Cursul 13-14

- Introducere în Teoria Categoriilor
- Conceptele din teoria categoriilor traduse în Haskell

Introducere in teoria categoriilor

Samuel Eilenberg (1913-1998)

Saunders Mac Lane (1909-2005)

https://en.wikipedia.org/wiki/Category_theory

- Samuel Eilenberg and Saunders Mac Lane introduced the concepts of categories, functors, and natural transformations in 1942–45 in their study of algebraic topology, with the goal of understanding the processes that preserve mathematical structure, and influenced by previous related ideas by Polish and German mathematicians. Category theory has practical applications in programming language theory, in particular for the study of monads in functional programming.

Categories in theory

1.1 Definition A **category** $\mathcal{C} = (\mathcal{O}_\mathcal{C}, \mathcal{M}_\mathcal{C}, \mathcal{T}_\mathcal{C}, \circ_\mathcal{C}, \mathrm{Id}_\mathcal{C})$ is a structure consisting of *morphisms* $\mathcal{M}_\mathcal{C}$, *objects* $\mathcal{O}_\mathcal{C}$, a *composition* $\circ_\mathcal{C}$ of morphisms, and *type information* $\mathcal{T}_\mathcal{C}$ of the morphisms, which obey to the following constraints.

Note, that we often omit the subscription with $\mathcal{C}$ if the category is clear from the context.

1. We assume the collection of **objects** $\mathcal{O}$ to be a set. Quite often the objects themselves also are sets, however, category theory makes *no* assumption about that, and provides no means to explore their structure.
2. The collection of **morphisms** $\mathcal{M}$ is also assumed to be a set in this guide.
3. The ternary relation $\mathcal{T} \subseteq \mathcal{M} \times \mathcal{O} \times \mathcal{O}$ is called the **type information** of $\mathcal{C}$. We want every morphism to have a type, so a category requires

   $$\forall\, f \in \mathcal{M} \quad \exists\, A, B \in \mathcal{O} \quad (f, A, B) \in \mathcal{T} \ .$$

   We write $f : A \xrightarrow[\mathcal{C}]{} B$ for $(f, A, B) \in \mathcal{T}_\mathcal{C}$. Also, we want the types to be unique, leading to the claim

   $$f : A \to B \wedge f : A' \to B' \ \Rightarrow \ A = A' \wedge B = B' \ .$$

   This gives a notion of having a morphism to "map from one object to another". The uniqueness entitles us to give names to the objects involved in a morphism. For $f : A \to B$ we call $\mathrm{src}\, f := A$ the **source**, and $\mathrm{tgt}\, f := B$ the **target** of f.
4. The *partial* function $\circ : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, written as a binary infix operator, is called the **composition** of $\mathcal{C}$. Alternative notations are $f\,;g := gf := g \circ f$.

Categories in theory

Morphisms can be composed whenever the target of the first equals the source of the second. Then the resulting morphism maps from the source of the first to the target of the second:

$$f : A \to B \wedge g : B \to C \;\Rightarrow\; g \circ f : A \to C$$

In the following, the notation $g \circ f$ implies these type constraints to be fulfilled.
Composition is associative, i.e. $f \circ (g \circ h) = (f \circ g) \circ h$.

5. Each object $A \in \mathcal{O}$ has associated a unique **identity morphism** id_A. This is denoted by defining the function

$$\begin{array}{rccc} \mathrm{Id} & : & \mathcal{O} & \longrightarrow & \mathcal{M} \\ & & A & \longmapsto & \mathrm{id}_A \end{array} ,$$

which automatically implies uniqueness.
The typing of the identity morphisms adheres to

$$\forall\, A \in \mathcal{O} \quad \mathrm{id}_A : A \to A \ .$$

To deserve their name, the identity morphisms are —depending on the types— left or right neutral with respect to composition, i.e.,

$$f \circ \mathrm{id}_{\mathrm{src}\, f} = f = \mathrm{id}_{\mathrm{tgt}\, f} \circ f \ .$$

Spot a category in Haskell

With the last section in mind, where would you look for "the obvious" category? It is not required that it models the whole Haskell language, instead it is enough to point at the things in Haskell that behave like the objects and morphisms of a category.

We call our Haskell category $\mathcal{H}$ and use Haskell's types —primitive as well as constructed— as the objects $\mathcal{O}_{\mathcal{H}}$ of the category. Then, unary Haskell functions correspond to the morphisms $\mathcal{M}_{\mathcal{H}}$, with function signatures of unary functions corresponding to the type information $\mathcal{T}_{\mathcal{H}}$.

`f :: A -> B` corresponds to $f : A \xrightarrow[\mathcal{H}]{} B$

Haskell's function composition '`.`' corresponds to the composition of morphisms $\circ_{\mathcal{H}}$. The identity in Haskell is typed

```
id :: forall a. a -> a
```

corresponding to

$$\forall\, A \in \mathcal{O}_{\mathcal{H}} \quad \mathrm{id}_A : A \to A \ .$$

Note that we do not talk about n-ary functions for $n \neq 1$. You can consider a function like $(+)$ to map a number to a function that adds this number, a technique called **currying** which is widely used by Haskell programmers. Within the context of this guide, we do not treat the resulting function as a morphism, but as an object.

Functors in theory

4.1.1 Definition Let $\mathcal{A}, \mathcal{B}$ be categories. Then two mappings

$$F_{\mathcal{O}} : \mathcal{O}_{\mathcal{A}} \to \mathcal{O}_{\mathcal{B}} \quad \text{and} \quad F_{\mathcal{M}} : \mathcal{M}_{\mathcal{A}} \to \mathcal{M}_{\mathcal{B}}$$

together form a **functor** F from $\mathcal{A}$ to $\mathcal{B}$, written $F : \mathcal{A} \to \mathcal{B}$, iff

1. they preserve type information, i.e.,
$$\forall\, f : A \underset{\mathcal{A}}{\longrightarrow} B \quad F_{\mathcal{M}} f : F_{\mathcal{O}} A \underset{\mathcal{B}}{\longrightarrow} F_{\mathcal{O}} B \ ,$$
2. $F_{\mathcal{M}}$ maps identities to identities, i.e.,
$$\forall\, A \in \mathcal{O}_{\mathcal{A}} \quad F_{\mathcal{M}}\, \mathrm{id}_A = \mathrm{id}_{F_{\mathcal{O}} A} \ ,$$
3. and application of $F_{\mathcal{M}}$ distributes under composition of morphisms, i.e.,
$$\forall\, f : A \underset{\mathcal{A}}{\longrightarrow} B ;\ g : B \underset{\mathcal{A}}{\longrightarrow} C \quad F_{\mathcal{M}}(g \circ_{\mathcal{A}} f) = F_{\mathcal{M}} g \circ_{\mathcal{B}} F_{\mathcal{M}} f \ .$$

4.1.2 Notation Unless it is required to refer to only one of the mappings, the subscripts $_{\mathcal{M}}$ and $_{\mathcal{O}}$ are usually omitted. It is clear from the context whether an object or a morphism is mapped by the functor.

4.1.3 Definition Let $\mathcal{A}$ be a category. A functor $F : \mathcal{A} \to \mathcal{A}$ is called **endofunctor**.

Functors in Haskell

Following our idea of a Haskell category $\mathcal{H}$, we can define an endofunctor $F : \mathcal{H} \to \mathcal{H}$ by giving a unary type constructor `F`, and a function `fmap`, constituting $F_{\mathcal{O}}$ and $F_{\mathcal{M}}$ respectively.

The type constructor `F` is used to construct new types from existing ones. This action corresponds to mapping objects to objects in the $\mathcal{H}$ category. The definition of `F` shows how a functor implements the structure of the constructed type. The function `fmap` lifts each function with a signature $f : A \to B$ to a function with signature $Ff : FA \to FB$.

Hence, functor application on an object is a *type level* operation in Haskell, while functor application on a morphism is a *value level* operation.

Haskell comes with the definition of a class

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

which allows overloading of `fmap` for each functor.

Exemple: Maybe, [ ]

The mapping function `fmap` differs from functor to functor. For the Maybe structure, it can be written as

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

while `fmap` for the List structure can be written as

```
fmap :: (a -> b) -> [a] -> [b]
fmap f [] = []
fmap f (x:xs) = (f x):(fmap f xs)
```

Note, however, that having a type being a member of the `Functor` class does not imply to have a functor at all. Haskell does not check validity of the functor properties, so we have to do it by hand:

1. $f : A \to B \;\Rightarrow\; Ff : FA \to FB$ is fulfilled, since being a member of the Functor class implies that a function `fmap` with the according signature is defined.

2. $\forall\, A \in \mathcal{O} \quad F\,\mathrm{id}_A = \mathrm{id}_{FA}$ translates to

   ```
   fmap id == id
   ```

   which must be checked for each type one adds to the class. Note, that Haskell overloads the `id` function: The left occurrence corresponds to id_A, while the right one corresponds to id_{FA}.

3. $F(g \circ f) = Fg \circ Ff$ translates to

   ```
   fmap (g . f) == fmap g . fmap f
   ```

   which has to be checked, generalising over all functions `g` and `f` of appropriate type.

Maybe and List are both functors.

For the Maybe structure, the data constructors are `Just` and `Nothing`. So we prove the second functor property, $F\,\mathrm{id}_A = \mathrm{id}_{FA}$, by

```
fmap id Nothing
  == Nothing
  == id Nothing

fmap id (Just y)
  == Just (id y)
  == Just y
  == id (Just y) ,
```

and the third property, $F(g \circ f) = Fg \circ Ff$, by

```
(fmap g . fmap h) Nothing
  == fmap g (fmap h Nothing)
  == fmap g Nothing
  == Nothing
  == fmap (g . h) Nothing ,

(fmap g . fmap h) (Just y)
  == fmap g (fmap h (Just y))
  == fmap g (Just (h y))
  == Just (g (h y))
  == Just ((g . h) y)
  == fmap (g . h) (Just y)  .
```

Maybe and List are both functors.

Note, that List is —in contrast to Maybe— a recursive structure. This can be observed in the according definition of `fmap` above, and it urges us to use induction in the proof: The second property, $F\,\mathrm{id}_A = \mathrm{id}_{FA}$, is shown by

```
fmap id []
  == []
  == id []

fmap id (x:xs)
  == (id x):(fmap id xs)
  == (id x):(id xs)  --here we use induction
  == x:xs
  == id (x:xs) ,
```

and the third property, $F(g \circ f) = Fg \circ Ff$ can be observed in

```
fmap (g . f) []
  == []
  == fmap g []
  == fmap g (fmap f [])
  == (fmap g . fmap f) []
```

Maybe and List are both functors.

```
fmap (g . f) (x:xs)
  == ((g . f) x):(fmap (g . f) xs)
  == ((g . f) x):((fmap g . fmap f) xs)  --induction
  == (g (f x)):(fmap g (fmap f xs))
  == fmap g ((f x):(fmap f xs))
  == fmap g (fmap f xs)
  == (fmap g . fmap f) xs  .
```

Natural transformations in theory

5.1.1 Definition Let $\mathcal{A}, \mathcal{B}$ be categories, $F, G : \mathcal{A} \to \mathcal{B}$. Then, a function

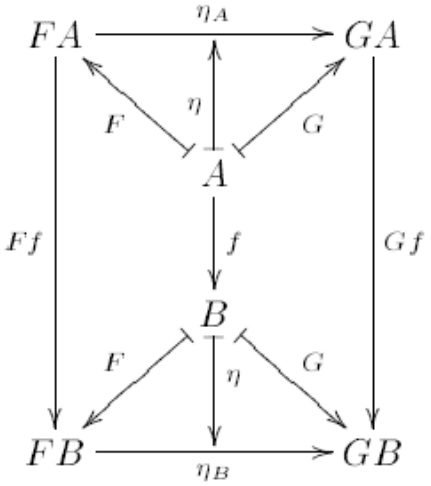
$$\begin{array}{rccc} \eta & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\ & & A & \longmapsto & \eta_A \end{array}$$

is called a **transformation** from F to G, iff

$$\forall\, A \in \mathcal{O}_{\mathcal{A}} \quad \eta_A : FA \underset{\mathcal{B}}{\longrightarrow} GA \ ,$$

and it is called a **natural** transformation, denoted $\eta : F \stackrel{\cdot}{\longrightarrow} G$, iff

$$\forall\, f : A \underset{\mathcal{A}}{\longrightarrow} B \quad \eta_B \circ_{\mathcal{B}} Ff = Gf \circ_{\mathcal{B}} \eta_A \ .$$



The definition says, that a natural transformation *transforms* from a structure F to a structure G, without altering the behaviour of morphisms on objects. I.e., it does not play a role whether a morphism f is lifted into the one or the other structure, when composed with the transformation.
In the drawing on the left, this means that the outer square *commutes*, i.e., all directed paths with common source and target are equal.

Natural transformations in Haskell

First note, that a unary function polymorphic in the same type on source and target side, in fact is a transformation. For example, Haskell's `Just` is typed

```
Just :: forall a. a -> Maybe a  .
```

If we imagine this to be a mapping $\texttt{Just} : \mathcal{O}_{\mathcal{H}} \to \mathcal{M}_{\mathcal{H}}$, the application on an object $A \in \mathcal{O}_{\mathcal{H}}$ means binding the type variable a to some Haskell type `A`. That is, `Just` maps an object A to a morphism of type $A \to \texttt{Maybe}\ A$.

To spot a transformation here, we still miss a functor on the source side of `Just`'s type. This is overcome by adding the identity functor $I_{\mathcal{H}}$, which leads to the transformation

$$\texttt{Just} : I_{\mathcal{H}} \to \texttt{Maybe}\ .$$

5.2.1 Lemma $\texttt{Just} : I_{\mathcal{H}} \dot{\longrightarrow} \texttt{Maybe}$.

5.2.2 Proof Let `f::A->B` be an arbitrary Haskell function. Then we have to prove, that

```
Just . I_H f == fmap f . Just
```

where the left `Just` refers to η_B, and the right one refers to η_A. In that line, we recognise the definition of the `fmap` for Maybe (just drop the $I_{\mathcal{H}}$). □

Another example, this time employing two non-trivial functors, are the `maybeToList` and `listToMaybe` functions. Their definitions read

```
maybeToList :: forall a. Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just x) = [x]

listToMaybe :: forall a. [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (x:xs) = Just x .
```

Note, that the *loss of information* imposed by applying `listToMaybe` on a list with more than one element does not contradict naturality, i.e., *forgetting* data is not *altering* data.

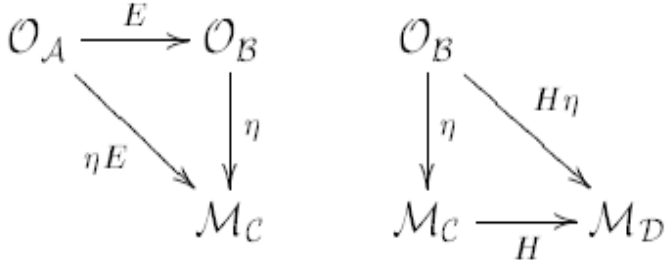
Composing transformations and functors

5.3.1 Definition Let $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ be categories, $E : \mathcal{A} \to \mathcal{B}$, $F, G : \mathcal{B} \to \mathcal{C}$, and $H : \mathcal{C} \to \mathcal{D}$. Then, for a natural transformation $\eta : F \dot{\to} G$, we define the transformations ηE and $H\eta$ with

$$(\eta E)A := \eta_{EA} \quad \text{and} \quad (H\eta)B := H(\eta_B) \ ,$$

where A varies over $\mathcal{O}_{\mathcal{A}}$, and B varies over $\mathcal{O}_{\mathcal{B}}$.

Again, due to the above definition, we can write ηEA and $H\eta B$ (for A and B objects in the respective category) without ambiguity. The following pictures show the situation:



5.3.2 Lemma In the situation of Definition 5.3.1,

$$H\eta : HF \dot{\to} HG \quad \text{and} \quad \eta E : FE \dot{\to} GE$$

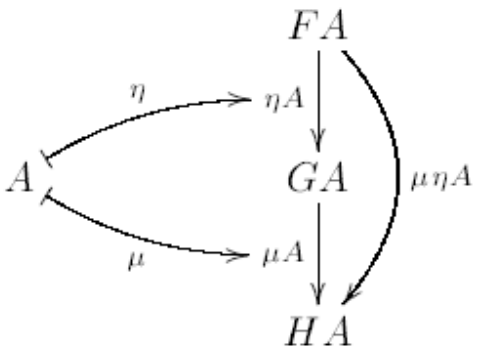
hold. In (other) words, $H\eta$ and ηE are both natural transformations.

Composing transformations and functors

Composing natural transformations Even natural transformations can be composed with each other. Since, however, transformations map objects to morphisms —instead of, e.g., objects to objects— they can not be simply applied one after another. Instead, composition is defined *component wise*, also called **vertical composition**.

5.3.4 Definition Let $\mathcal{A}, \mathcal{B}$ be categories, $F, G, H : \mathcal{A} \to \mathcal{B}$ and $\eta : F \dot{\longrightarrow} G$, $\mu : G \dot{\longrightarrow} H$. Then we define the transformations

$$\begin{array}{rcccc} \mathrm{id}_F & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\ & & A & \longmapsto & \mathrm{id}_{FA} \end{array} \quad \text{and} \quad \begin{array}{rcccc} \mu\eta & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\ & & A & \longmapsto & \mu A \circ \eta A \end{array} .$$



The picture on the left (with $A \in \mathcal{O}_{\mathcal{A}}$) clarifies why this composition is called *vertical*.
All compositions defined before Definition 5.3.4 are called **horizontal**. ✎ Why?

Monads in theory

6.1.1 Definition Let $\mathcal{C}$ be a category, and $F : \mathcal{C} \to \mathcal{C}$. Consider two natural transformations $\eta : I_{\mathcal{C}} \dot{\to} F$ and $\mu : F^2 \dot{\to} F$.

The triple (F, η, μ) is called a **monad**, iff

$$\mu(F\mu) = \mu(\mu F) \quad \text{and} \quad \mu(F\eta) = \mathrm{id}_F = \mu(\eta F) \ .$$

(Mind, that the parenthesis group composition of natural transformations and functors. They do not refer to function application. This is obvious due to the types of the expressions in question.)

The transformations η and μ are somewhat contrary: While η adds one level of structure (i.e., functor application), μ removes one. Note, however, that η transforms to F, while μ transforms from F^2. This is due to the fact that claiming the existence of a transformation from a functor F to the identity $I_{\mathcal{C}}$ would be too restrictive:

Consider the set category $\mathcal{S}$ and the list endofunctor L. Any transformation ϵ from L to $I_{\mathcal{S}}$ has to map every object A to a morphism ϵ_A, which in turn maps the empty list to some element in A, i.e.,

$$\begin{array}{ll} & \epsilon : L \to I_{\mathcal{S}} \\ \Rightarrow & \\ & \forall\, A \in \mathcal{O}_{\mathcal{S}} \quad \epsilon_A : LA \to A \\ \Rightarrow & \\ & \forall\, A \in \mathcal{O}_{\mathcal{S}} \quad \exists\, c \in A \quad \epsilon_A[\,] = c \ , \end{array}$$

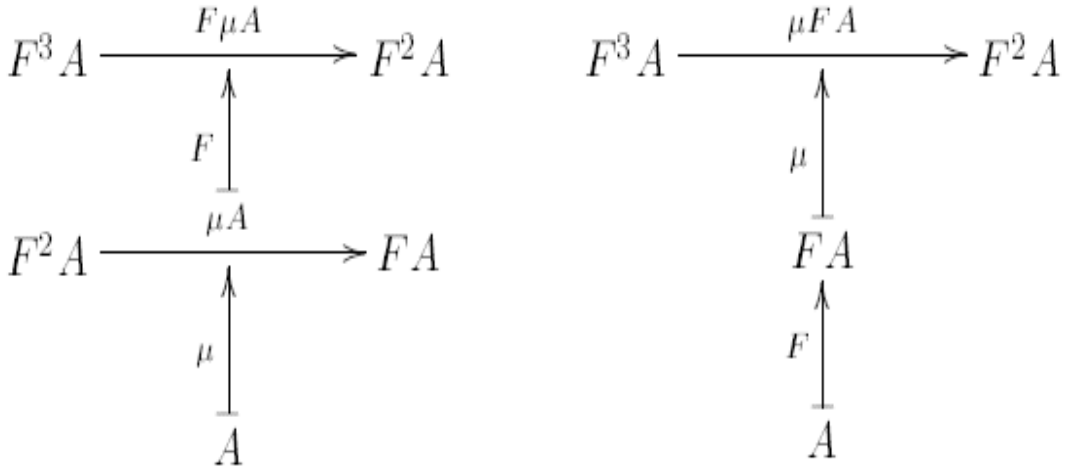
where $[\,]$ denotes the empty list. This, however, implies

$$\forall\, A \in \mathcal{O}_{\mathcal{S}} \quad A \neq \emptyset \ .$$

The following drawings are intended to clarify Definition 6.1.1: The first equation of the definition is equivalent to

$$\forall\, A \in \mathcal{O} \quad \mu A \circ F\mu A = \mu A \circ \mu F A \ .$$

So what are $F\mu A$ and μFA? You can find them at the top of these drawings:



Since application of μA unnests a nested structure by one level, $F\mu A$ pushes application of this unnesting one level into the nested structure. We need at least two levels of nesting to apply μA.

So we need at least three levels of nesting to push the application of μA one level in. This justifies the type of $F\mu A$.

The morphism μFA, however, applies the reduction on the outer level. The FA only assures that there is one more level on the inside which, however, is not touched.

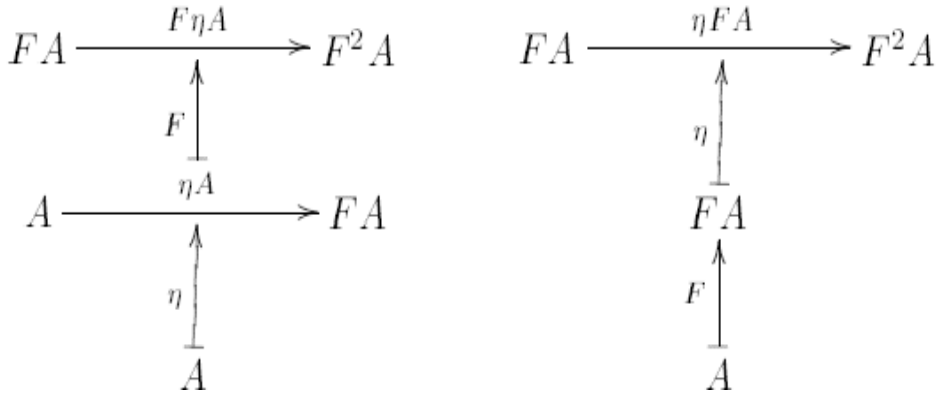
Hence, $F\mu A$ and μFA depict the two possibilities to flatten a three-level nested structure into a two-level nested structure through application of a mapping that flattens a two-level nested structure into an one-level nested structure.

The statement $\mu A \circ F\mu A = \mu A \circ \mu FA$ says, that after another step of unnesting (i.e., μA), it is irrelevant which of the two inner structure levels has been removed by prior unnesting (i.e., $F\mu A$ or μFA).

Let us have a look at the second equation as well. It is equivalent to

$$\forall\, A \in \mathcal{O}_C \quad \mu A \circ F\eta A = \mathrm{id}_{FA} = \mu A \circ \eta FA \ .$$

Again, we examine $F\eta A$ and ηFA, which are at the top of the drawings.



MONADE

- Conceptele din teoria categoriilor traduse în Haskell
- Clasa Functor
- Clasa Monad
- Legi ale monadei
- Modulul Monad
- Studiu de caz

Categoria **Hask**

- Obiectele sunt tipurile din Haskell
- Morfismele sunt funcțiile din Haskell văzute ca funții de o singură variabilă
- Constructorii de tip sunt aplicații ce transformă un tip în alt tip
- Funcții de ordin înalt: transformă o funcție în altă funcție

Functori în Hask

- Functorii în Haskell sunt aplicații de la **Hask** la *func,* unde *func este subcategoria lui* **Hask** definită pe tipurile acestui functor
  - Functorul listă este un functor de la **Hask** la **Lst**

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

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

Maybe este functor

- Transformă orice tip*T* într-un nou tip*, Maybe T*
- Restricția lui fmap la tipurile Maybe transformă o funcție a -> b într-o funcție Maybe a -> Maybe b.
  - fmap id = id
  - fmap (f . g) = fmap f . fmap g

Monade

- O *monadă* este un tip special de functor care suportă o structură adițională. Orice monadă este un functor de la o categorie la ea însăși
- O monadă este un functor M : *C* -> *C*, împreună cu 2 morfisme atașate fiecărui obiect *X din C:*
  - $unit^M_X : X \rightarrow M(X)$
  - $join^M_X : M(M(X)) \rightarrow M(X)$

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

Monade

- Se poate defini join și (>>=) unul din altul:

```
join :: Monad m => m (m a) -> m a
join x = x >>= id
(>>=) :: Monad m => m a -> (a -> m b) -> m b
x >>= f = join (fmap f x)
```

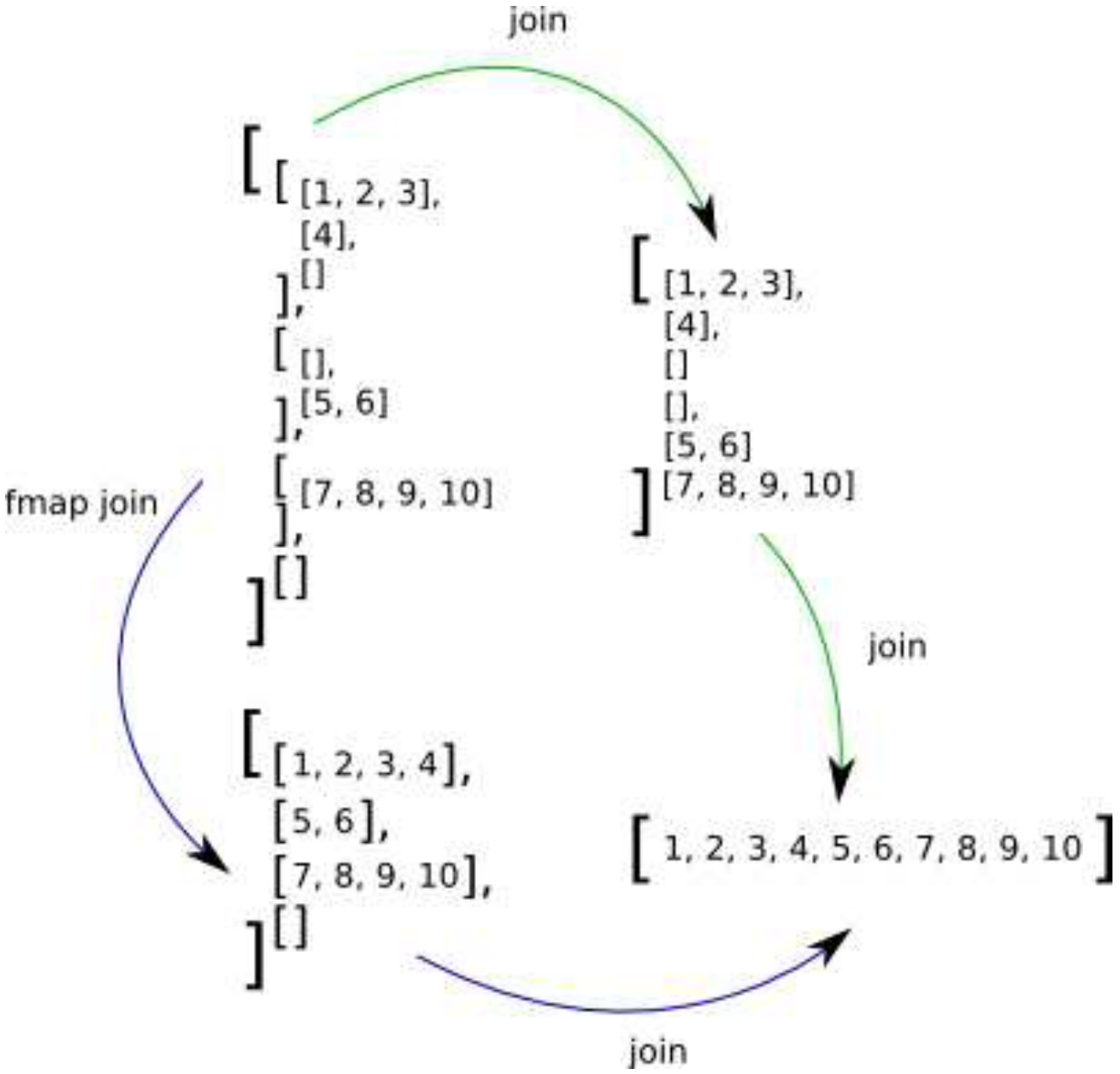
- Exemplu: functorul powerset *P* : **Set** -> **Set** este o monadă. Pentru orice mulțime *S:*
  - $unit_S(x) = \{x\}$,
  - $join_S(L) = \cup L$
- *P* seamănă cu monada listă

Legi ale monadei

- Dată o monadă M : *C* -> *C* și un morfism f : A -> B unde A, B sunt obiecte în *C*
  - join o M(join) = join o join
  - join o M(unit) = join o unit = id
  - unit o f = M(f) o unit
  - join o M(M(f)) = M(f) o join

- În Haskell:

```
1. join . fmap . join = join . join
2. join . fmap return = join . return = id
3. return . f = fmap f . return
4. join . fmap (fmap f) = fmap f . join
```


join
[[[1, 2, 3], [4], []], [[], [5, 6]], [[7, 8, 9, 10]], []]
[[1, 2, 3], [4], [], [], [5, 6] [7, 8, 9, 10]]
fmap join
join
[[1, 2, 3, 4], [5, 6], [7, 8, 9, 10], []]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
join

Clasa Monad în Haskell

```
Prelude> :i Monad
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
       -- Defined in GHC.Base
instance Monad Maybe -- Defined in Data.Maybe
instance Monad IO -- Defined in GHC.IOBase
instance Monad [] -- Defined in GHC.Base
```

```
Prelude> :i Functor
class Functor f where fmap :: (a -> b) -> f a -> f b
        -- Defined in GHC.Base
instance Functor Maybe -- Defined in Data.Maybe
instance Functor IO -- Defined in GHC.IOBase
instance Functor [] -- Defined in GHC.Base
```

```
Prelude> :i []
data [] a = [] | a : [a]      -- Defined in GHC.Types
instance (Eq a) => Eq [a] -- Defined in GHC.Base
instance Monad [] -- Defined in GHC.Base
instance Functor [] -- Defined in GHC.Base
instance (Ord a) => Ord [a] -- Defined in GHC.Base
instance (Read a) => Read [a] -- Defined in GHC.Read
instance (Show a) => Show [a] -- Defined in GHC.Show
```

Legi ale monadei

- Operațiile unei monade trebuie sa îndeplinească trei legi fundamentale:
  - **`return`** este element neutru la stânga pentru **`>>=`**

    `(return x) >>= f == f x`
  - **`return`** este element neutru la dreapta:

    `m >>= return == m`
  - operația **`>>=`** este asociativă:

    `(m >>= f) >>= g == m >>= (\x -> f x >>= g)`
- Orice constructor de tip împreună cu **`return`** și **`>>=`** ce îndeplinește aceste legi este o monadă; programatorul are sarcina să asigure acest lucru la definirea unei noi monade
- Clasa Monad mai are definite două funcții: **`fail`** și **`>>`**

Variante monadă a unor funcții pe liste

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
foldM f a [] = return a
foldM f a (x:xs) = f a x >>= \y -> foldM f y xs
```

Studiu de caz:
evaluator monadic pentru împărțire

- Expresii cu operatorul de împărțire:

```
data Term = Con Int | Div Term Term
          deriving Show

e1, e2 :: Term
e1 = Div(Div(Con 1972)(Con 2))(Con 23)
e2 = Div(Con 2)(Div(Con 1)(Con 0))
```

Evaluator monadic pentru împărțire

- Evaluatorul eval: dacă m este o monadă atunci eval primeşte la intrare un term și realizează un calcul ce conduce la un întreg astfel:
  - evaluarea lui Con x înseamnă return x
  - evaluarea lui Div t u înseamnă:
    - evaluarea lui t şi legarea lui x cu valoarea lui t
    - evaluarea lui u şi legarea lui y cu valoarea lui u
    - return x `div` y

```
eval :: Monad m => Term -> m Int
eval(Con x) = return x
eval(Div t u) = do x <-eval t
                   y <-eval u
                   return (x `div` y)
```

Evaluare fără semnalare excepții

- Evaluatorul evalId: specializarea lui eval pentru m = Id
- Monada Id este declarată pentru tipul Id a care este izomorf cu a:
  - return este izomorf cu funcția identitate
  - **>>=** este izomorf cu funcția aplicație

```
newtype Id a = MkId a
instance Monad Id where
                return x = MkId x
                (MkId x) >>= q = q x
```

- Pentru a specifica modul de afişare:

```
instance Show a => Show(Id a) where
show(MkId x) = "valoare exp: " ++ show x
```

- Evaluatorul de bază:

```
evalId :: Term -> Id Int
evalId = eval
```

- Exemple:

```
Main> evalId e1
valoare exp: 42
Main> evalId e2
valoare exp:
Program error: divide by zero
```

Evaluare cu semnalarea excepțiilor

- Tipul Exc a al excepțiilor pentru tipul a:

```
data Exc a = Raise Exception | Return a
type Exception = String
```

- Monada Exc:

```
instance Monad Exc where
      return x = Return x
      (Raise e) >>= q = Raise e
      (Return x) >>= q = q x
      raise:: Exception -> Exc a
      raise e = Raise e
```

Evaluare cu semnalarea excepțiilor

```
evalEx :: Term -> Exc Int
evalEx(Con x) = return x
evalEx(Div t u) = do   x <- evalEx t
                       y <- evalEx u
                       if y == 0
                              then raise "impartire prin zero\n"
                              else return (x `div` y)

instance Show a => Show(Exc a) where
show(Raise e) = "exceptie: " ++ e
show(Return x) = "valoare exp: " ++ show x

Main> evalEx e1
valoare exp: 42
Main> evalEx e2
exceptie: impartire prin zero
```