Cursul 11 – 12 MONADE

- Monade
  - Introducere
  - Tipul de data Maybe
  - Constructori de tip
  - Monada în Haskell
    - Exemplul 1: monada Maybe
    - Exemplul 2: monada []
  - Clasa Monad
  - Notația do
  - Monada IO
  - Exemple
- Introducere în teoria categoriilor

“Side Effects” în Haskell

- Programele de până acum nu au “side-effects”: programele sunt executate pentru a afla nişte valori care se afişează.
- Cum construim programe interactive? (citirea unor valori, afișare, lucru cu fişiere, desenarea unei figuri etc.)

- În Haskell, “valorile pure” sunt separate de “acțiuni”, în două moduri:
  - **Tipuri**: O expresie de tip `IO a` are asociate execuției sale eventuale *acțiuni* , ultima dintre ele fiind returnarea unei valori de tip `a`.
  - **Sintaxă**: Construcția sintactică `do` realizează o acțiune care, în general, este secvență de acțiuni “elementare”.

Introducere

- Monada – o cale de a structura calculele ca valori și secvențe de calcule ce utilizează aceste valori
  - Valorile monadice sunt numite calcule

- Permit construirea de calcule folosind blocuri secvențiale care, la rândul lor pot fi secvențe de calcul

- Monada determină modul în care o combinație de calcule formează un nou calcul, scutindu-l pe programator de a scrie cod pentru astfel de combinații

- O monadă este văzută ca o strategie pentru combinarea calculelor într-un calcul complex

Tipul de data Maybe

- Tipul "calcul care poate eșua sau poate returna o valoare":

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

```
mydiv:: Float -> Float -> Maybe Float
mydiv x 0 = Nothing
mydiv x y = Just (x/y)
```

- Tipul Maybe sugerează o strategie de combinare a calculelor: daca un calcul compus constă dintr-un calcul B care depinde de rezultatul altui calcul A, atunci calculul combinat va produce `Nothing` daca A sau B produce `Nothing`; altfel, dacă ambele se produc cu succes, calculul combinat va produce rezultatul calculului B, calcul care este aplicat rezultatului calculului A

Proprietăți fundamentale

- Modularitate
  - calcule (complexe) compuse din calcule simple
  - separarea strategiilor de combinare a calculelor de calculele în sine
- Flexibilitate
  - Programele scrise cu monade sunt mai adaptabile decât cele fără monade
- Izolare
  - monadele permit crearea de structuri de calcul în stil imperativ (sistemul I/O de ex.) care rămân izolate de corpul principal al programului funcțional

Constructori de tip

- Un constructor de tip este o definiție de tip parametrizat ce folosește tipuri polimorfe
- In definiția tipului Maybe

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

Maybe este un constructor de tip iar **Nothing, Just** sunt constructori de date (de tipul Maybe)

- Se poate construi o data aplicând constructorul de data **Just** unei valori:

```
tara = Just "Romania"
```

- Se poate construi un tip aplicând constructorul de tip **Maybe** unui tip concret:

```
Maybe Int, Maybe String
```

- Tipurile polimorfe pot fi privite ca niște containere ce sunt capabile să conțină valori de diverse tipuri:
  - **`Maybe String`** ar putea fi privit ca un container **`Maybe`** ce poate conține o valoare de tip **`String`** (sau **`Nothing`**)

- Se poate ca tipul containerului sa fie polimorf: **`m a`** reprezintă un container de un anumit tip m ce poate conține o valoare de un anumit tip a

- Se folosesc adesea variabile tip cu constructorii de tip pentru a descrie trăsăturile abstracte ale unui calcul: **`Maybe a`** este tipul tuturor calculelor care pot returna o valoare de tip **`a`** sau **`Nothing`**

Monada în Haskell

- O monadă în Haskell este dată prin:
  - un constructor de tip (notat m) care este numit constructorul de tip monadă
  - O funcție – constructor de tip, **`a -> m a`**, care construiește instanțe ale monadei, numită **`return`**
  - O funcție **`m a -> (a -> m b) -> m b`** care combină o instanță **`m a`** cu un calcul (funcție) ce produce dintr-o valoare de tip **`a`** o instanță **`m b`** a monadei și în final produce o nouă instanță **`m b`**. Această funcție este denumită **`bind`** și se scrie, ca operator, **`>>=`**

Monada in Haskell

- constructorul de tip monadă definește un tip de calcul
- funcția **`return`** creează valori primitive (instanțe) ale acestui tip
- operatorul (funcția) **`>>=`** combină calcule de acest tip pentru a obține calcule mai complexe de tipul respectiv
- Analogia cu containerul:
  - constructorul de tip m este un container ce poate conține diferite valori
  - m a este un container ce conține o valoare de tip a
  - funcția **`return`** pune o valoare în containerul monadă
  - Operatorul **`>>=`** :
    - ia valoarea din container
    - o transmite unei funcții
    - produce un nou container ce conține o noua valoare, posibil de alt tip
  - funcția **`>>=`** se cheamă **`bind`** pentru că ea leagă valoarea din containerul monadă cu primul argument al funcției

Acces la componentele unei date

- Tipul de dată Point:

```
data Point = Pt Float Float
let p = Pt 3.2 5.4
```

- Accesul la componente poate fi făcut prin funcții:

```
pointx :: Point -> Float
pointx (Pt x _) =  x
pointx p
```

- Mai comod declarația echivalentă:

```
data Point = Pt {pointx, pointy :: Float}
```

```
pointx ::   Point -> Float
pointy ::   Point -> Float
```

Acces la componentele unei date

```
data T    = C1 {f1,f2 :: Int}
          | C2 {f1 :: Int,
                f3,f4 :: Char}
```

Expresie	Traducere
`C1 {f1 = 3}`	`C1 3 undefined`
`C2 {f1 = 1, f4 = 'A', f3 = 'B'}`	`C2 1 'B' 'A'`
`x {f1 = 1}`	`case x of C1 _ f2 -> C1 1 f2` `C2 _ f3 f4 -> C2 1 f3 f4`

Exemplu: Monada **Maybe**

- Studiu de caz: determinarea strămoșilor: bunici, străbunici etc. din arborele genealogic
- Tipul de data Person:

```
data Person= Person {name::String,
      mother::Maybe Person, father::Maybe Person}
```

- Pentru a determina bunicul unei persoane am putea defini o funcție:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather s = case (mother s) of
                              Nothing -> Nothing
                              Just m  -> father m
```

Exemplu: Monada **Maybe**

- Un alt strămoș:

```
mothersPaternalGrandfather :: Person -> Maybe Person
mothersPaternalGrandfather s = case (mother s) of
                                 Nothing -> Nothing
                                 Just m -> case (father m) of
                                             Nothing -> Nothing
                                             Just gf -> father gf
```

- Soluția nu este eficientă
- Odată găsită valoarea **`Nothing`** într-un punct al calculului, aceasta rămâne **`Nothing`** ca rezultat final
- Sa încercăm să implementăm acest lucru o singură dată

Exemplu: Monada **Maybe**

- Crearea unui combinator ce înglobează acest aspect:

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

- Utilizarea acestui combinator pentru a construi secvențe mai complicate:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather s = (Just s) `comb` mother `comb` father

fathersMaternalGrandmother :: Person -> Maybe Person
fathersMaternalGrandmother s = (Just s) `comb` father `comb`
                               mother `comb` mother

mothersPaternalGrandfather :: Person -> Maybe Person
mothersPaternalGrandfather s = (Just s) `comb` mother `comb`
                               father `comb` father
```

Exemplu: Monada **`Maybe`**

- Combinatorul **`comb`**
  - Este o funcție polimorfă, nu este specializată pentru tipul Person
  - Capturează strategia generală de combinare a calculelor care pot, în particular, să eșueze
  - Poate fi folosit și în alte aplicații: interogare baze de date, căutare în dicționare, etc.

- Tipul **`Maybe`** împreună cu funcția **`Just`** (care acționează ca și **`return`**) și combinatorul **`comb`** (ce acționează ca și **`>>=`**) formează o monadă folosită pentru construirea de calcule ce pot eșua

Monada `[]`

- Constructorul de tip `[]` (pentru construirea listelor) este de asemenea o monadă:
  - Funcția return crează lista singleton

```
return x = [x]
```

  - Operația de legare pentru liste construește o noua listă conținând rezultatul aplicării funcției tuturor valorilor listei originale

```
l >>= f = concatMap f l
concatMap :: (a -> [b]) -> [a] -> [b]
```

Clasa **Monad**

- În Haskell există o clasă standard **Monad** care definește numele si signatura funcțiilor **return** și **>>=**

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

- Se recomandă ca monadele utilizator să fie instanțe ale clasei **Monad**

Exemplul 1 revizuit

- Monada Maybe este instanță a clasei Monad:

```
instance Monad Maybe where
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
    return = Just
```

- Se pot utiliza operatorii standard din Monad:

```
maternalGrandfather s = (return s) >>= mother >>= father

fathersMaternalGrandmother s = (return s) >>= father >>=
  mother >>= mother
```

Notația do

- Alternativa în a utiliza funcțiile monadice
- Similară cu utilizarea “list comprehensions“ pentru liste
- Scrierea calculelor monadice în stil pseudo-imperativ folosind variabile:
  - Rezultatul unui calcul monadic poate fi “asignat” unei variabile folosind operatorul `<-`
  - Utilizarea variabilei într-o subsecvență de calcul monadic, se face automat legarea
  - Tipul expresiei din dreapta semnului `<-` este tip monadic m a

Notatia do

```
mothersPaternalGrandfather s = do m <- mother s
                                  gf <- father m
                                  father gf
```

sau:

```
mothersPaternalGrandfather s =
      do {m <- mother s; gf <- father m; father gf}
```

- Monadele oferă prin notația do posibilitatea de a crea calcule în stil imperativ în cadrul programelor funcționale

- Notația do este doar “syntactic sugar”

Transformarea do în >>=

- Comanda **x <- expr1** devine:

  expr1 >>= \x ->
- Comanda **expr2** devine:

  expr2 >>= _ ->
- De exemplu:

```
mothersPaternalGrandfather s = do  m <- mother s
                                   gf <- father m
                                   father gf
```

devine

```
mothersPaternalGrandfather s = mother s >>= \m ->
                               father m >>= \gf ->
                               father gf
```

- Operatorul **>>=** (bind) leagă valoarea din monadă cu argumentul din următoarea lambda expresie

Monada IO

- Operațiile de intrare/ieșire sunt incompatibile cu stilul funcțional pur: efect secundar, modificarea valorilor
- Soluția: monada IO a
  - Calcule ce realizează I/O în cadrul monadei IO
  - Calculele în monada IO se numesc acțiuni de intrare /ieșire (*I/O actions*)

“Side Effects” în Haskell

- Un program interactiv este conceput ca o funcție care are ca argument ”state of the world” (o valoare de tip World) și produce o valoare de tip World care este o altă stare ce reflectă efectul colateral produs de program:

```
type IO = World -> World
```

- În general, un program interactiv poate returna și o anume valoare pe lângă efectul ce-l are asupra stării (de exemplu, se citește un caracter de la tastatură (acțiune) și se returnează acel caracter). Așadar, mai corect:

```
type IO a = World -> (a, World)
```

Exemplu: `IO Char, IO Int; IO ()`

- O expresie de tip `IO a` are asociate execuției sale *acțiuni,* ultima dintre ele fiind returnarea unei valori de tip `a`.

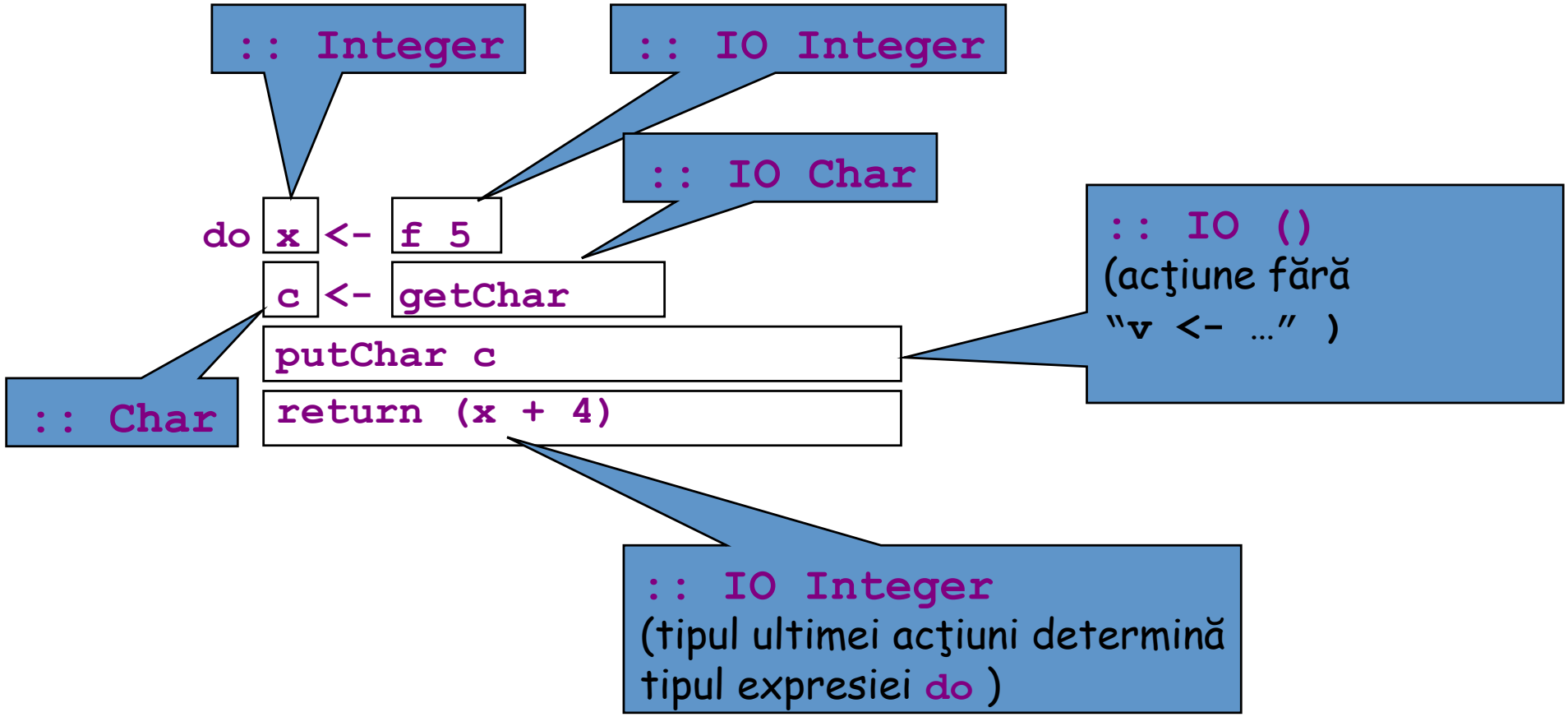
Construcția sintactică **do**

- Dacă **act** este o acțiune de tip **IO a** atunci putem descrie realizarea acțiunii **act**, returna valoarea corespunzătoare şi eventual adăugarea altor acțiuni, astfel:

```
do val <- act
   ...          -- actiunea urmatoare
   ...          -- actiunea urmatoare
   return x     -- actiunea finală
```

- Toate acțiunile de după **val <- act** pot folosi **val**.
- Funcția **return** are un parametru de tip **a**, devine o acțiune de tip **IO a**, care nu face altceva decât să returneze valoarea respectivă.

do Exemplu


:: Integer
:: IO Integer
:: IO Char
do x <- f 5
c <- getChar
putChar c
return (x + 4)
:: Char
:: IO ()
(acţiune fără
"v <- …")
:: IO Integer
(tipul ultimei acţiuni determină
tipul expresiei do)

Acțiuni IO

- O valoare de tip **`IO a`** este o acțiune; această valoare va avea efect atunci când va fi executată.

- În Haskell, o valoare program este valoarea variabilei **`main`** din modulul **`Main`**. Dacă această valoare este de tip **`IO a`**, atunci va fi executată ca o acțiune. Dacă este de alt tip atunci valoarea sa va fi afişată.

Acțiuni IO predefinite

- Introducerea unui caracter de la tastatură

```
getChar :: IO Char
```

- Scrie un caracter la terminal

```
putChar :: Char -> IO ()
```

- Introducerea unei linii de la tastatură

```
getLine :: IO String
```

- Citirea unui fişier ca şi String

```
readFile :: FilePath -> IO String
```

- Scriere String în fişier

```
writeFile :: FilePath -> String -> IO ()
```

Acțiuni recursive

- Acțiunea **`getLine`** poate fi definită recursiv din acțiuni mai simple:

```
getLine :: IO String
getLine =  do c <- getChar          -- citeste un caracter
              if c == '\n'          -- daca este newline
                then return []      -- return sirul vid
                else do l <- getLine -- recursiv restul
              return (c:l)      -- return intreaga linie
```

Acțiuni recursive

- Acțiunea **`putStr`** poate fi definită recursiv din acțiuni mai simple:

```
putStr:: String -> IO ()

putStr[]      = return ()
putStr (x:xs) = do putChar x
                   putStr xs

putStrLn:: String -> IO ()

putStrLn xs   = do putStr xs
                   putChar '\n'
```

Exemplul 1: Lungimea unui șir citit

- Acțiunea de citire a unui șir de la tastatură și afișarea lungimii sale:

```
strlen:: IO()
strlen = do putStr "Introdu un sir: "
            xs <- getLine
            putStr " Sirul are "
            putStr(show(length(xs))
            putStrLn " caractere"
```

```
Main> strlen
Introdu un sir: abracadabra
Sirul are 11 caractere
```

Exemplul 2: Suma unor intregi

```
getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)


sumInts :: IO Int
sumInts = do n <- getInt
             if n==0
                then return 0
                else
                   do m <- sumInts
                      return (n+m)
suma::IO ()
suma = do x <- sumInts
          putStrLn("Suma este: " ++ show x )
```

```
Main> suma
0
Suma este: 0


Main> suma
354
24
100
0
Suma este: 478
```

Exemplul 3: Comanda Unix wc

- Programul Unix wc (word count) citeşte un fişier şi determină numărul de caractere, cuvinte şi linii pe care le afişează
- Citirea fişierului este o acțiune, restul este un calcul.
- Strategia:
  - Se defineşte o funcție care determină numărul de caractere, cuvinte şi linii într-un string.
    - Numărul de linii = numărul de '\n'
    - Numărul de cuvinte ~= numărul de ' ' plus numărul de '\t'
  - Se defineşte o acțiune care citeşte un fişier într-un string, aplică funcția, apoi afișează rezultatul.

Implementarea

```
wcf :: (Int,Int,Int) -> String -> (Int,Int,Int)
wcf (cc,w,lc) []           = (cc,w,lc)
wcf (cc,w,lc) (' '  : xs) = wcf (cc+1,w+1,lc) xs
wcf (cc,w,lc) ('\t' : xs) = wcf (cc+1,w+1,lc) xs
wcf (cc,w,lc) ('\n' : xs) = wcf (cc+1,w+1,lc+1) xs
wcf (cc,w,lc) (x : xs)    = wcf (cc+1,w,lc) xs

wc :: IO ()
wc = do putStr "Dati numele fisierului: "
        name     <- getLine
        contents <- readFile name
        let (cc,w,lc) = wcf (0,0,0) contents
        putStrLn ("Fisierul: " ++ name ++ " are ")
        putStrLn (show cc ++ " caractere ")
        putStrLn (show w  ++ " cuvinte ")
        putStrLn (show lc ++ " linii ")
```

Exemplu execuție

```
*Main> wc
Dati numele fisierului: wcount.hs
Fisierul: wcount.hs are
615 caractere
173 cuvinte
17 linii
```

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

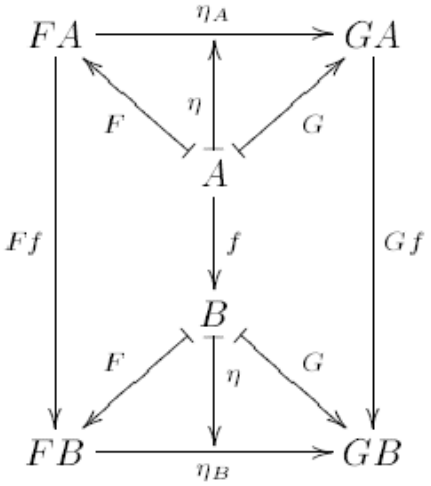
$$\begin{array}{rccc} \eta & : & \mathcal{O}_{\mathcal{A}} & \longrightarrow & \mathcal{M}_{\mathcal{B}} \\ & & A & \longmapsto & \eta_A \end{array}$$

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.

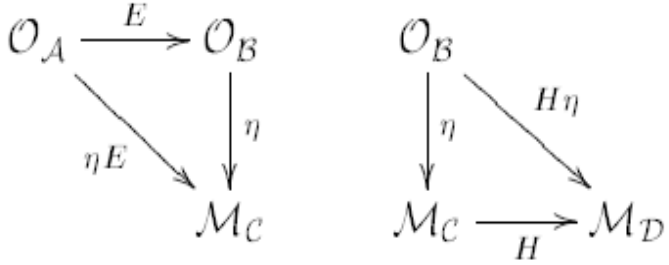
Composing transformations and

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,

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

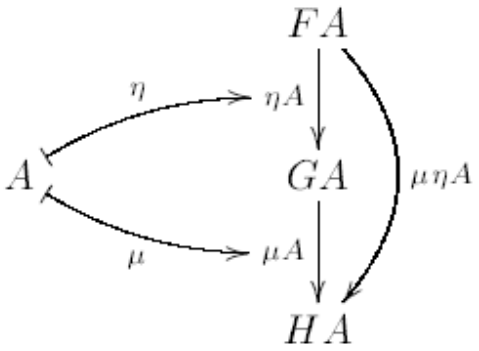
hold. In (other) words, $H\eta$ and ηE are both natural transformations.

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.,

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

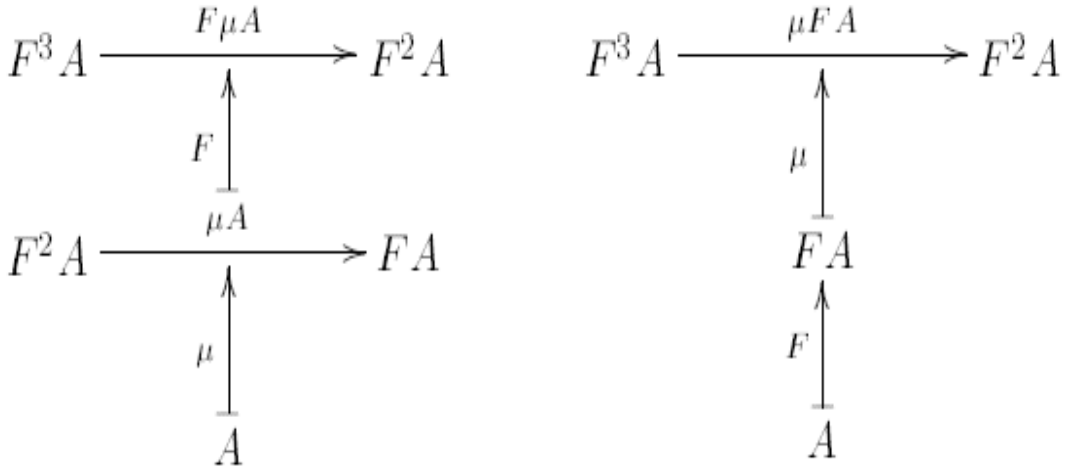
where $[\,]$ denotes the empty list. This, however, implies

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

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.