

ATIAM

fpA Informatique

2023

Cours 4 : Haskell / Monades

Carlos Agon
agonc@ircam.fr

C'est quoi une monade ?

Divers connotations :

Philosophiques



Mathématiques



Informatiques



Une monade est une
notion abstraite

Les applications des monades
sont très variées et elles
peuvent être compliquées



les monades sont compliquées

Plan du cours

- Motivation
- Définitions : Catégorie, Foncteur, Monade
- Exemples (Haskell)
- do notation (Haskell)

Pourquoi des monades ?

On se place dans la programmation fonctionnelle.

Garder la **sémantique des programmes** comme **fonctions**

Conditions d'une fonction **pure** :

- Une fonction renvoie exactement le même résultat à chaque fois qu'elle est appelée avec les mêmes arguments
- Une fonction n'a pas d'état, ni ne peut accéder à aucun état externe.
- Une fonction n'a pas d'effets de bord.

Mais les programmes posent des problèmes qui ne peuvent pas être résolus par une fonction **pure** :

- Partialité: Calculs qui peuvent ne pas se terminer
- Non déterminisme.
- Effets de bord: Calculs qui accèdent / modifient l'état
- Exceptions:
- Continuations : Sauvegarder et restaurer l'état d'un programme
- Interactive Input/Output

Comment un programme construit à partir de fonctions **pures** peut être utile dans la vie réelle ?

Pourquoi des monades ?

Les monades viennent au secours pour séparer la partie fonctionnelle pure des autres aspects d'une computation.

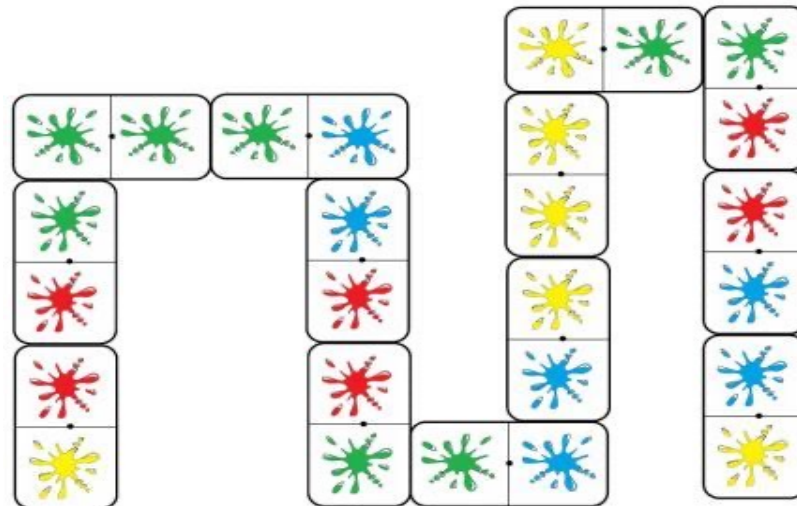
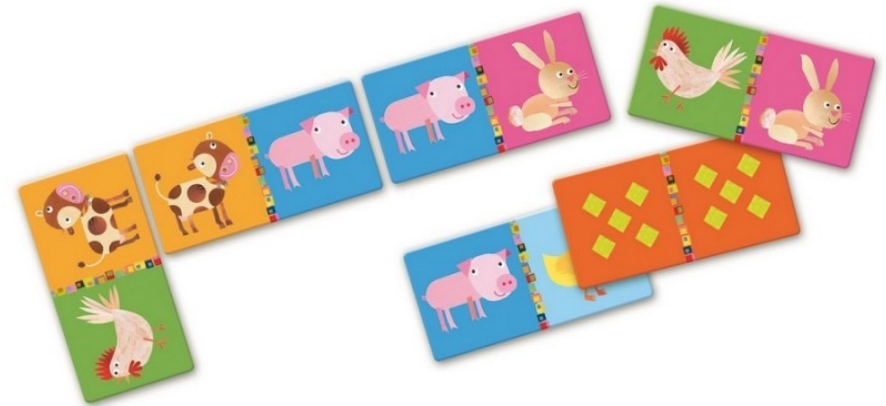
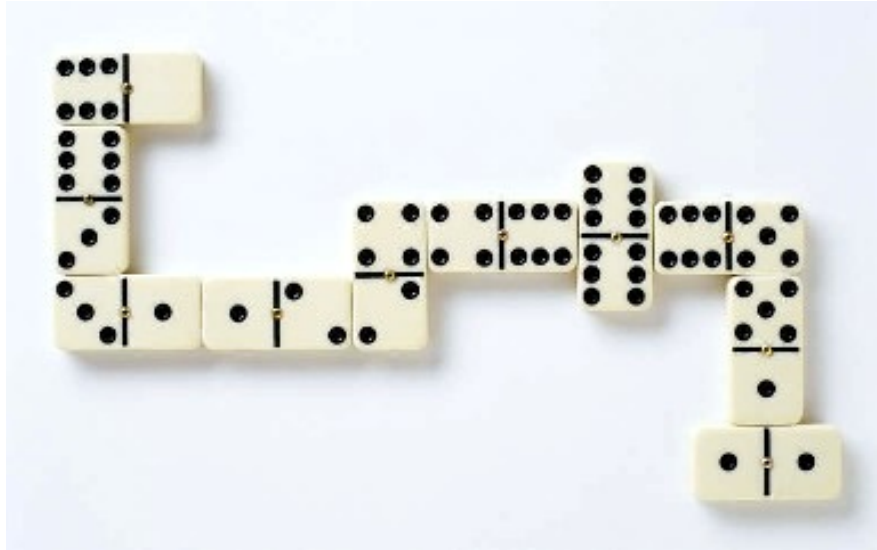
La monade est une notion issue de la Théorie de Catégories, **1958**.

Eugenio Moggi, **Notions of computation and monads**. Information and Computation , 1991.

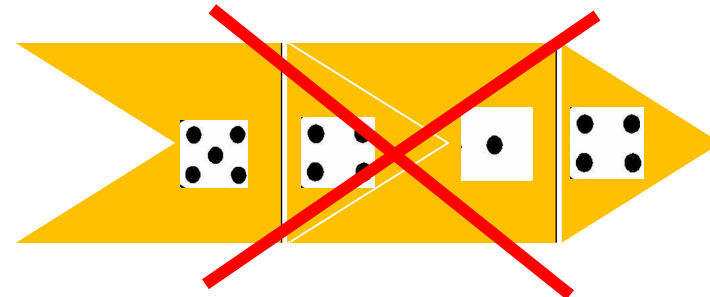
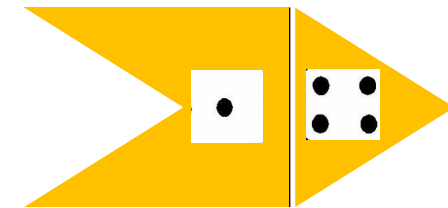
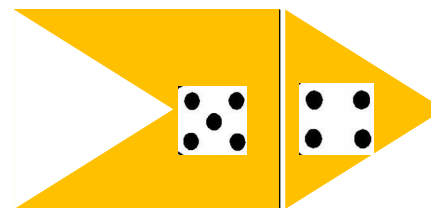
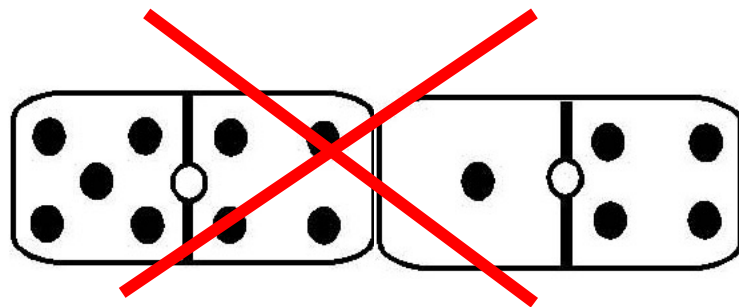
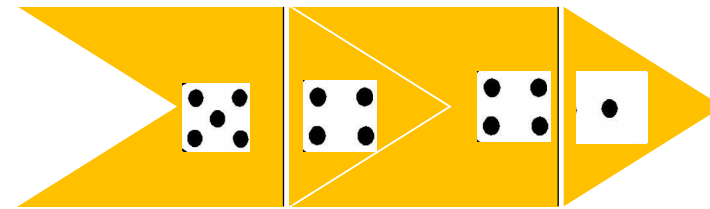
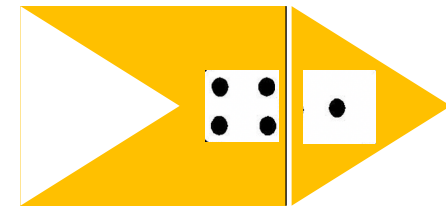
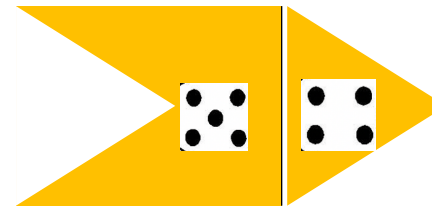
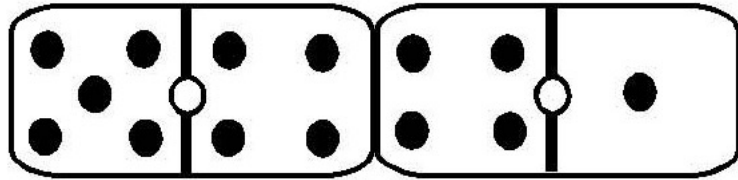
Philip Wadler, **Computational lambda calculus and monads**. In IEEE Symposium on Logic in Computer Science 1989.

But : après ce cours vous devriez pouvoir lire l'article de Wadler

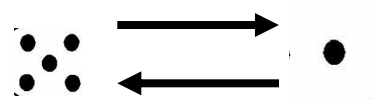
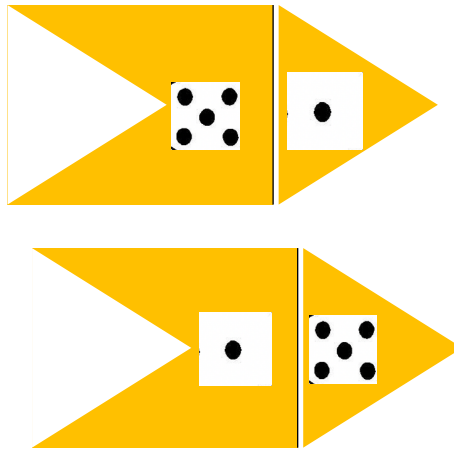
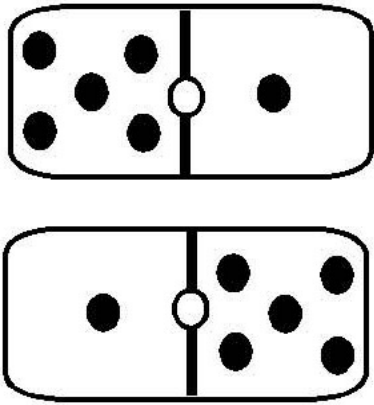
Dominos et catégories



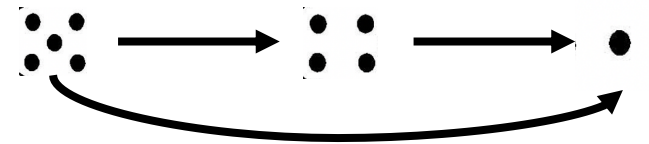
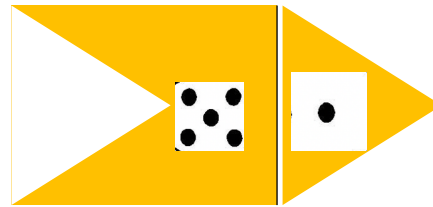
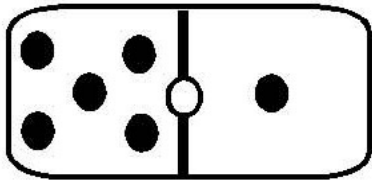
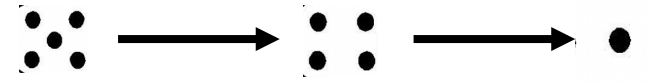
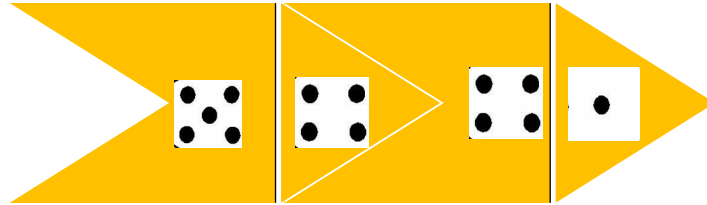
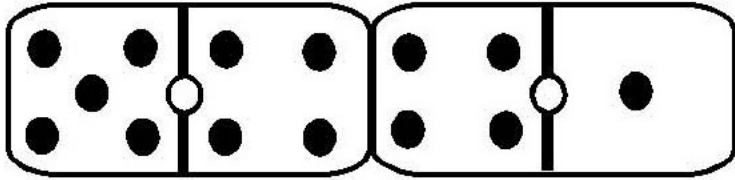
Fiches et Fishes



Fiches, Fishes et Flèches

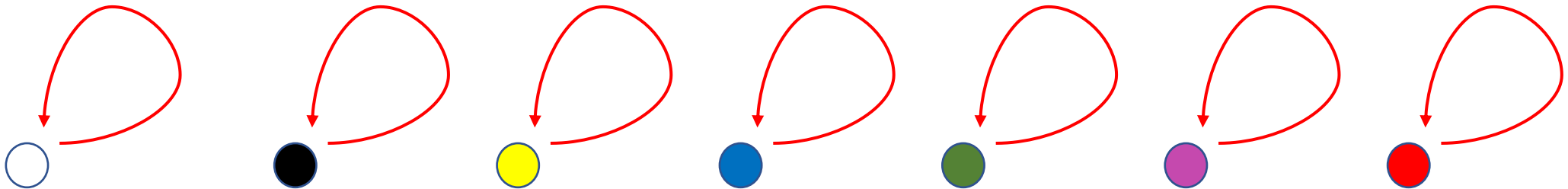
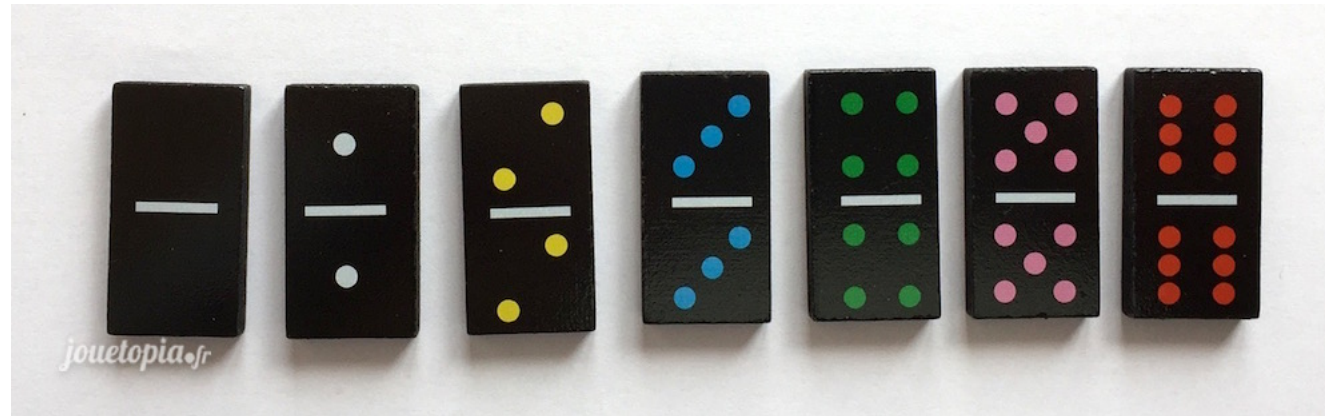


Fiches et Composition

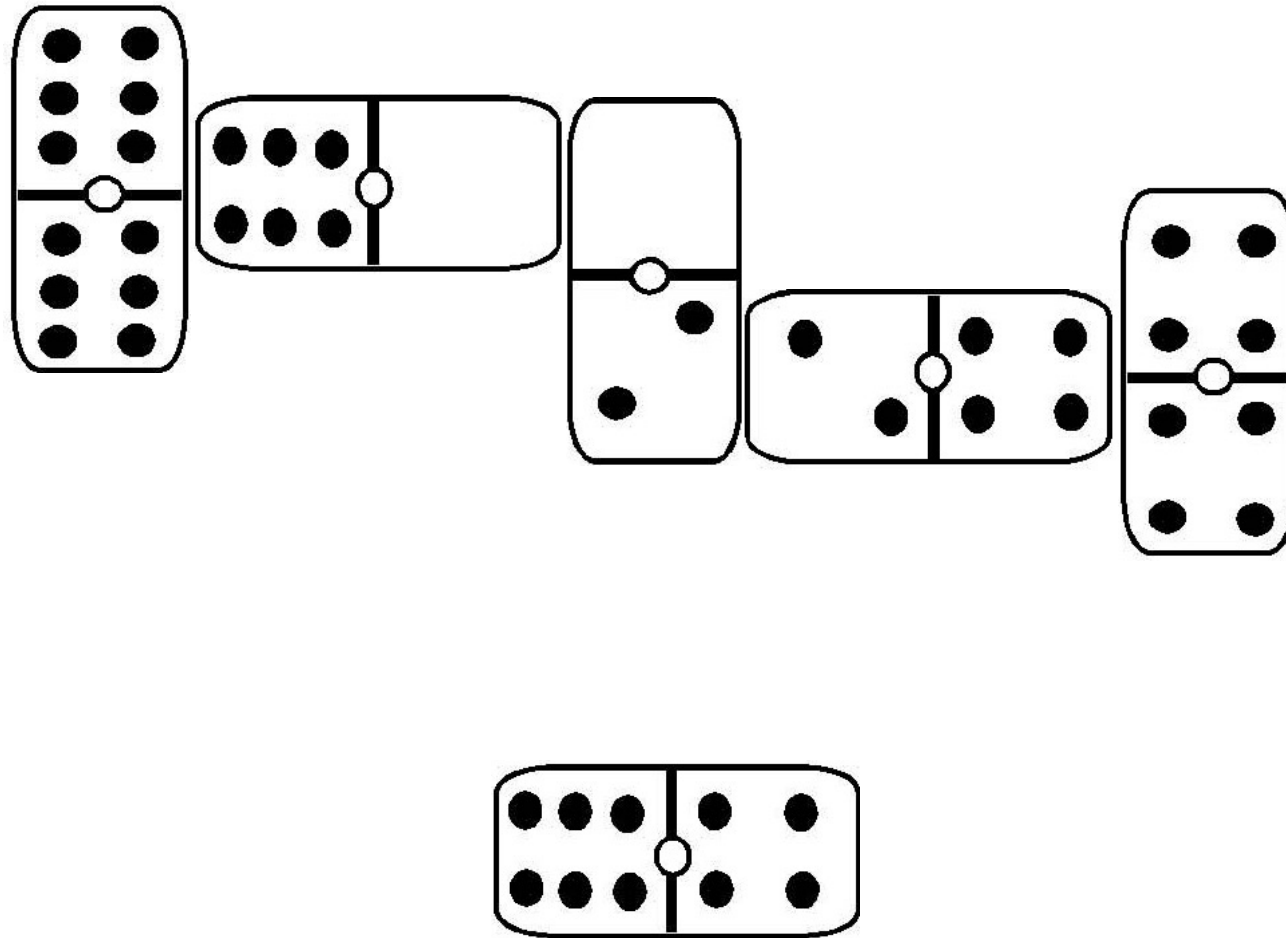


Comment identifier les figures/objets du domino ?

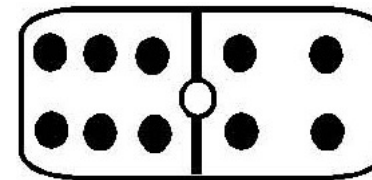
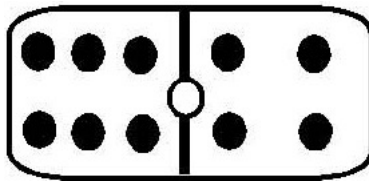
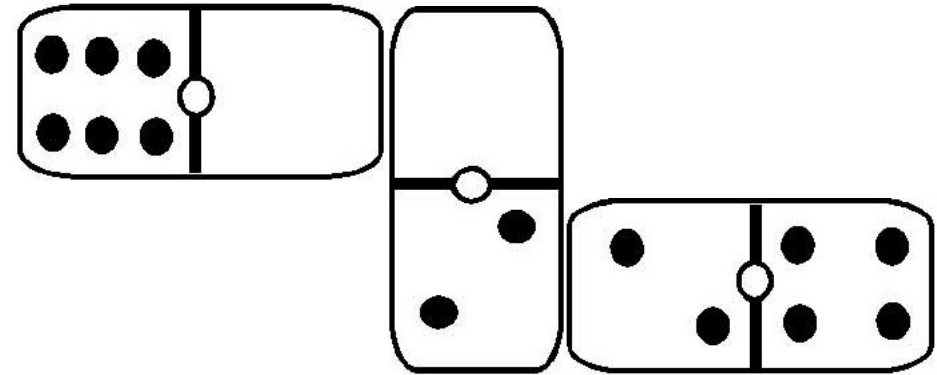
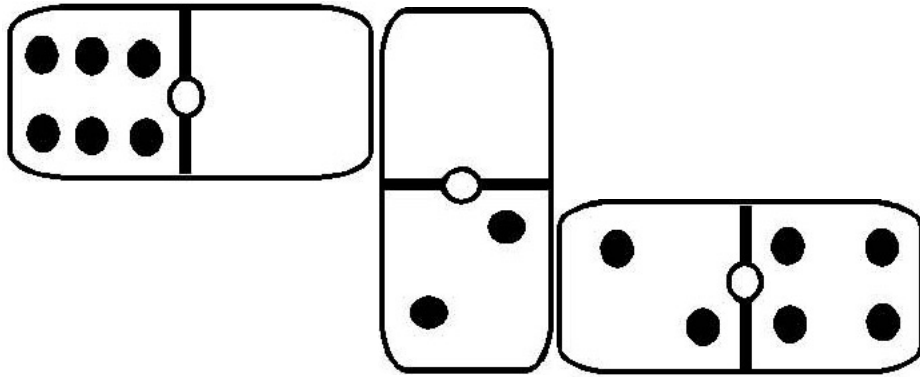
Doubles et identités



Identité

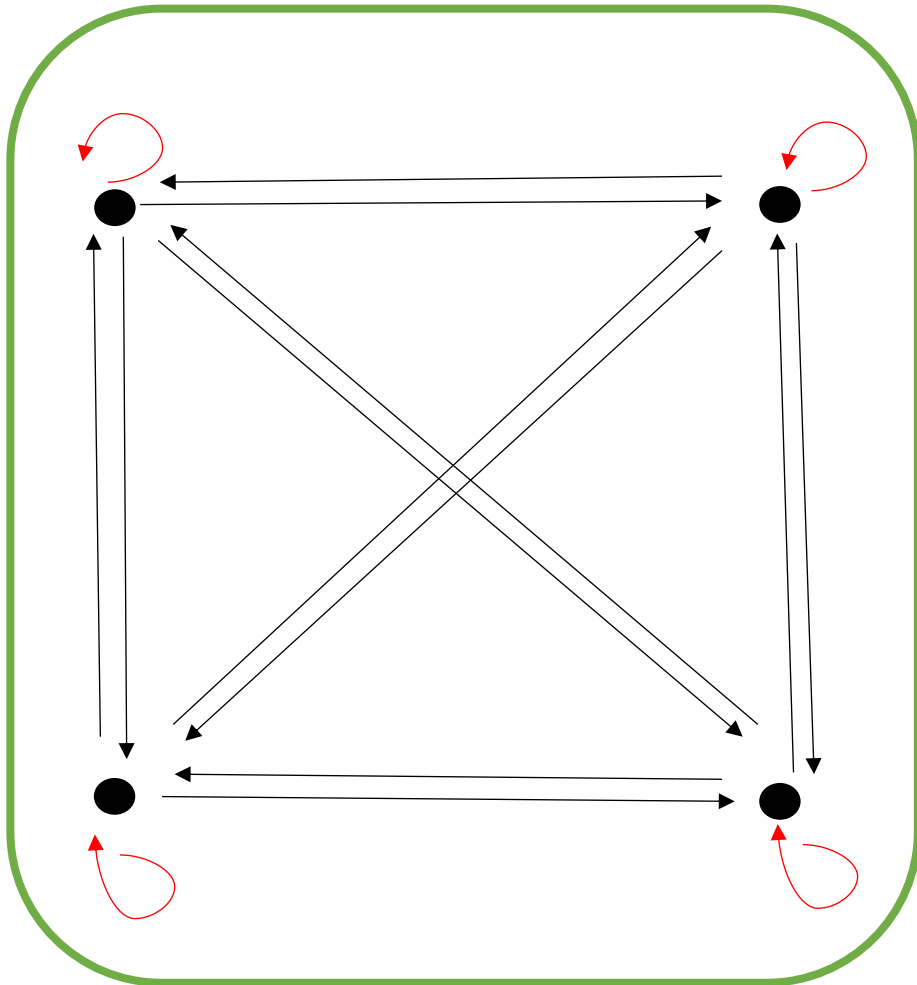


Associativité



Catégorie : définition

Donc

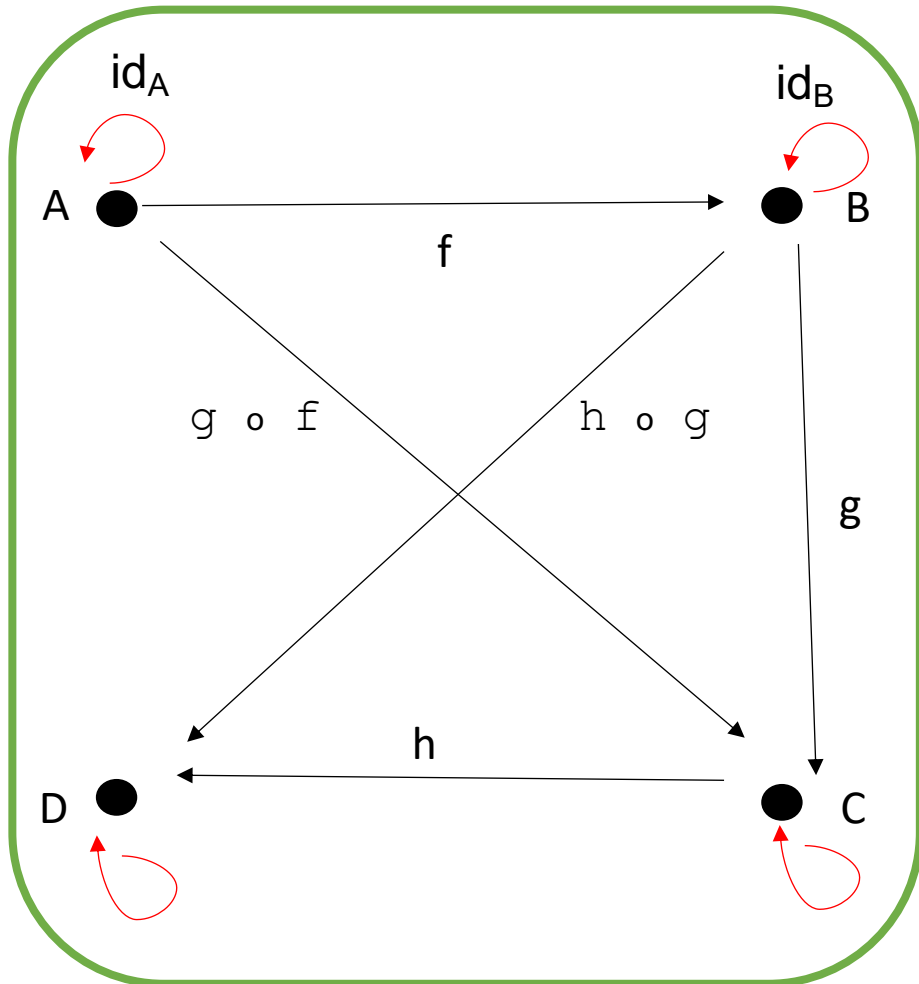


Une catégorie \mathcal{C} est définie par :

- Une collection d'objets
- Une collection de flèches entre objets
- Une flèche **id** de $\text{obj} \rightarrow \text{obj}$ pour tout obj

Catégorie : définition

Donc



Une catégorie \mathcal{C} est définie par :

- Une collection d'objets
- Une collection de flèches entre objets (morphismes)
- Une flèche **id** de $\text{obj} \rightarrow \text{obj}$ pour tout obj
- Une composition \circ que pour tout pair de morphismes $f: A \rightarrow B$ et $g: B \rightarrow C$ associe un morphisme $g \circ f: A \rightarrow C$

Avec les conditions suivantes :

1) la composition est associative :

Pour toutes $f: A \rightarrow B$; $g: B \rightarrow C$ et $h: C \rightarrow D$

$$(h \circ g) \circ f = h \circ (g \circ f)$$

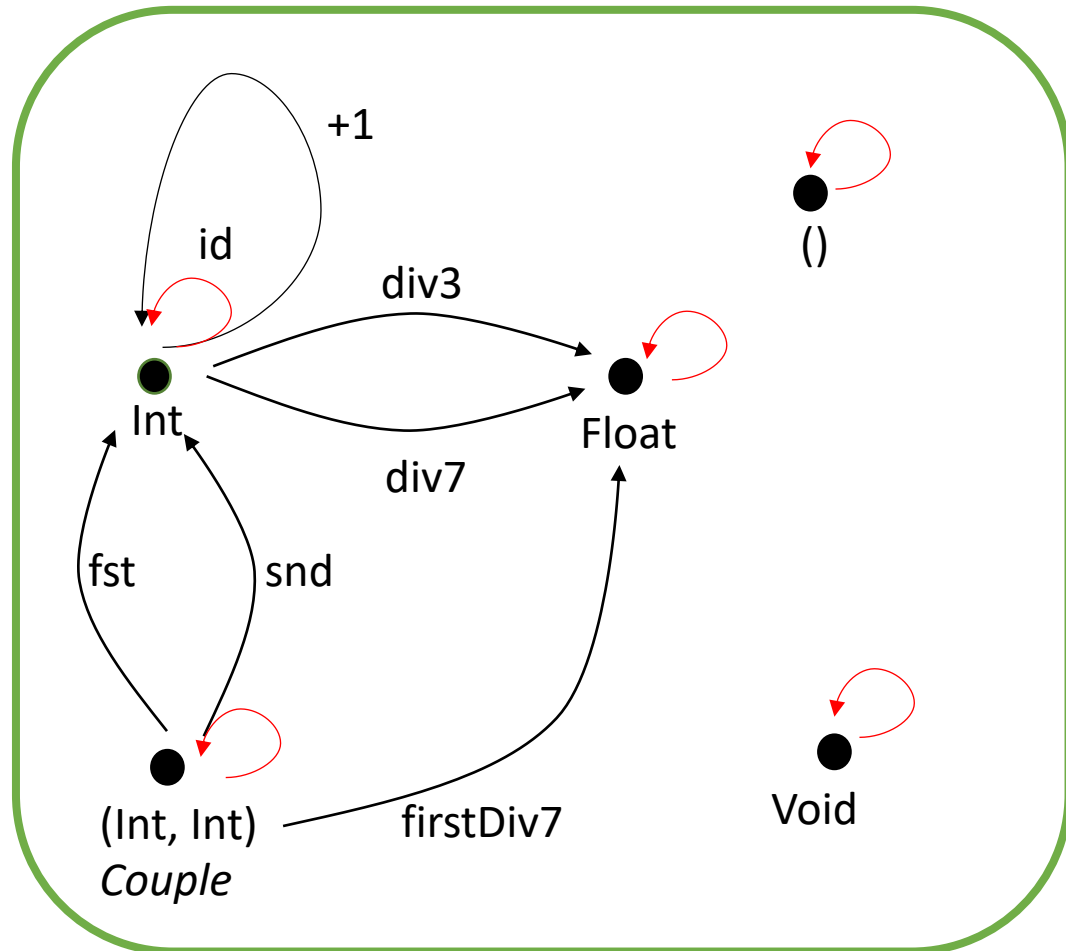
2) Les id sont des éléments neutres

Pour tout $f: A \rightarrow B$

$$id_B \circ f = f = f \circ id_A$$

La catégorie des types et fonctions en Haskell

Hask



```
div7 :: Int -> Float
div7 n = (fromIntegral n) / 7.0
```

```
+1 :: Int -> Int
+1 n = n+1
```

```
id :: a -> a
id x = x
```

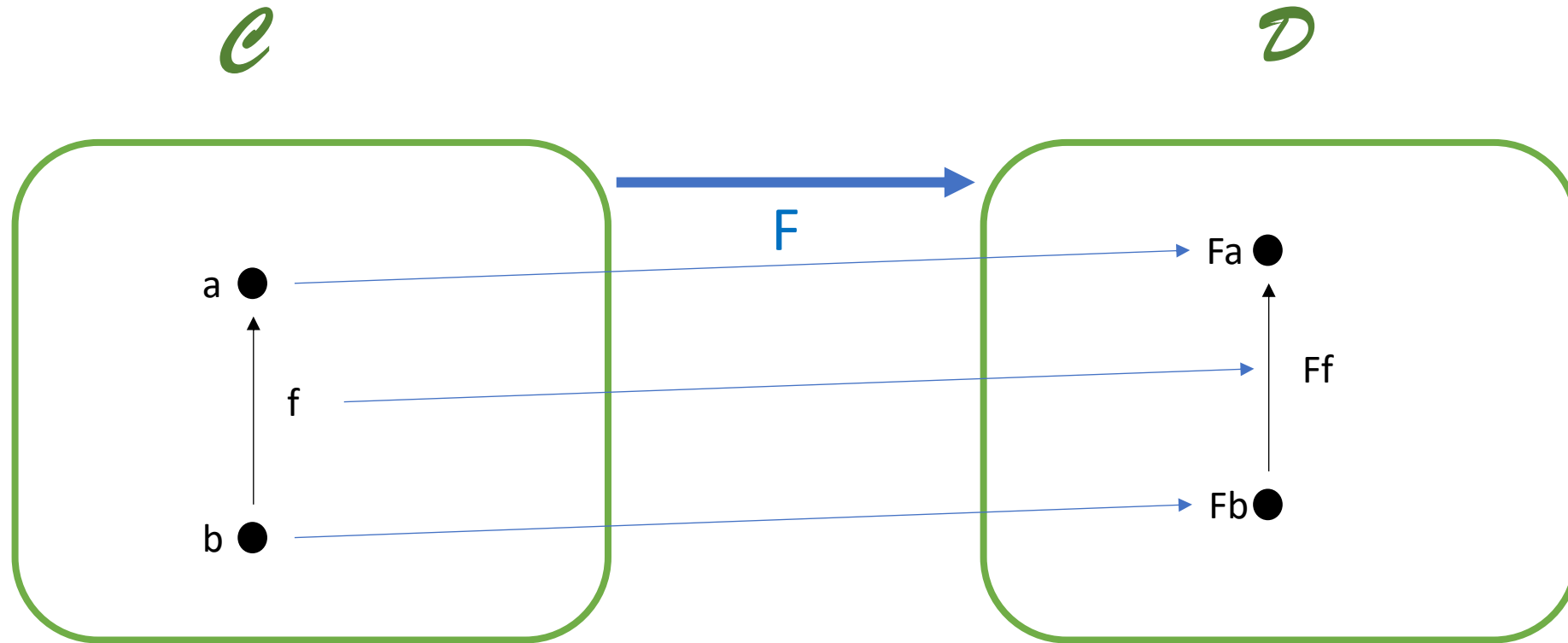
```
type Couple = (Int,Int)
```

```
> let ca = (7,3)
> fst ca
7
```

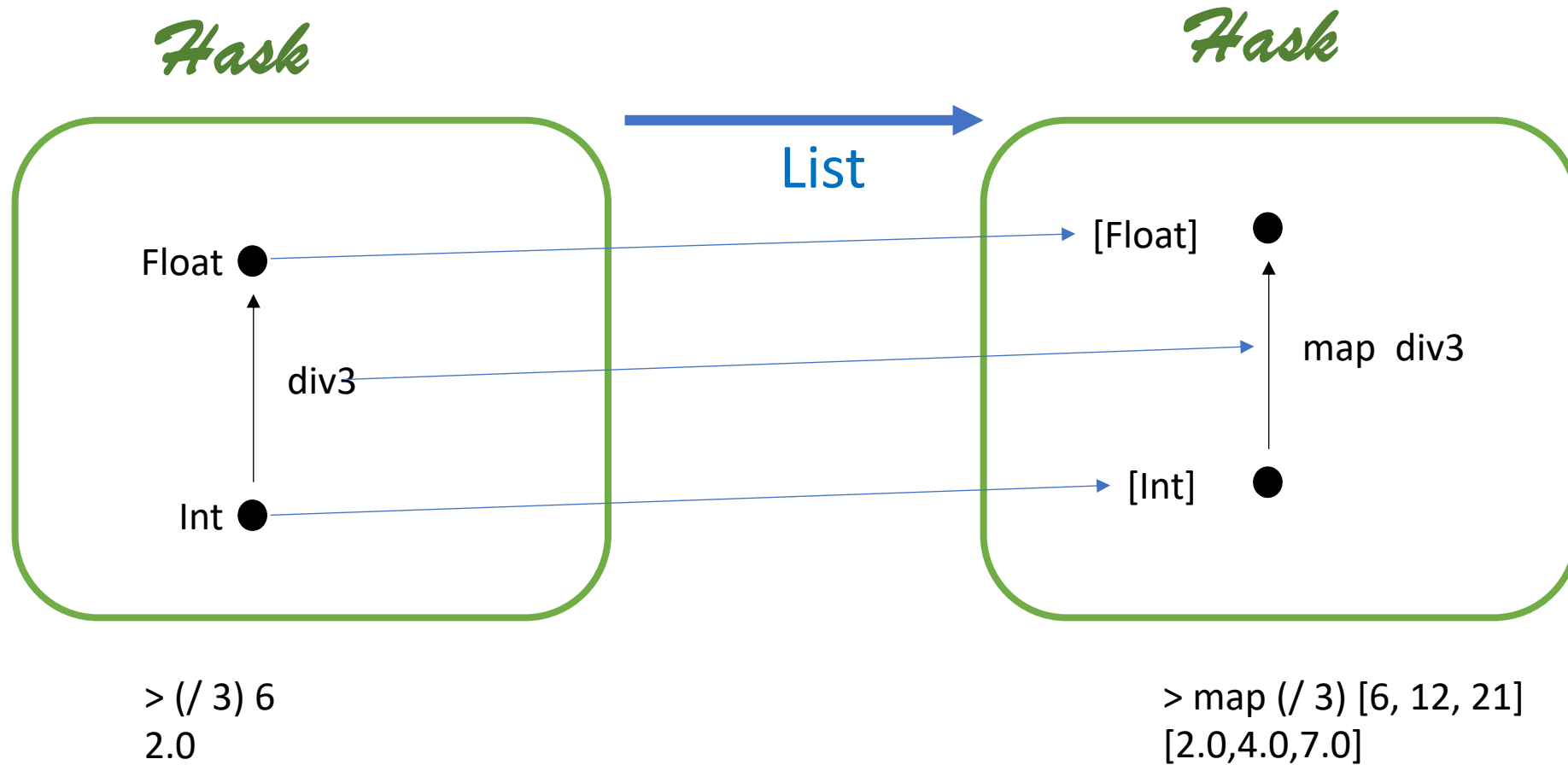
```
> :t fst
fst :: (a, b) -> a
```

```
firstDiv7 :: Couple -> Float
firstDiv7 = div7 . fst -- Pointfree Style
firstDiv7 ca
> 1.0
```

Foncteurs



Le Foncteur List



L'endofoncteur List

Hask



`map :: (a -> b) -> [a] -> [b]`

`map (/ 3) :: Fractional b => [b] -> [b]`

List : un exemple de foncteur

Pour les flèches

```
data List a = Nil | Cons a (List a)

instance Functor List where
  fmap f Nil = Nil
  fmap f (Cons x t) = Cons (f x) (fmap f t)

myliste :: List Int
myliste = Cons 6 (Cons 9 (Cons 12 Nil))
> [6,9,12]

fmap (/3) myliste
> [2.0,3.0,4.0]
```

Pour les objets

Démo OM ->

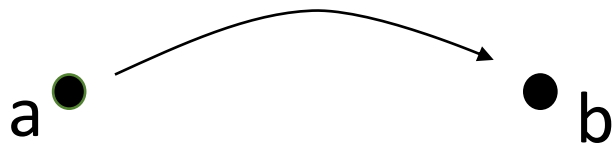
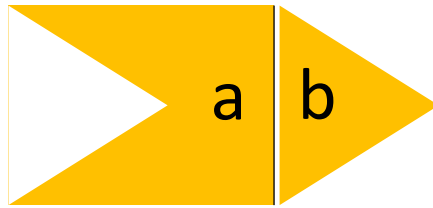
Pourquoi des monades ?

Mais les programmes posent des problèmes qui ne peuvent pas être résolus par une fonction **pure** :

- **Partialité**
- Non déterminisme
- Effets de bord
- Exceptions:
- Continuations
- Interactive Input/Output

Kleisli arrows

Une Kleisli arrow est une fonction dont le type de sortie est modifié pour enrichir la fonction avec de actions « extra fonctionnelles ».



Emboitement

$b \rightarrow [b]$

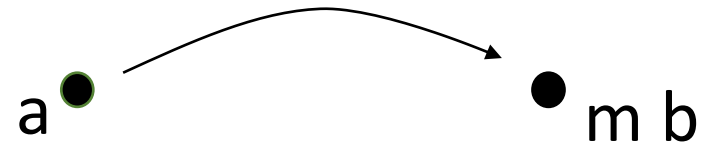
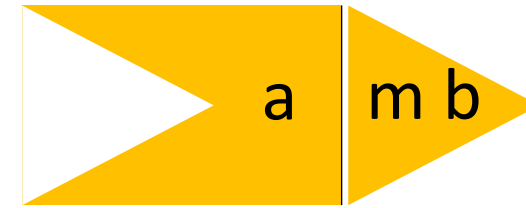
Modifié Comment ?

Embellissement

$b \rightarrow (b, \text{printer})$

Mise en contexte

$b \rightarrow (c, a) \rightarrow e$



Comment modifier le type de sortie pour une action particulière ?

Pourquoi des monades ?

Mais les programmes posent des problèmes qui ne peuvent pas être résolus par une fonction **pure** :

- Partialité
- Non déterminisme
- Effets de bord
- Exceptions
- Continuations
- Input/**Output**

Quoi faire si on veut un *Writer* ?

```
paire :: Int -> Bool
paire 0 = True
paire n = impaire (n-1)

impaire :: Int -> Bool
impaire 0 = False
impaire n = paire (n-1)
```

Solution 1

```
str :: String
str = " "

paire n = str = str ++ " paire "
           impaire (n-1)
```

```
> paire 5
False
impaire  paire  impaire  paire  impaire  paire
```

Quoi faire si on veut un *Writer* ?

```
paire :: Int -> Bool
paire 0 = True
paire n = impaire n-1
```

```
impaire :: Int -> Bool
impaire 0 = False
impaire n = paire n-1
```

```
strsize :: String -> Int
strsize s = length s
```

```
paireStr :: String -> Bool
paireStr = paire . strsize
-- point free style
```

```
paire1 :: Int -> Writer Bool
paire1 0 = (True, 'paire1')
paire1 n = impaire1 n-1
```

```
impaire1 :: Int -> Writer Bool
impaire1 0 = False
impaire1 n = paire n-1
```

```
strsize1 :: String -> Writer Int
strsize1 s = length s
```

```
paireStr1 :: String -> Writer Bool
paireStr1 = paire1 . strsize1
-- point free style
```

```
b - - > (b, String)
```

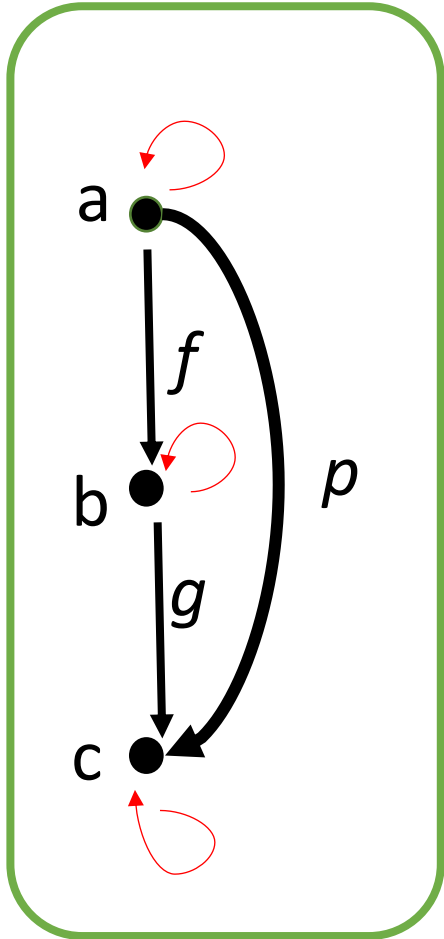
```
type Writer b = (b ,String)
```

```
> paireStr1 'hello'
```

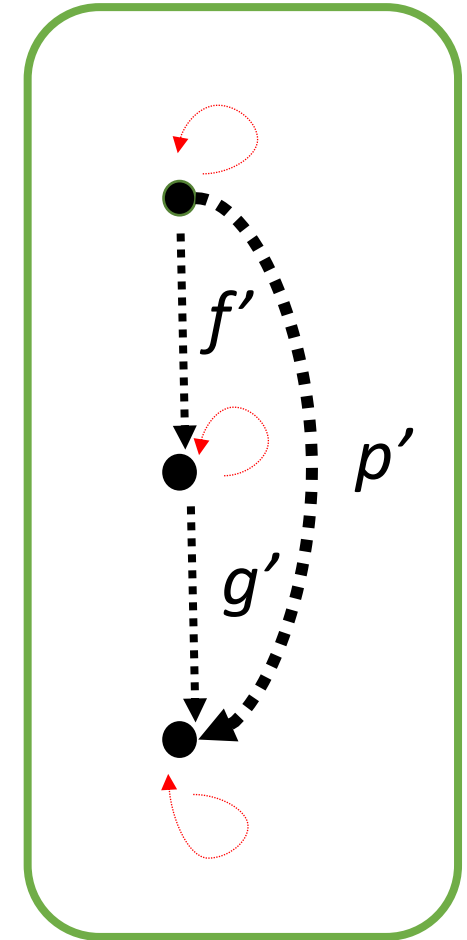
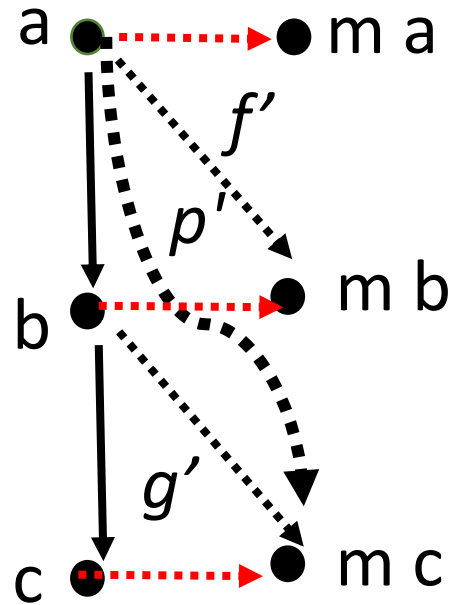
Couldn't match expected type 'Bool'
with actual type '(Bool, String)'

Kleisli Catégorie

$m\ t : t \multimap (t, \text{String})$



$$p = g \circ f$$



$$p' = f' \gg g'$$

Une définition de Monade

Une monade est définie par un triplet :

- Un constructeur de type **m** appelé **type monadique**
- Un opérateur **>=>** pour la composition de **fonctions monadiques**
- Un opérateur **return** pour la construction de **valeurs monadiques**

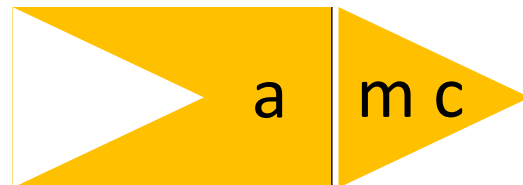
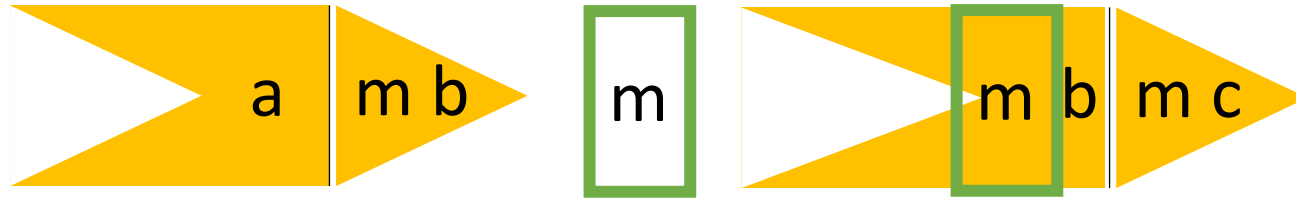
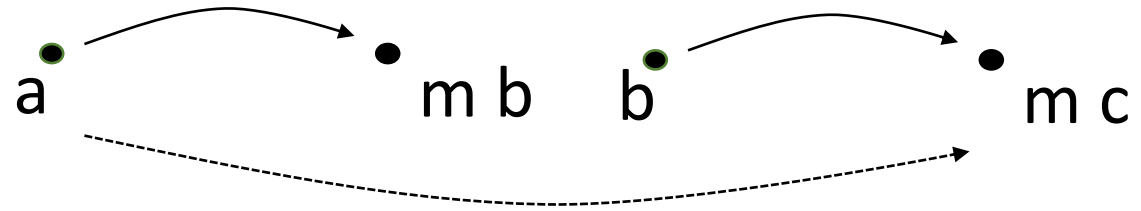
```
type Writer a = (a, String)
```

```
class Monad m where  
    (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)  
    return :: a -> m a
```

Une monade définie la composition d'une Kleisli catégorie

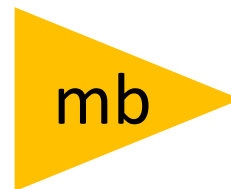
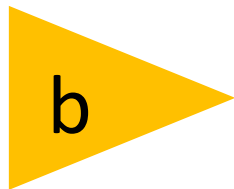
Composition : l'opérateur FISH (\Rightarrow)

$(\Rightarrow) :: (a \rightarrow m b) \rightarrow (b \rightarrow m c) \rightarrow (a \rightarrow m c)$



Identité : return

`return :: b -> m b`



La monade Writer

Une monade est définie par un triplet :

- Un constructeur de type **m** appelé **type monadique**
- Un opérateur **>=>** pour la composition de **fonctions monadiques**
- Un opérateur **return** pour la construction de **valeurs monadiques**

```
type Writer a = (a ,String)
```

```
                                m1                m2                m1 >=> m2
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
m1 >=> m2 = \x:a ->
    let (y, s1) = m1 x
        (z, s2) = m2 y
    in (z, s1 ++ s2)
```

```
return :: a -> Writer a
return a = (a , "")
```

La monade Writer

```
pairel :: Int -> Writer Bool
pairel 0 = (True, 'pairel')
pairel n = impairel n-1

impairel :: Int -> Writer Bool
impairel 0 = False
impairel n = paire n-1

strsize1 :: String -> Writer Int
strsize1 s = length s

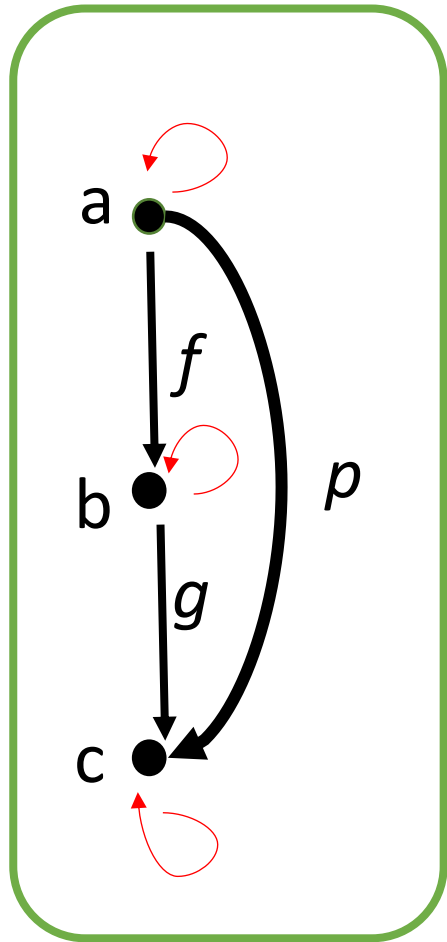
paireStr1 :: String -> Writer Bool
paireStr1 = strsize1 >=> pairel
-- point free style
```

```
> paireStr1 'hi'
True
```

```
'paireStr1 paire impaire paire'
```

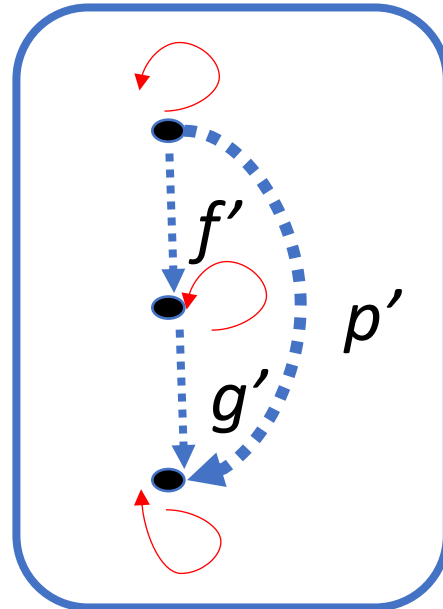
\Rightarrow l'abstraction de la composition fonctionnelle !

HasK



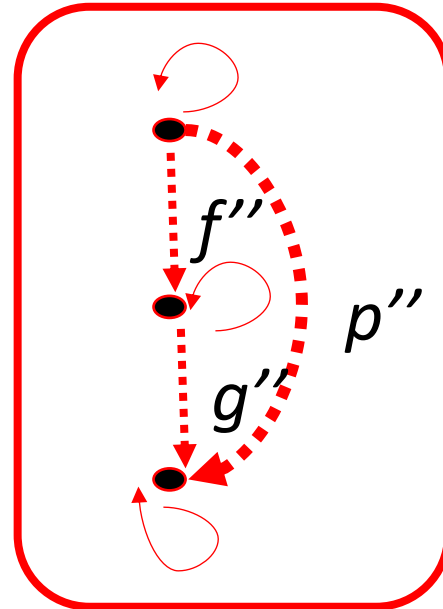
$$p = g \circ f$$

$\lambda 1$



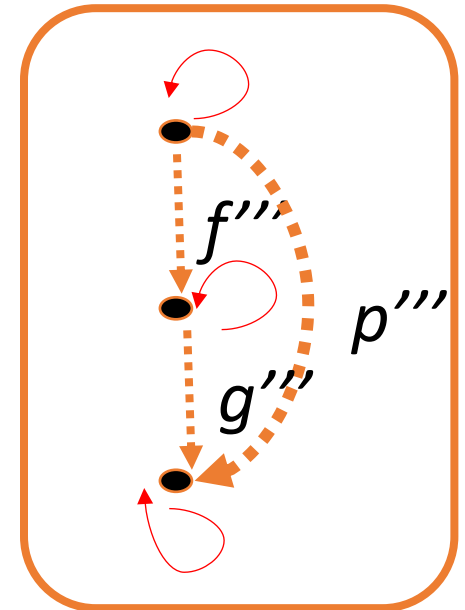
$$p' = f' \Rightarrow g'$$

$\lambda 2$



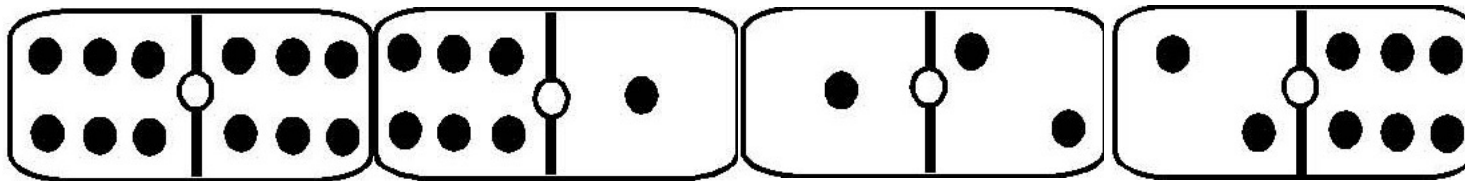
$$p'' = f'' \Rightarrow g''$$

λn



$$p''' = f''' \Rightarrow g'''$$

Un nouveau dominos



Le cadavre Exquis Boira Le vin Nouveau

C'est tout !

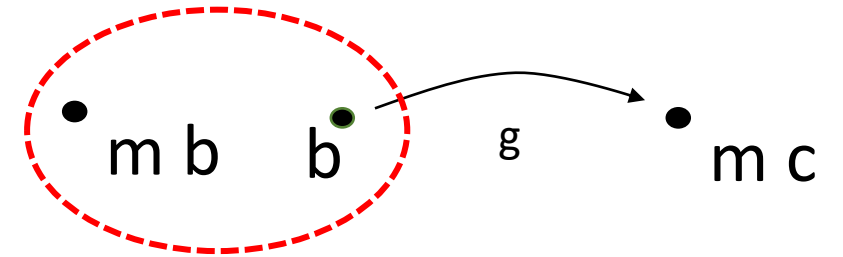
L'opérateur BIND (>>=)

$(>=>) :: (a \rightarrow m\ b) \rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ c)$

$(>>=) :: m\ b \rightarrow (b \rightarrow m\ c) \rightarrow m\ c$

$f\ >=>\ g = \backslash va \rightarrow (f\ va)\ >>= g$

>>= ➡ >=>



```
type Writer a = (a ,String)

(>>=) :: Writer a -> (a -> Writer b) -> Writer b
(a1, s1) >>= f = let (b, s2) = f a1 in
                  (b, s1 ++ s2)

return :: a -> Writer a
return a = (a , "")
```

```
class Monad m where
  (>>=) :: m b -> (b -> m c) -> m c

  return :: a -> m a
```

Les lois de la monade avec >>=

Pour être une **vraie** monade, `return` et `>>=` doivent satisfaire les 3 lois suivantes :

1. $(\text{return } x) \gg= f = f\ x$
2. $mv \gg= \text{return} = mv$
3. $(mv \gg= f) \gg= g = mv \gg= (\backslash x \rightarrow f\ x \gg= g)$

La première loi dit que `return` soit l'identité de gauche par rapport à `bind`

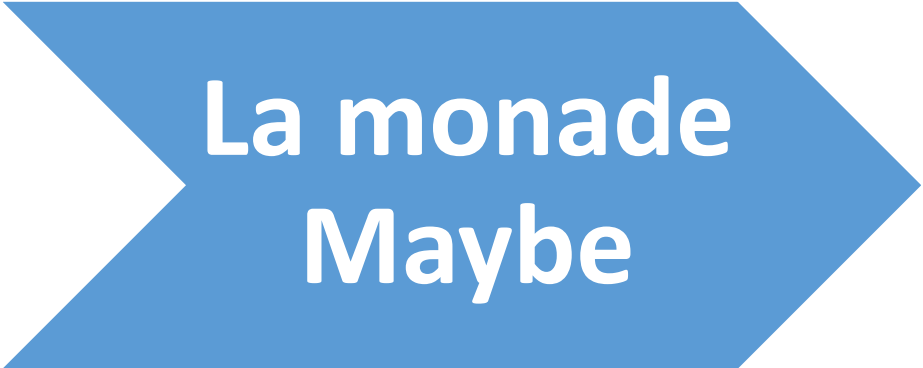
La deuxième loi dit que `return` soit l'identité à droite par rapport `bind`.

La troisième loi est une sorte de loi d'associativité pour `bind`

Tout constructeur de type avec des opérateurs `return` et `bind` qui satisfont aux trois lois de la monade est une monade.

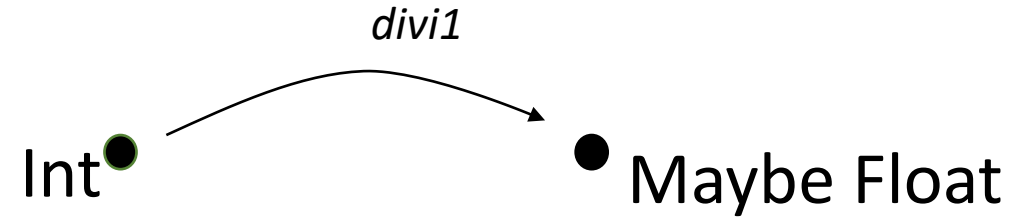
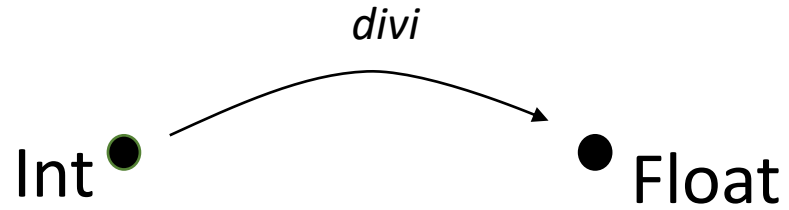
Le compilateur Haskell ne vérifie pas que les lois sont valables pour une instance de la classe `Monad`.

Il appartient au programmeur de s'assurer que toute instance `Monad` qu'il crée satisfait aux lois de la monade.



**La monade
Maybe**

Pourquoi des monades ?



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

```
divi :: Int -> Float
divi n = 100.0/ (fromIntegral n)
```

```
>divi 4
25.0
```

```
>divi 0
error:
```

```
divi1 :: Int -> Maybe Float
divi1 0 = Nothing
divi1 n = Just (100.0/ (fromIntegral n))
```

```
>divi1 4
Just 25.0
```

```
>divi1 0
Nothing
```

Le foncteur Maybe

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just x) = Just (f x)
```

```
:t fmap
fmap :: Functor f => (a -> b) -> f a -> f b

(a -> b) -> Maybe a -> Maybe b
```

```
divi :: Int -> Float
divi n = 100.0 / (fromIntegral n)
```

```
fmap (+1) (Just 6)
> Just 7
```

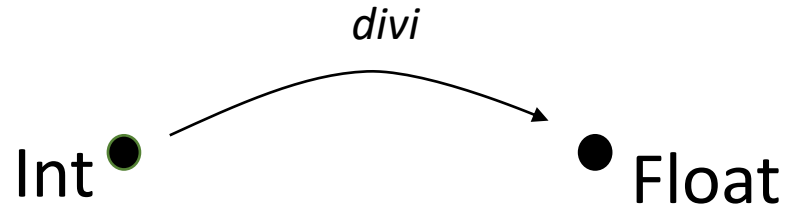
```
>fmap divi Nothing
Nothing
```

```
>divi <$> Nothing
Nothing
```

```
>fmap divi (Just 4)
Just 25.0
```

```
>divi <$> (Just 4)
Just 25.0
```

Pourquoi des monades ?

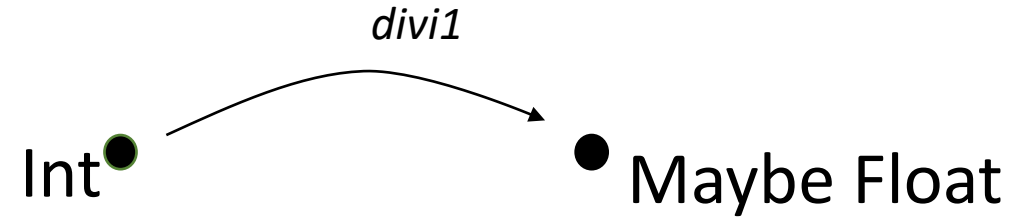


```
divi :: Int -> Float
divi n = 100.0/ (fromIntegral n)
```

```
racine :: Float -> Float
racine n = sqrt n
```

```
comp :: Int -> Float
comp = racine . divi
```

```
> comp 4
5.0
> comp (-2)
error:
```



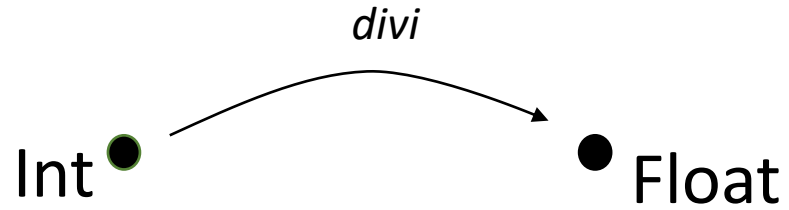
```
divi1 :: Int -> Maybe Float
divi1 0 = Nothing
divi1 n = Just (100.0/ (fromIntegral n))
```

```
racine1 :: Float -> Maybe Float
racine1 n = if (n < 0) then Nothing
           else Just (sqrt n)
```

```
comp1 :: Int -> Maybe Float
comp1 = racine1 . Divi1
```



Pourquoi des monades ?

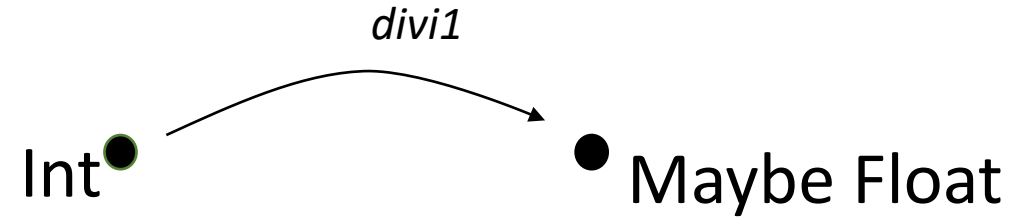


```
divi :: Int -> Float
divi n = 100.0/ (fromIntegral n)
```

```
racine :: Float -> Float
racine n = sqrt n
```

```
comp :: Int -> Float
comp = racine . divi
```

```
> comp 4
5.0
> comp (-2)
error:
```



```
divi1 :: Int -> Maybe Float
divi1 0 = Nothing
divi1 n = Just (100.0/ (fromIntegral n))
```

```
racine1 :: Float -> Maybe Float
racine1 n = if (n < 0) then Nothing
            else Just (sqrt n)
```

```
comp1 :: Int -> Maybe Float
comp1 = divi1 ==> racine1
```

```
> comp1 4
Just 5.0

> comp1 (-2)
Nothing
```


La Monade MayBe

Une monade est définie par un triplet :

- Un constructeur de type **m** appelé **type monadique**
- Un opérateur **>=>** pour la composition de **fonctions monadiques**
- Un opérateur **return** pour la construction de **valeurs monadiques**

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

```
class Monad m where  
    (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)  
    return :: a -> m a
```

```
(>=>) :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)  
return :: a -> Maybe a
```

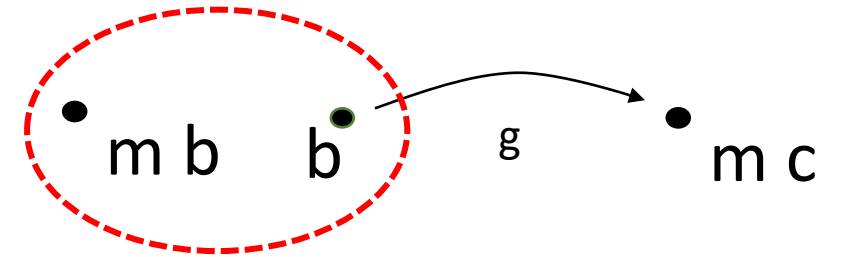
```
instance Monad Maybe where  
    return a = Just a
```

```
m1 >=> m2 =  
    \va ->  
        let rep = m1 va in  
        case rep of  
            Nothing -> Nothing  
            Just vb -> m2 vb
```

L'opérateur BIND >>=

$(>=>) :: (a \rightarrow m\ b) \rightarrow (b \rightarrow m\ c) \rightarrow (a \rightarrow m\ c)$

$(>>=) :: m\ b \rightarrow (b \rightarrow m\ c) \rightarrow m\ c$



$f\ >=>\ g = \backslash va \rightarrow (f\ va)\ >>= g$

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b

  return :: a -> m a
```

```
instance Monad Maybe where
  Nothing >>= g = Nothing
  Just x >>= g = g x

  return a = Just a
```



La monade List

Non-déterminisme/ monade List

Non-déterminisme : Computations qui peuvent retourner plusieurs résultats.

Solution : retourner tous le valeurs possibles.



$(\gg=) :: [b] \rightarrow (b \rightarrow [c]) \rightarrow [c]$

Une fonction monadique est appliquée à chaque élément de la liste d'entrée (valeurs possibles).
Le listes résultantes sont concaténées pour produire une liste avec tous les résultats possibles.

La monade List

```
dice :: Int -> Int
dice n = 3

op :: Int -> Int
op r = r * 9

op1 :: Int -> Int
op1 r = r + 9

prog :: Int -> Int
prog = op1 . op . dice

> dice 6
3

> prog 6
36
```

```
diceND :: Int -> [Int]
diceND n = [1..n]

opND :: Int -> [Int]
opND r = [r * 9]

op1ND :: Int -> [Int]
op1ND r = [r + 9]

progND :: Int -> [Int]
progND n = (diceND n) >>= opND >>= op1ND

> diceND 12
[1,2,3,4,5,6]

> progND 6
[18,27,36,45,54,63]
```



La monade Erreur

La monade Erreur

```
data Erreur a = Err String | Bien a
               deriving (Show)
```

```
instance Functor Erreur where
    fmap _ (Err x) = Err x
    fmap f (Bien x) = Bien (f x)
```

```
> fmap (+1) (Bien 3)
Bien 4
```

```
> fmap (+1) (Err "Mauvais numero")
Err "Mauvais numero"
```

Monade Erreur

```
data Erreur a = Err String | Bien a
               deriving (Show)
```

```
instance Monad Erreur where
    return x  = Bien x
    Err x >>= f  = Err x
    Bien x >>= f  = f x
```

```
divi2 :: Int -> Erreur Float
divi2 0 = Err "Division par zero"
divi2 n = Bien (100.0/ (fromIntegral n))


racine2 :: Float -> Erreur Float
racine2 n  = if (n < 0) then
               Err "Racine d'un nombre negatif"
            else Bien (sqrt n)
```

```
comp2 :: Int -> Erreur Float
comp2  n = (return n) >>= divi2 >>= racine2
```

```
comp3 :: Int -> Erreur Float
comp3 n = do
    x <- divi2 n
    y <- racine2 x
    return y
```

```
> comp2 0
Err "Division par zero"
```

```
> comp2 (-2)
Err "Racine d'un nombre negatif"
```

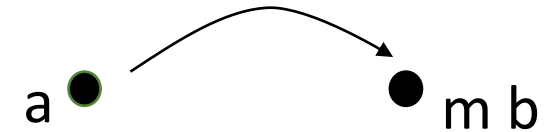
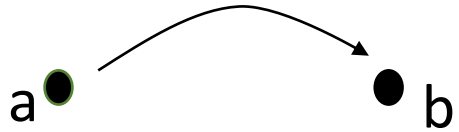
La monade Maquette

Démo->OM

State monade

Lecture et écriture d'un état en mémoire

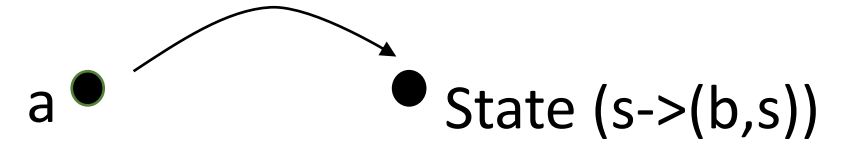
offset	durée
0	2



$a \rightarrow b$

$(a, s) \rightarrow (b, s)$

$a \rightarrow (s \rightarrow (b, s))$



```
newtype State s a = State (s -> (a, s))
```

State monade

```
newtype State s a = State (s -> (a,s))
```

```
instance Monad (State s) where
```

```
    return a = State ( \s -> (a,s) )
```

```
    f >=> g = \va -> (f va) >>= g
```

```
    State (h) >>= g = State ( \s -> let (v,s1) = h s in  
                                runState (g v) s1)
```

```
f :: a --> State (s -> (b,s))
```

```
g :: b --> State (s -> (c,s))
```

```
r :: a --> State (s -> (c,s))
```

```
h :: s -> (b,s)
```

```
v :: b
```

```
runState (g v) s1 :: (c, s)
```

Maquete/State monade

```
data Box = Box
  { offset :: Int
  , dur    :: Int }

move :: State Box Int
move sec = do
  boite <- get
  let newoffset = (offset boite) + (sec * 1000)
  put ( boite {offset = newoffset } )
  return newoffset

main :: IO ()
main = do
  let (offset, box) = runState (move 3) (Box 0 0)
  putStrLn ( "L'offset est " ++ (show offset))

> main
L'offset est 3000
```

```
{-
get :: State s s
put :: s -> State s ()
runState :: State s a -> s -> (a, s)

-}
```

First Attempted Escape From Silence : Tunnels

Karim Haddad



1er acte de l'opera " Seven Attempted Escapes from Silence"

Livret : Jonathan Safran Foer

Soprano : Anna Prohaska, Ksenija Lukic

Ténor : Noriyuki Sawabu

Baryton : Nicholas Isherwood

Récitante : Kate Strong

Orchesterakademie de la Staatskapelle de Berlin

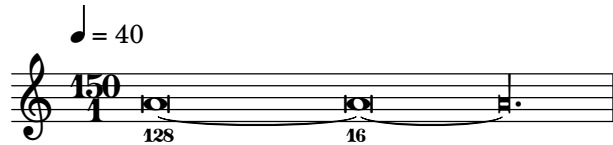
Direction : Max Renne

Mise-en-scène : Eszter Salamon

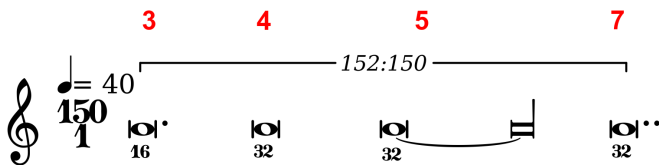
Création : Staatsoper Berlin

First Attempted Escape From Silence : Tunnels

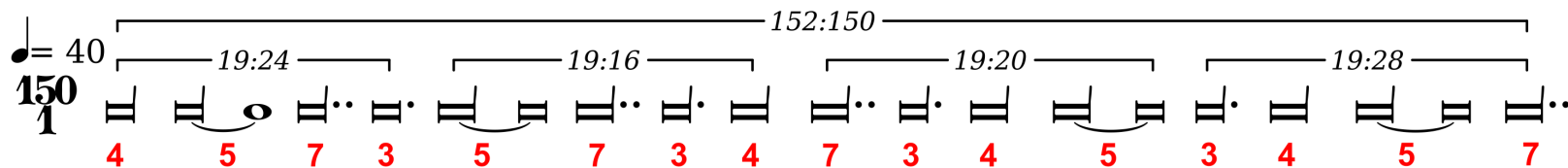
Karim Haddad



A partir de ce bloc de temps, on construit trois autres voix par rotation de chaque sous-groupe :



4	5	7	3
5	7	3	4
7	3	4	5



First Attempted Escape From Silence : Tunnels

Karim Haddad

Third voice:

19:24 19 38:150 19:20 19:14

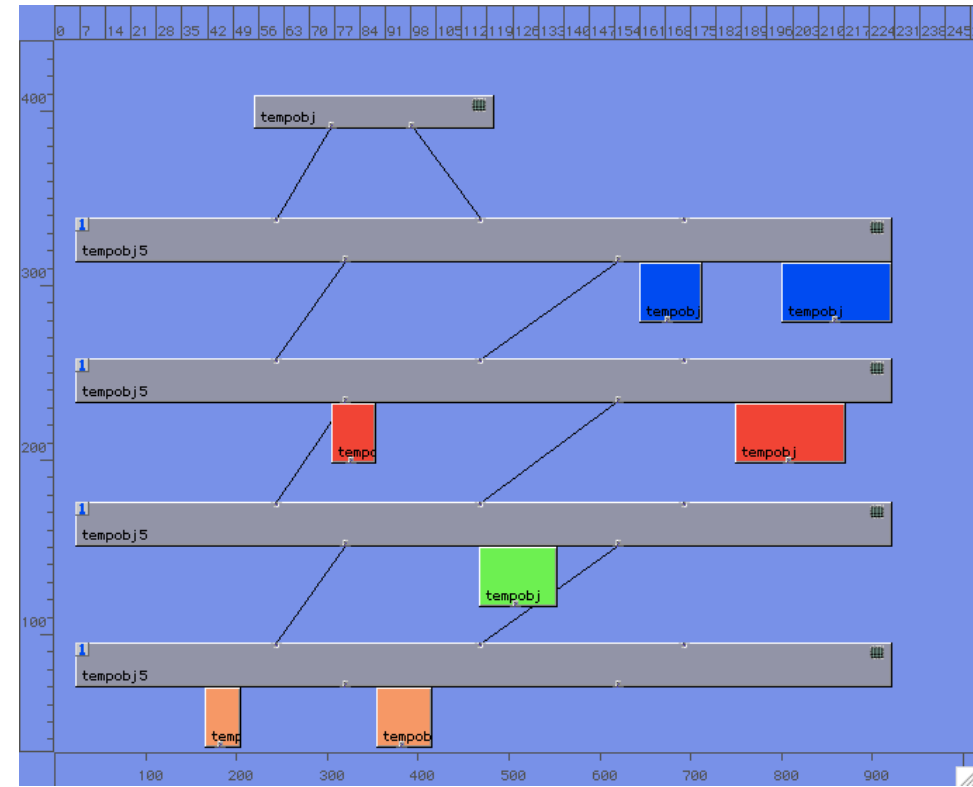
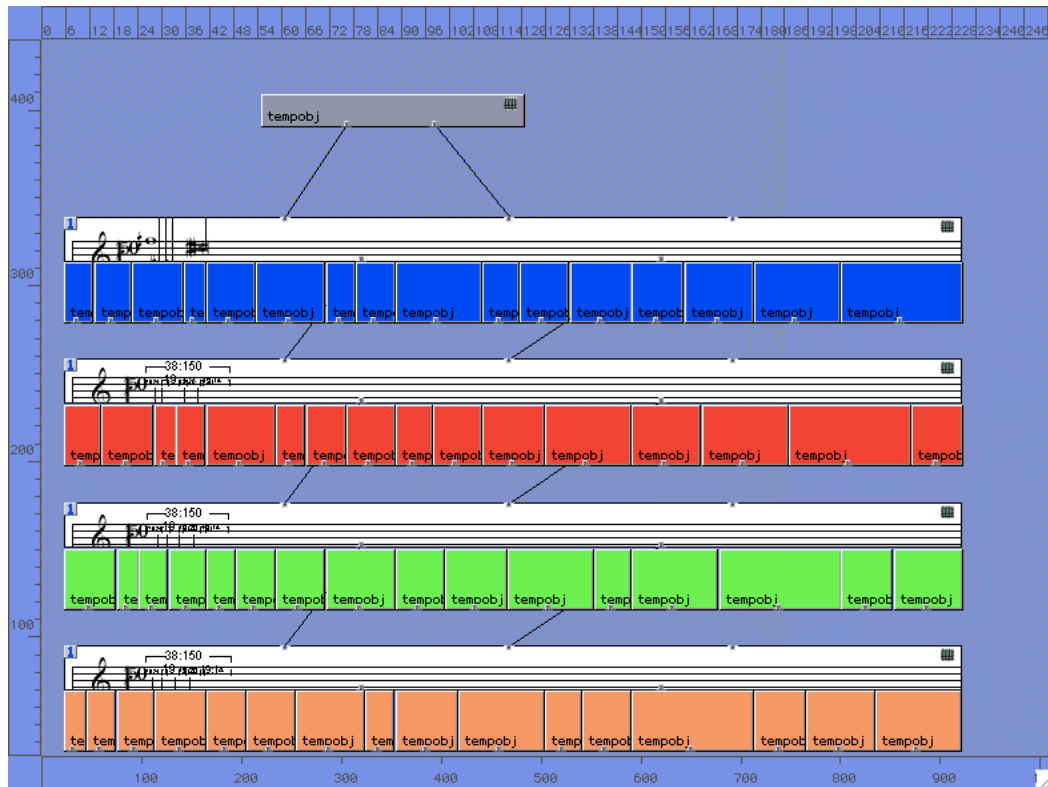
Vocal quartet:

52:50 361 72:2625

Un exemple montrant l'une des opérations homothétiques.

First Attempted Escape From Silence : Tunnels

Karim Haddad



Par de C. Claire jusqu'à 141.1.

INTRO

189.5 236.8 331.6

10-11-12 Solo

babel

Solo 2 Tenor Babel

4 solo

1st solo -

Solo 1

duo

2 voices

Ten Solo

4 voices

Solo hands.

Solo

Démo OM->

SD

39

2

2

58

2

2

2

9

9

pa
mf sempre

DE _____
mf SCHNITZ

$$(pp)$$

(pp)

 $\pi f_{\text{sub.}} \text{ sost.}$ $\overline{m}f_{sub}$ cost. $\frac{1}{m} f_{\text{sub.}} \quad \text{cost.}$ $\overline{mf}_{sub.}$ syst.

P

P

P

2

 $\mathcal{P}_{\text{stab}}$

P such

Références

- « All About Monads » by Jeff Newbern.
https://wiki.haskell.org/All_About_Monads#Doing_it_with_class
- « Category Theory for Programmers » by Bartosz Milewski
<https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface>