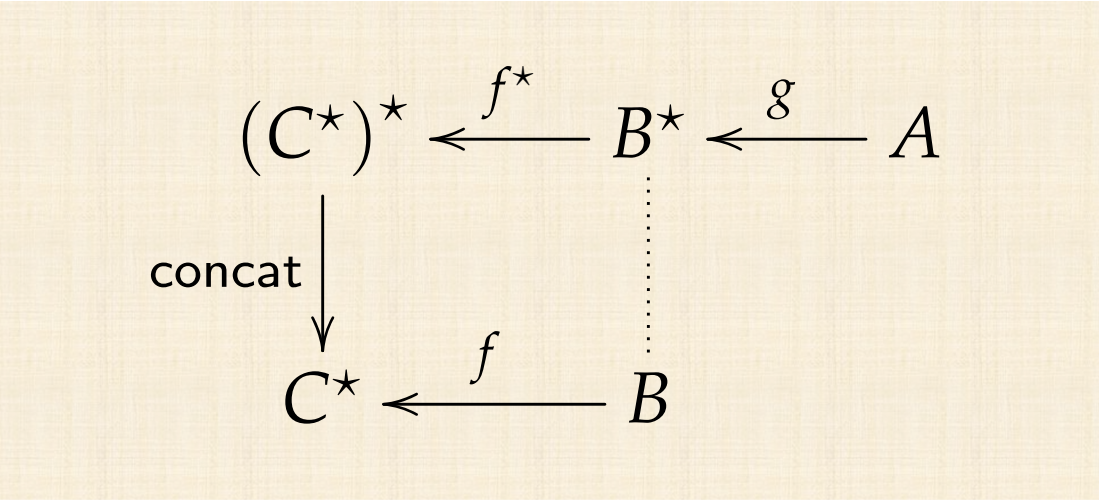
$$\mathbf{T}\ f\ x = \mathbf{do}\ \{b \leftarrow x; \text{return}\ (f\ b)\}$$

icular case : $g, h := u, id$:

$$(f \bullet g)\ a = \mathbf{do}\ \{b \leftarrow g\ a; f\ b\}$$

MONADS: list compreensions

$$[\, e \mid a_1 \leftarrow x_1, \ldots, a_n \leftarrow x_n \,] = \mathbf{do}\, \{a_1 \leftarrow x_1; \ldots; a_n \leftarrow x_n; \text{return } e\}$$



Monad – more laws

$$(f \bullet g)\ a = \mathbf{do}\ \{b \leftarrow g\ a; f\ b\} \tag{87}$$

$$x \gg\!= f \quad = \quad (\mu \cdot \mathsf{T}\ f)x \tag{88}$$

$$\mathbf{do}\ \{x \leftarrow a; b\} \quad = \quad a \gg\!= (\lambda x \rightarrow b) \tag{89}$$

$$\mu\ x \quad = \quad x \gg\!= id \tag{90}$$

$$x \gg y \quad = \quad x \gg\!= \underline{y} \tag{91}$$

I/O monad : interfacing with the file system

I/O monad : interfacing with the file system

$$\text{IO } String \xleftarrow{readFile} FilePath$$

$$\text{IO } 1 \xleftarrow{writeFile\ o} String$$

$$copy\ i\ o = (writeFile\ o \bullet readFile)\ i$$

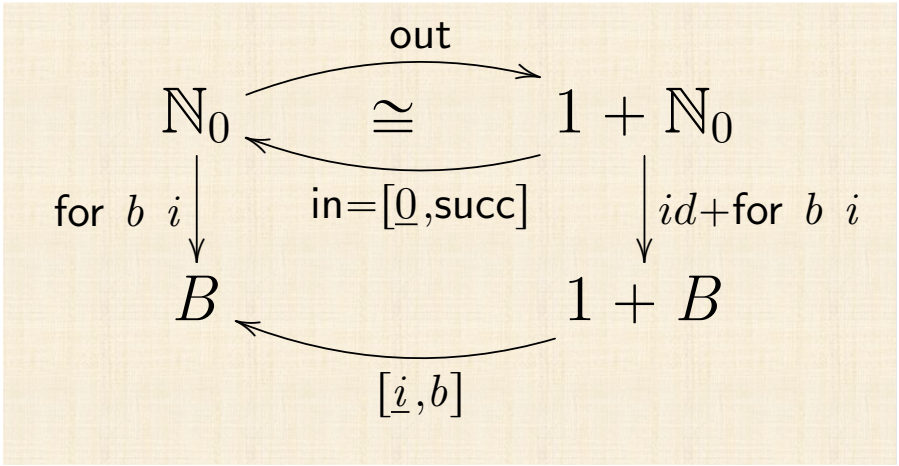
$$\equiv \quad \{\ \textbf{do-notation}\ \}$$

$$copy\ i\ o = \textbf{do}\ \{s \leftarrow readFile\ i; writeFile\ o\ s\}$$


O monad (trivial) da identidade.

Monadic recursion

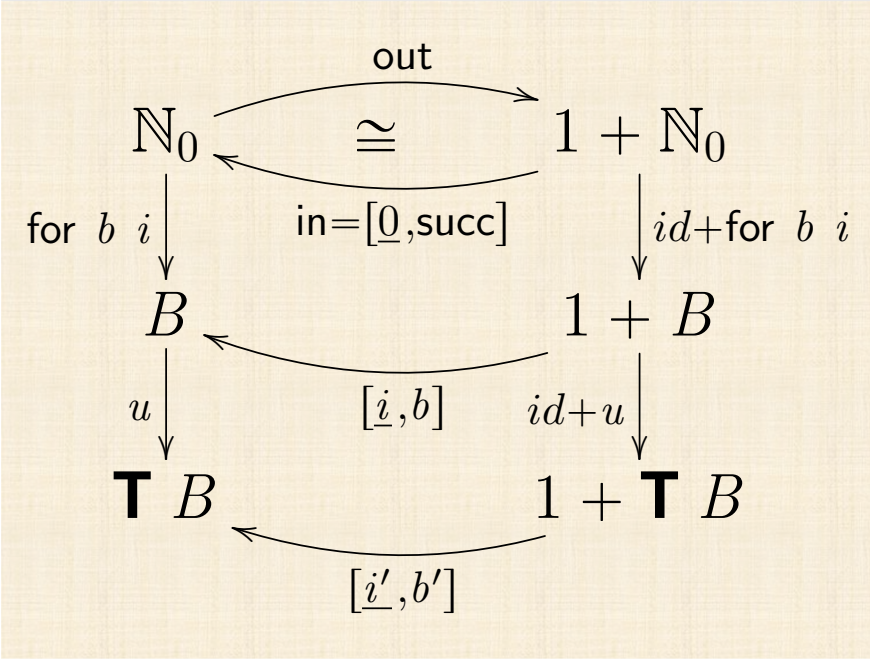
$$X \xrightarrow{u} \mathsf{T}\,X \xleftarrow{\mu} \mathsf{T}\,(\mathsf{T}\,X)$$



$$\mathsf{for}\ b\ i = (\!|\, [\underline{i}, b] \,|\!)$$

Monadic recursion

$$X \xrightarrow{u} \mathbf{T}\,X \xleftarrow{\mu} \mathbf{T}\,(\mathbf{T}\,X)$$



for b i $= (\!|\, [\underline{i}, b] \,|\!)$

Monadic recursion

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}\ (\mathbf{T}\ X)$$

$$
\begin{array}{ll}
& u \cdot \mathsf{for}\ b\ i = \mathsf{for}\ b'\ i' \\
\equiv & \quad \{\ \text{for } b\ i = (\!|\ [\underline{i}, b]\ |\!)\ \} \\
& u \cdot (\!|\ [\underline{i}, b]\ |\!) = (\!|\ [\underline{i'}, b']\ |\!) \\
\Leftarrow & \quad \{\ \text{fusão-cata}\ \} \\
& u \cdot [\underline{i}, b] = [\underline{i'}, b'] \cdot (id + u) \\
\equiv & \quad \{\ \text{coprodutos (fusão, absorção, eq)}\ \} \\
& \begin{cases} u \cdot \underline{i} = \underline{i'} \\ u \cdot b = b' \cdot u \end{cases}
\end{array}
$$

Monadic recursion

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}\ (\mathbf{T}\ X)$$

$$
\begin{array}{ll}
& u \cdot \mathsf{for}\ b\ i = \mathsf{for}\ b'\ i' \\
\equiv & \quad \{\ \text{for } b\ i = (\!|\ [\underline{i}, b]\ |\!)\ \} \\
& u \cdot (\!|\ [\underline{i}, b]\ |\!) = (\!|\ [\underline{i'}, b']\ |\!) \\
\Leftarrow & \quad \{\ \text{fusão-cata}\ \} \\
& u \cdot [\underline{i}, b] = [\underline{i'}, b'] \cdot (id + u) \\
\equiv & \quad \{\ \text{coprodutos (fusão, absorção, eq)}\ \} \\
& \begin{cases} u \cdot \underline{i} = \underline{i'} \\ u \cdot b = b' \cdot u \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
\equiv & \quad \{\ f \cdot \underline{x} = \underline{f\ x}\ ;\ \mathbf{T}\ f \cdot u = u \cdot f\ \} \\
& \begin{cases} u\ i = i' \\ \mathbf{T}\ b \cdot u = b' \cdot u \end{cases} \\
\Leftarrow & \quad \{\ \text{trivial}\ \} \\
& \begin{cases} i' = u\ i \\ b' = \mathbf{T}\ b \end{cases}
\end{array}
$$

$$\mathsf{mfor}\ b\ i = (\!|\ [\underline{u\ i}, \mathbf{T}\ b]\ |\!)$$

Monadic recursion

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}\ (\mathbf{T}\ X)$$

$$\mathsf{mfor}\ b\ i = (\!|\ [\underline{u\ i}, \mathbf{T}\ b]\ |\!)$$

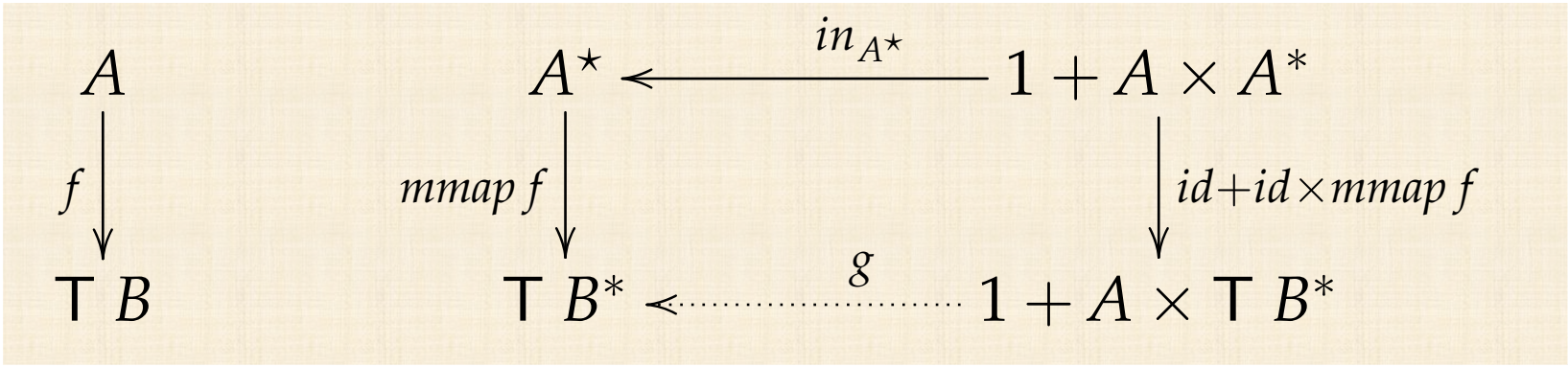
$$\begin{array}{ll}
 & \mathsf{mfor}\ b\ i = (\!|\ [\underline{u\ i}, \mathbf{T}\ b]\ |\!) \\
\equiv & \quad \{\ \text{universal cata}\ \} \\
 & \mathsf{mfor}\ b\ i \cdot [\underline{0}, \mathsf{succ}] = [\underline{u\ i}, \mathbf{T}\ b] \cdot (id + \mathsf{mfor}\ b\ i) \\
\equiv & \quad \{\ \text{coprodutos (fusão, absorção, eq) ; variáveis}\ \} \\
 & \begin{cases} \mathsf{mfor}\ b\ i\ 0 = u\ i \\ \mathsf{mfor}\ b\ i\ (n+1) = \mathbf{T}\ b\ (\mathsf{mfor}\ b\ i\ n) \end{cases} \\
\equiv & \quad \{\ u = \mathsf{return}\ ;\ \mathbf{T}\ f\ x = \mathbf{do}\ \{a \leftarrow x; \mathsf{return}\ (f\ a)\}\ \} \\
 & \begin{cases} \mathsf{mfor}\ b\ i\ 0 = \mathsf{return}\ i \\ \mathsf{mfor}\ b\ i\ (n+1) = \mathbf{do}\ \{x \leftarrow \mathsf{mfor}\ b\ i\ n; \mathsf{return}\ (b\ x)\} \end{cases}
\end{array}$$

Monadic recursion

$$X \xrightarrow{u} \mathbf{T}\ X \xleftarrow{\mu} \mathbf{T}\ (\mathbf{T}\ X)$$

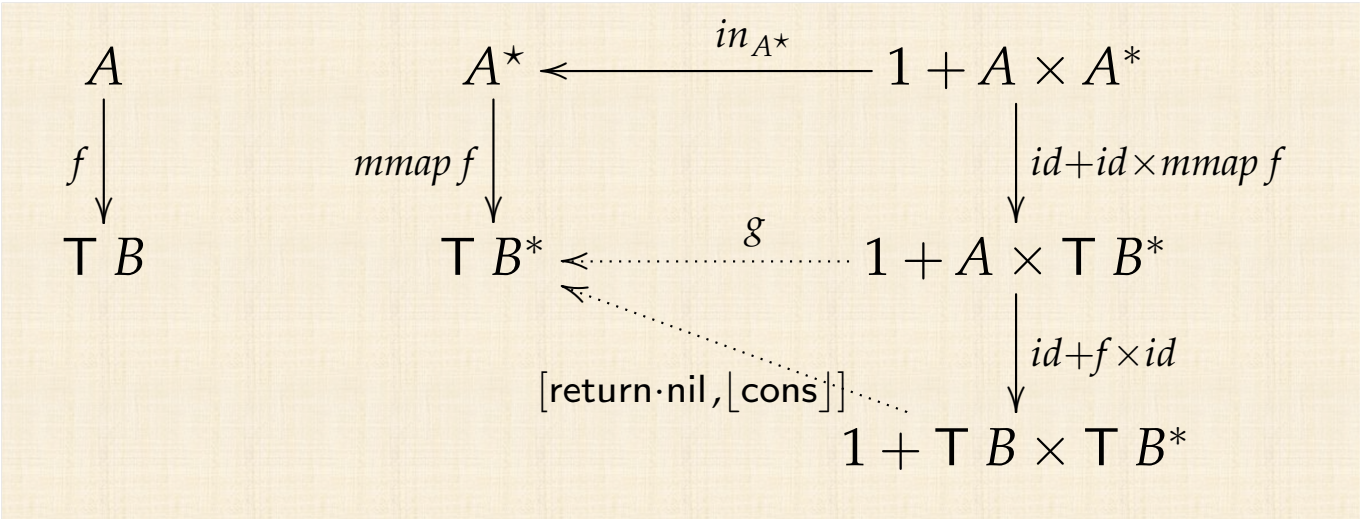
$$\begin{cases} mfor\ b\ i\ 0 = \mathsf{return}\ i \\ mfor\ b\ i\ (n+1) = \mathbf{do}\ \{x \leftarrow mfor\ b\ i\ n; \mathsf{return}\ (b\ x)\} \end{cases}$$

Monadic map (mmap)

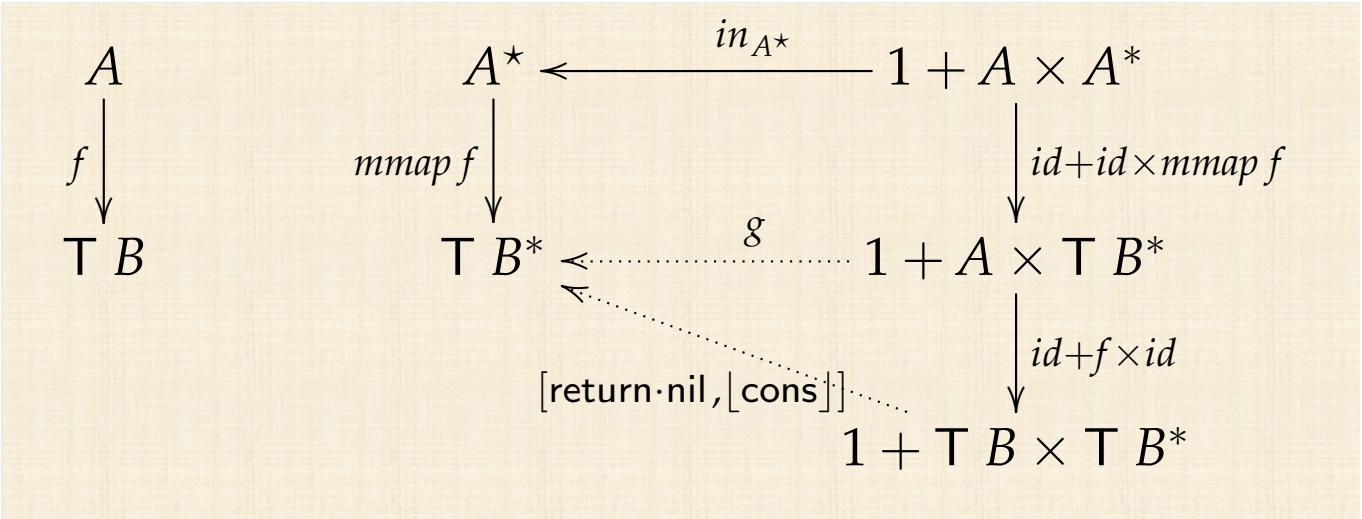


g?

Monadic map (mmap)



Monadic map (mmap)



$$\lfloor f \rfloor\ (x, y) = \mathbf{do}\ \{a \leftarrow x; b \leftarrow y; \mathsf{return}\ (f\ (a, b))\}$$

Recall…

$$\lfloor f \rfloor\ (x,y) = \mathbf{do}\ \{a \leftarrow x; b \leftarrow y; \text{return}\ (f\ (a,b))\}$$

```
*Main> lift f (x,y) = do { a <- x ; b <- y ; return (f (a,b)) }
*Main> d1 = uniform [1..6]
*Main> d2 = uniform [1..6]
*Main> lift add (d1, d2)
 7  16.7%
 6  13.9%
 8  13.9%
 5  11.1%
 9  11.1%
 4   8.3%
10   8.3%
 3   5.6%
11   5.6%
 2   2.8%
12   2.8%
```

Monadic map (mmap)

$$mmap\ f = (\!|\, [\mathsf{return} \cdot \mathsf{nil}, \lfloor \mathsf{cons} \rfloor \cdot (f \times id)] \,|\!)$$

$$\begin{aligned}
&mmap :: (Monad\ m) \Rightarrow (a \to m\ b) \to [a] \to m\ [b] \\
&mmap\ f\ [\,] = \mathsf{return}\ [\,] \\
&mmap\ f\ (h : t) = \mathbf{do}\ \{ b \leftarrow f\ h; x \leftarrow mmap\ f\ t; \mathsf{return}\ (b : x) \}
\end{aligned}$$

Monadic map (mmap)

Getting the minimum of a list (if possible…)

$$mgetmin :: Ord\ a \Rightarrow [a] \rightarrow Maybe\ a$$

Get the list of all minima (where possible…)

```
mmap mgetmin [[1,2],[3]] = Just [1,3]
mmap mgetmin [[1,2],[]] = Nothing
```

Monadic map (mmap)

Getting the minimum of a list (if possible…)

$$mgetmin :: Ord\ a \Rightarrow [a] \rightarrow Maybe\ a$$

$$\begin{aligned} mgetmin\ [\,] &= \text{Nothing} \\ mgetmin\ [a] &= \text{return}\ a \\ mgetmin\ (h:t) &= \mathbf{do}\ \{x \leftarrow mgetmin\ t; \text{return}\ (min\ h\ x)\} \end{aligned}$$

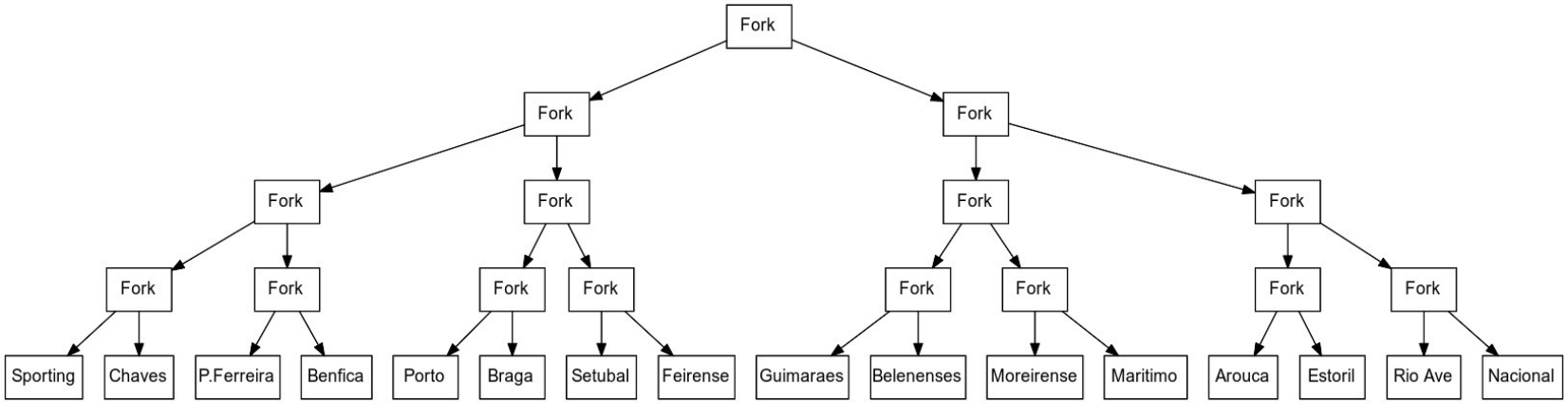
MONADS

$$X \xrightarrow{u} \quad \xleftarrow{\mu}$$


programming
productivity
work
giants
productivity
troubleshooters
hardware
methodologies
estimation
potpourri
project
staffs
india
tools
interval
waxed
iterative
management
promulgated

Probabilistic monadic programming

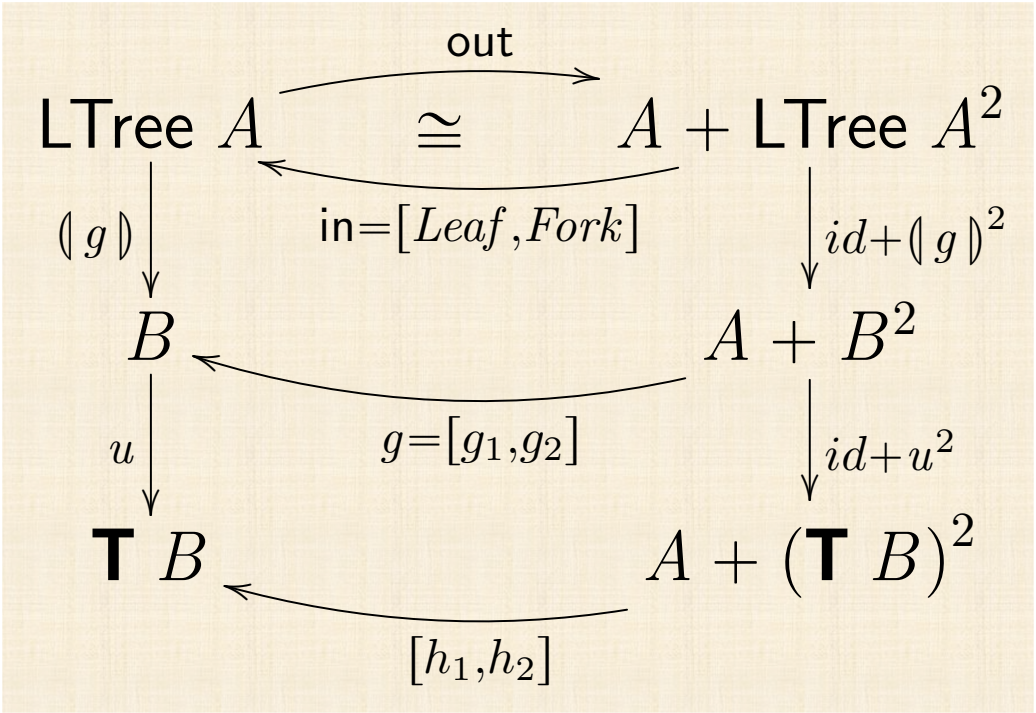
Example – football league



$Arouca$ — 28.6%
$Braga$ — 71.4%

etc

LTree ... catas with monads



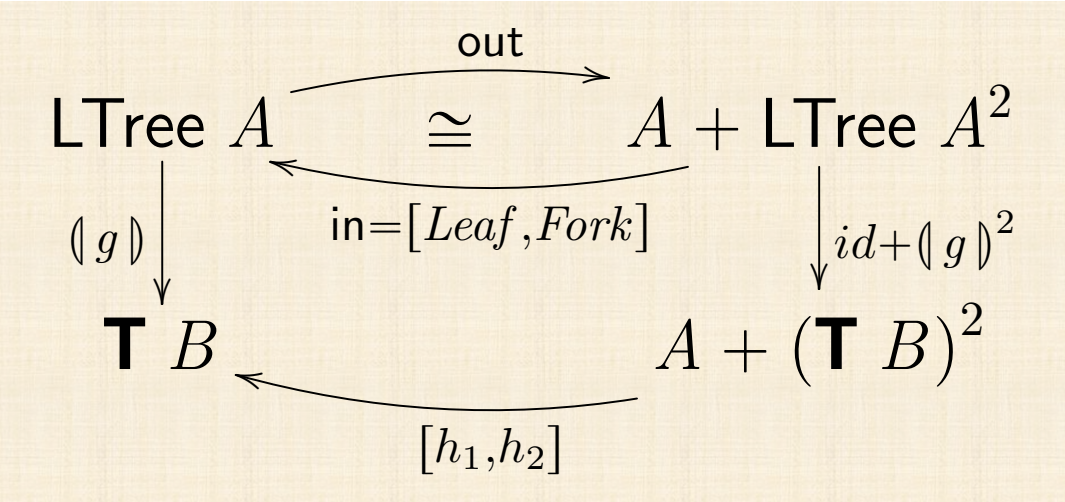
$$\begin{cases} h_1 = u \cdot g_1 \\ h_2 = \mathsf{T}\ g_2 \cdot \delta \end{cases}$$

$$\begin{aligned} &\delta\ (x, y) = \mathbf{do}\ \{ \\ &\quad a \leftarrow x; \\ &\quad b \leftarrow y; \\ &\quad \mathsf{return}\ (a, b) \\ &\quad \} \end{aligned}$$

LTree … catas with monads

$$
\begin{aligned}
&mcataLTtree\ g = k\ \mathbf{where}\\
&\quad k\ (Leaf\ \ a) = \mathsf{return}\ (g_1\ a)\\
&\quad k\ (Fork\ (x, y)) = \mathbf{do}\ \{\, a \leftarrow k\ x; b \leftarrow k\ y; \mathsf{return}\ (g_2\ (a, b))\,\}\\
&\quad g_1 = g \cdot i_1\\
&\quad g_2 = g \cdot i_2
\end{aligned}
$$

LTree ... monadic catas



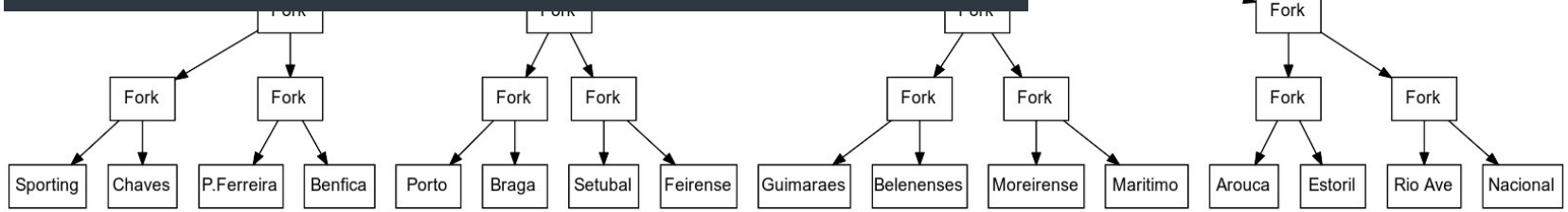
$$f : B^2 \to \mathsf{T}\ B$$

Arouca	28.6%
Braga	71.4%

$$h_2\ (x, y) = \mathbf{do}\ \{a \leftarrow x; b \leftarrow y; f\ (a, b)\}$$

Example – football league

```
jogo :: (Equipa, Equipa) -> Dist Equipa

*Main> jogo ("Braga","Arouca")
 "Braga"  71.4%
"Arouca"  28.6%
```

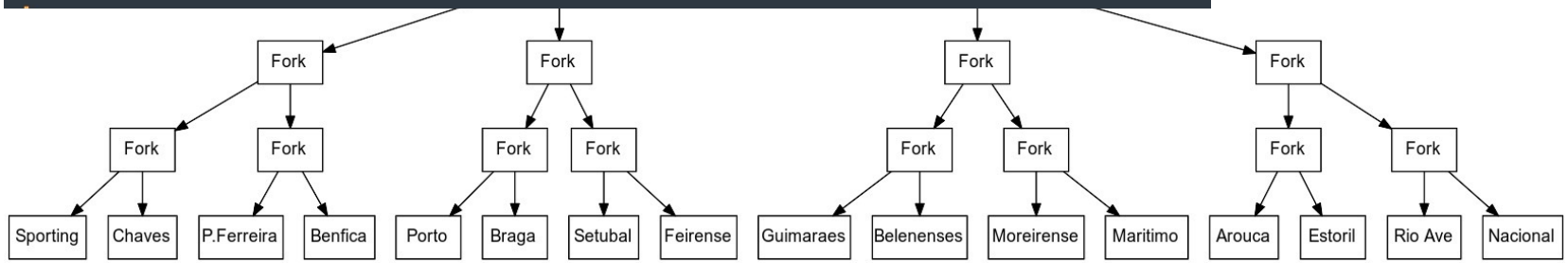


$Arouca$ ▬ 28.6%
$Braga$ ▬▬▬ 71.4%

etc

Example – football league

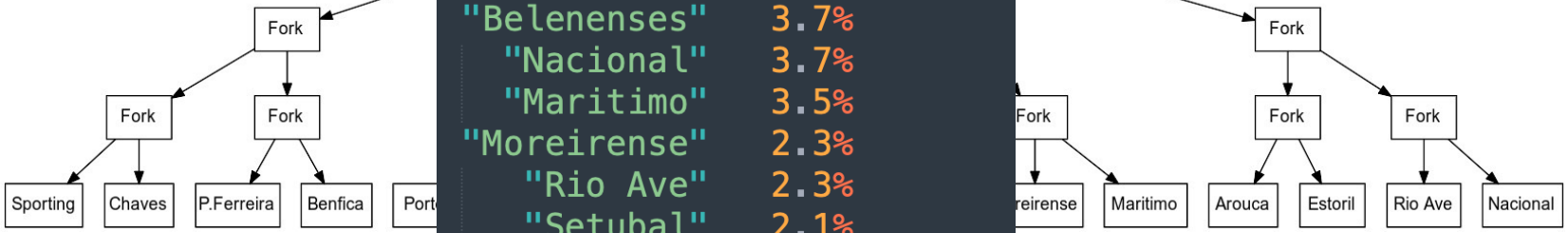
```
h2(d1,d2) = do { a <- d1; b <- d2; jogo(a,b) }

simular  = cataLTree (either return h2)
```




```
calendario = anaLTree lsplit equipas
```

Example – football league

```
*Main> simular calendario
     "Porto"  24.0%
   "Benfica"  22.0%
  "Sporting"  19.8%
     "Braga"   6.0%
 "Guimaraes"   4.5%
"Belenenses"   3.7%
  "Nacional"   3.7%
  "Maritimo"   3.5%
"Moreirense"   2.3%
   "Rio Ave"   2.3%
   "Setubal"   2.1%
"P.Ferreira"   1.8%
    "Arouca"   1.2%
    "Chaves"   1.2%
  "Feirense"   1.1%
   "Estoril"   0.8%
```



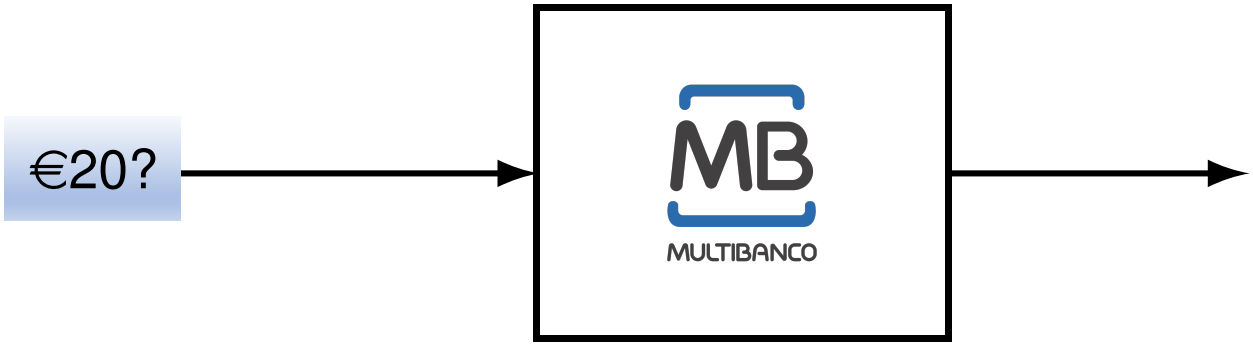
etc

Monad = “racing” functor

$$X \xrightarrow{u} \mathbf{T}\,X \xleftarrow{\mu} \mathbf{T}\,(\mathbf{T}\,X)$$

STATE MONAD (REACTIVE SYSTEMS)

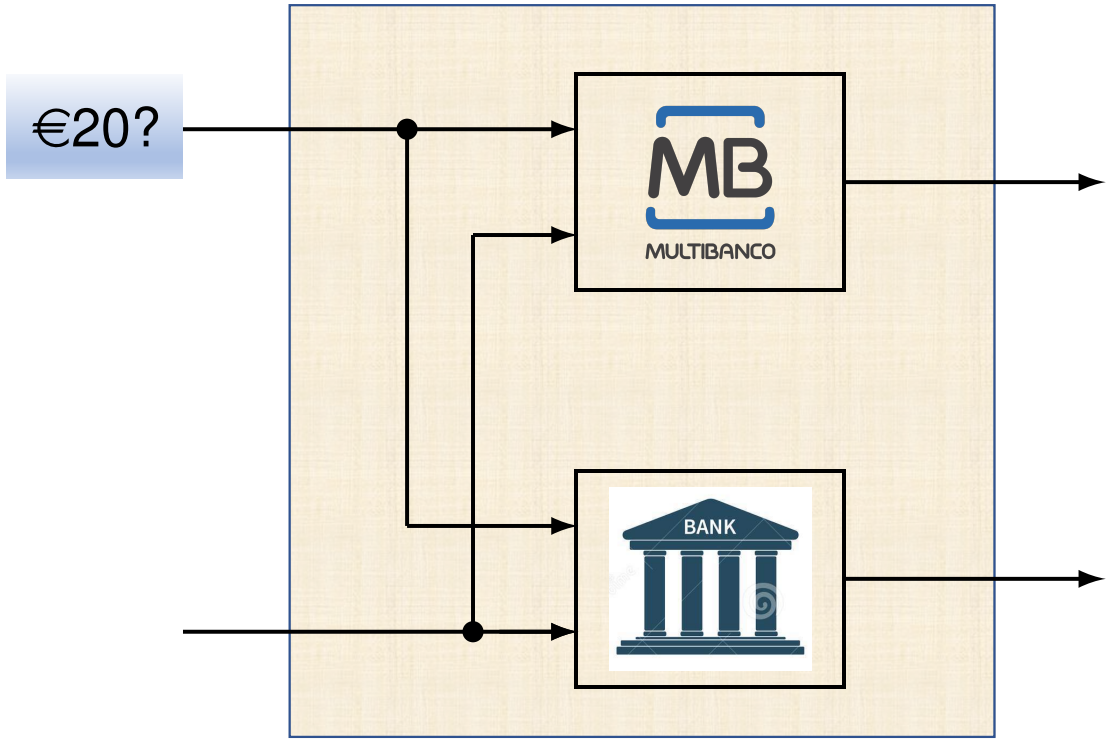
REACTIVE SYSTEMS



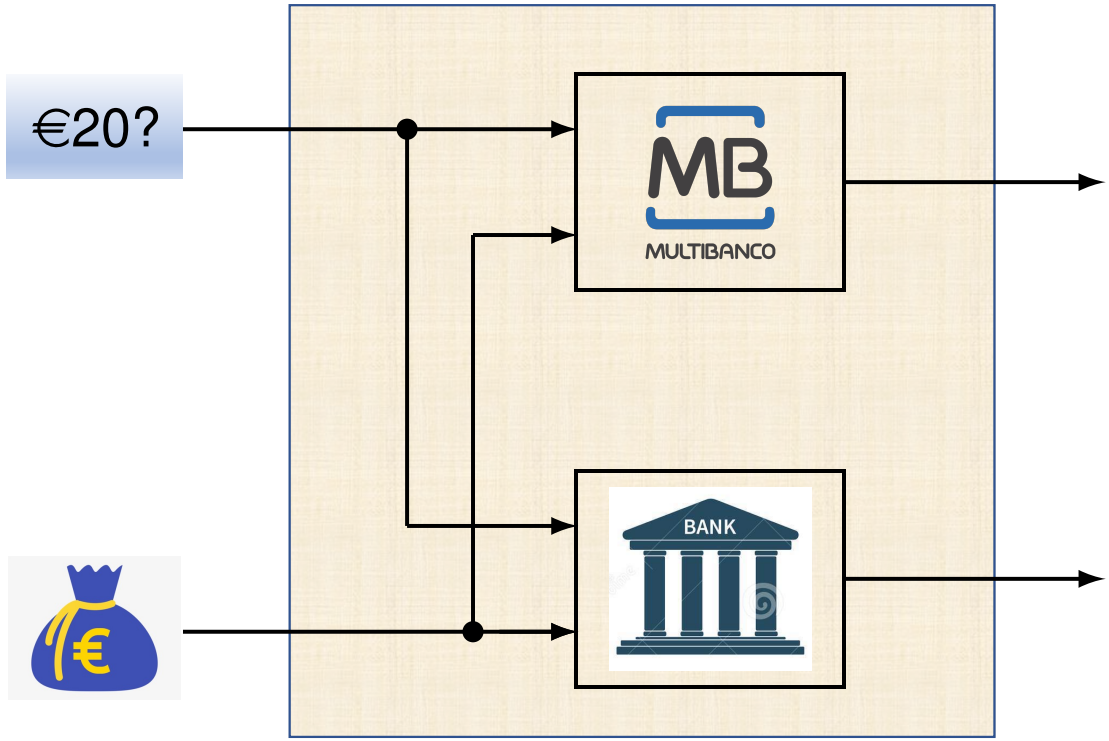
REACTIVE SYSTEMS

REACTIVE SYSTEMS

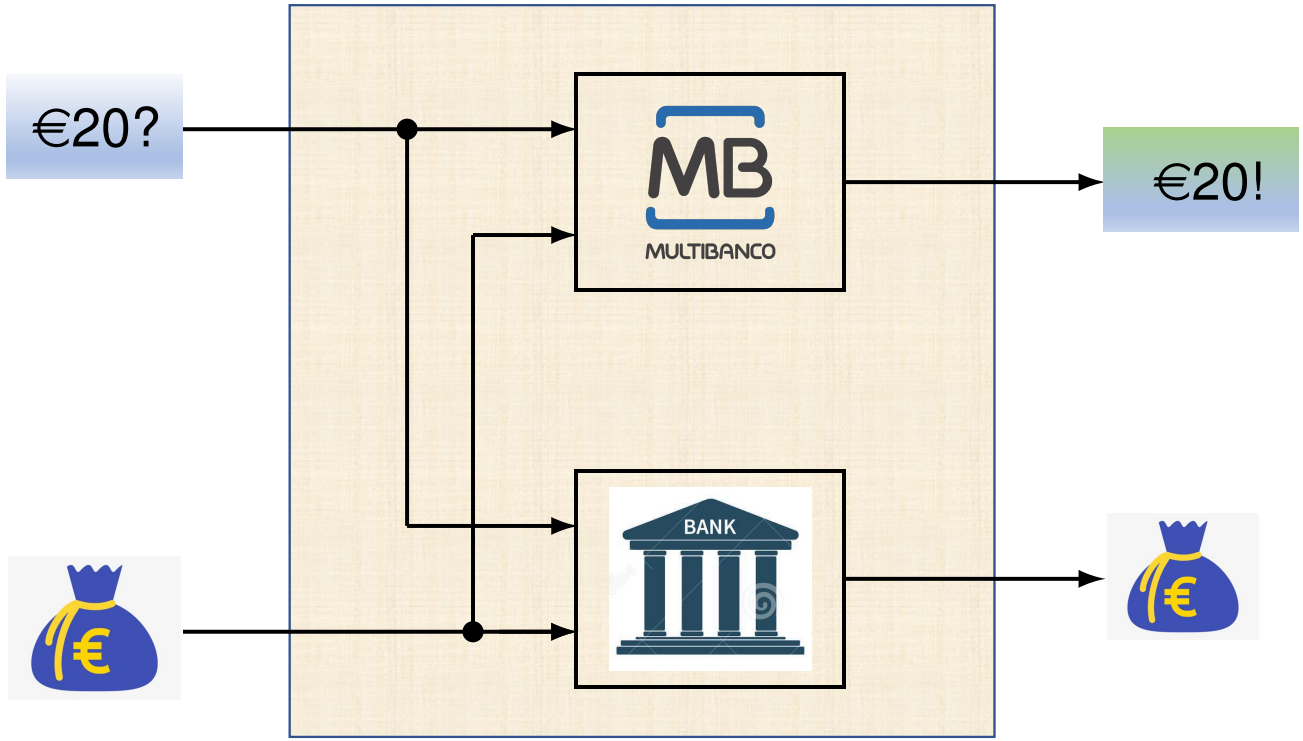
REACTIVE SYSTEMS



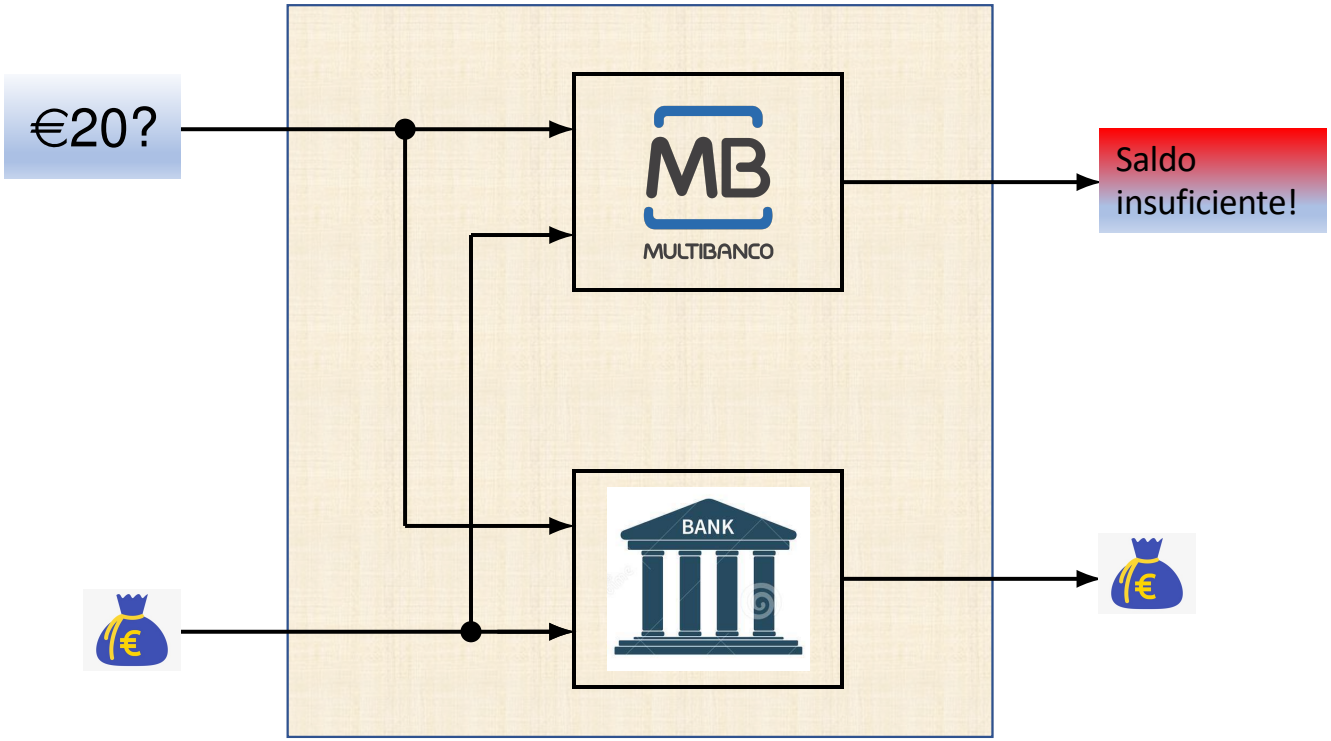
REACTIVE SYSTEMS



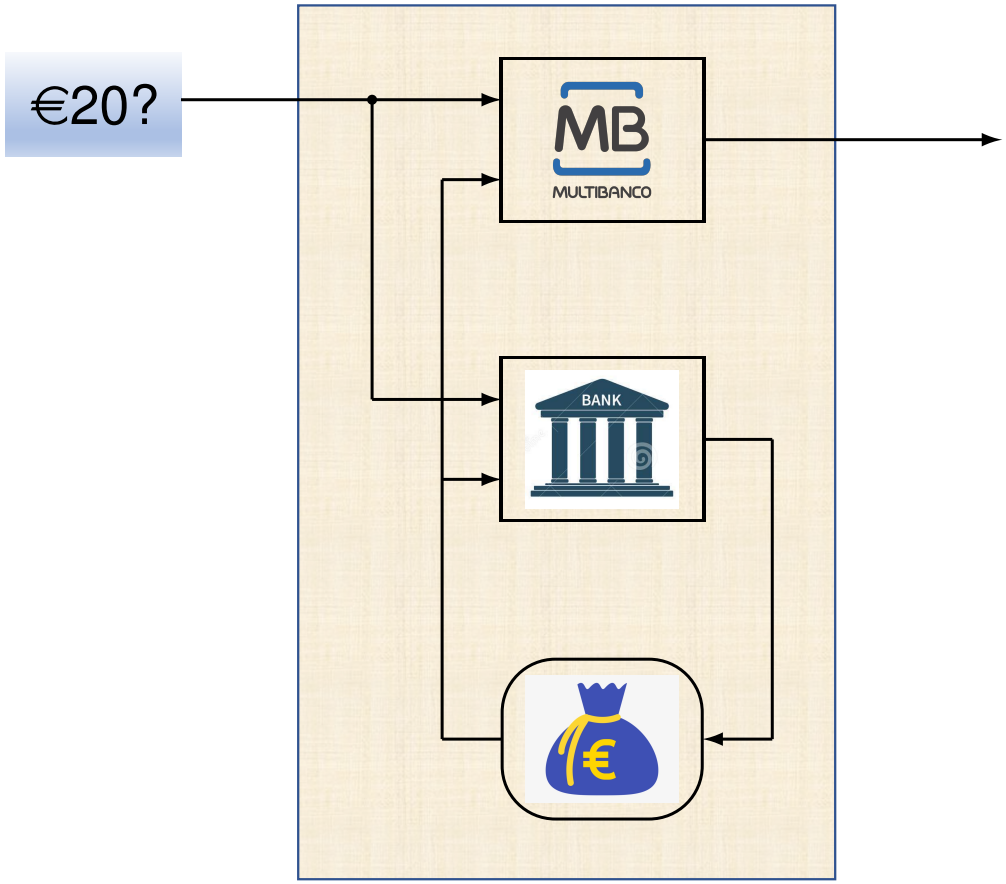
REACTIVE SYSTEMS



REACTIVE SYSTEMS

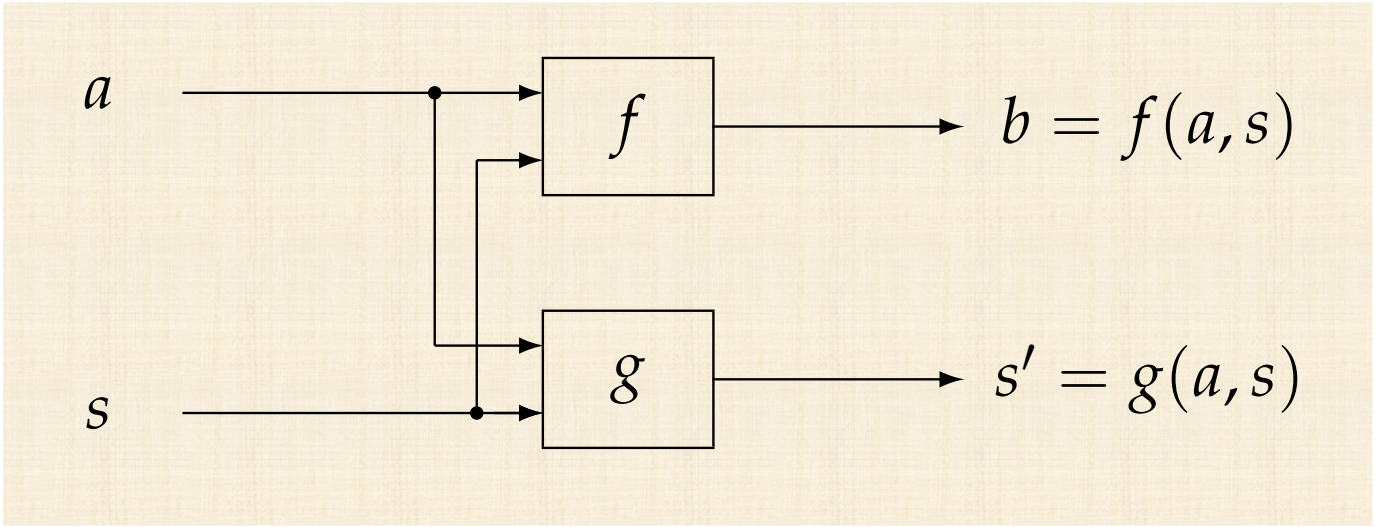


REACTIVE SYSTEMS



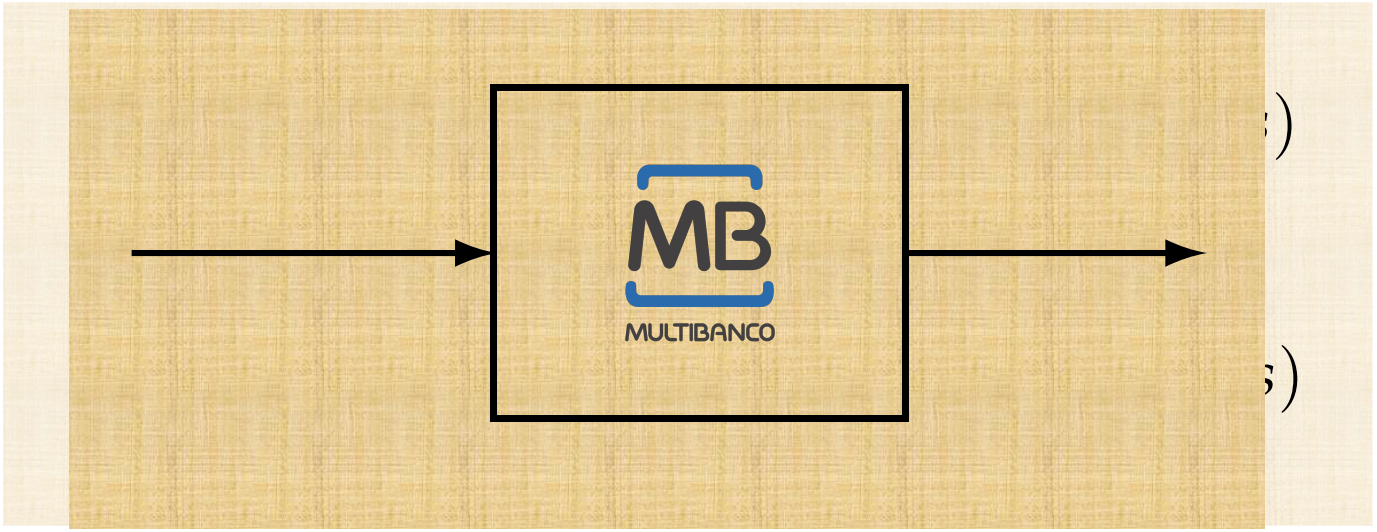
REACTIVE SYSTEMS

$$A \times S \xrightarrow{f} B$$
$$A \times S \xrightarrow{g} S$$
$$A \times S \xrightarrow{\langle f,g \rangle} B \times S$$



REACTIVE SYSTEMS

$$A \times S \xrightarrow{f} B$$
$$A \times S \xrightarrow{g} S$$
$$A \times S \xrightarrow{\langle f,g \rangle} B \times S$$



Cálculo de Programas
T13