The monad and examples from Haskell

A computer-checked library for Category Theory in Lean

Csanád Farkas
Supervisors: Benedikt Ahrens, Lucas Escot
Technical University of Delft

Introduction

- The aim of this project was to make a library for category theory.
- We used a **computer checked language**
  - this can check the correctness of proofs
  - we chose to use Lean 3
- My question was to add the definition of the **monad**, and add **examples** from Haskell
  - The two examples are: *Maybe*, *List*
- This poster will give both the regular definitions as well as the corresponding code snippets.

Category

A category $\mathcal{C}$ is defined as the following [1]:

- A collection of objects, $A, B, \ldots$
- A collection of arrows, $f : A \to B$
- A composition operator between arrows:

$$f : A \to B, g : B \to C$$
$$g \circ f : A \to C$$

- The composition must follow three laws:

$$f \circ id_A = f$$

$$id_B \circ f = f$$

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

The first two of which are sometimes combined into one unit law. id_A is the identity arrow on object A.

```
structure category :=
  --aтributes
  (C₀      : Sort u)
  (hom     : Π (X Y : C₀), Sort v)
  (id      : Π (X : C₀), hom X X)
  (compose : Π {X Y Z : C₀}
    (g : hom Y Z)
    (f : hom X Y),
    hom X Z
  )

  --axioms
  (left_id  : ∀ {X Y : C₀} (f : hom X Y),
    compose f (id X) = f)

  (right_id : ∀ {X Y : C₀} (f : hom X Y),
    compose (id Y) f = f)

  (assoc    : ∀ {X Y Z W : C₀}
    (f : hom X Y) (g : hom Y Z) (h : hom Z W),
    compose h (compose g f) =
    compose (compose h g) f
  )
```

Functor

The full definition is as follows [1]:

- A functor $F : \mathcal{C} \to \mathcal{D}$ between categories $\mathcal{C}$ and $\mathcal{D}$.
- F maps each $\mathcal{C}$-object to a $\mathcal{D}$-object.
- F maps each $\mathcal{C}$-arrow to a $\mathcal{D}$-arrow.
- It must follow the laws below:

$$F(id_A) = id_{F(A)}$$
$$F(g \circ f) = F(g) \circ F(f)$$

- Two functors $F : \mathcal{C} \to \mathcal{D}, G : \mathcal{D} \to \mathcal{E}$ can be composed into $G \cdot F : \mathcal{C} \to \mathcal{E}$

```
structure functor (𝒞 𝒟 : category) :=
  (map_obj : 𝒞 → 𝒟)
  (map_hom : Π {X Y : 𝒞} (f : 𝒞.hom X Y),
    𝒟.hom (map_obj X) (map_obj Y))
  (id : ∀ (X : 𝒞), map_hom (𝒞.id X) = 𝒟.id
  (map_obj X))
  (comp : ∀ {X Y Z : 𝒞} (f : 𝒞.hom X Y) (g : 𝒞
  .hom Y Z),
    map_hom (𝒞.compose g f) = 𝒟.compose (map_hom
  g) (map_hom f))

def composition_functor {𝒞 𝒟 ℰ : category}
  (G : 𝒟 ⇒ ℰ) (F : 𝒞 ⇒ 𝒟) : 𝒞 ⇒ ℰ :=
{
  map_obj := λ X, G.map_obj (F.map_obj X),
  map_hom := λ _ _ f, G.map_hom (F.map_hom f),
  id := begin intro, rw F.id, rw G.id, end,
  comp := begin intros, rw F.comp, rw G.comp, end,
}
```

Natural Transformatiion

Assume the functors F, G are from $\mathcal{C}$ to $\mathcal{D}$. The transformation must for each $\mathcal{C}$-object A assign a $\mathcal{D}$-arrow from $F(A)$ to $G(A)$, denoted α_A. To be natural, it must also:

$$\forall f : A \to B, (A, B \in \mathcal{C}) \Rightarrow$$
$$\alpha_B \circ F(f) = G(f) \circ \alpha_A$$

Natural transformations can also be composed:

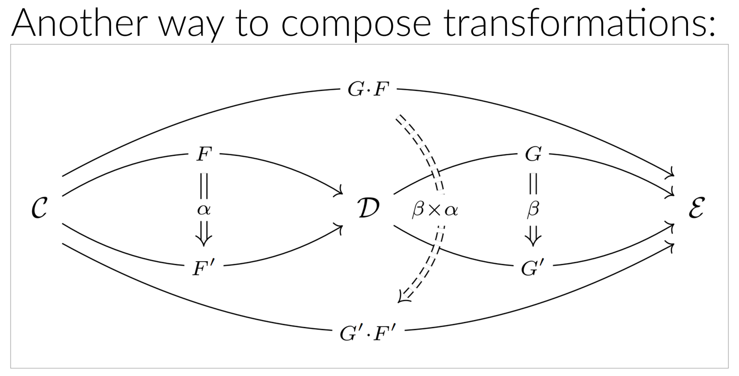
$$\alpha : F \dot\to G, \beta : G \dot\to H$$
$$\beta \circledcirc \alpha : F \dot\to H$$

```
structure natural_transformation {𝒞 𝒟 : category}
  (F G : 𝒞 ⇒ 𝒟) :=
  (α : Π (X : 𝒞) ,
    𝒟.hom (F.map_obj X) (G.map_obj X))
  (naturality_condition :
    ∀ {X Y : 𝒞} (f : 𝒞.hom X Y),
    𝒟.compose (G.map_hom f) (α X)  =
    𝒟.compose (α Y) (F.map_hom f)
  )

def nt_comp {𝒞 𝒟 : category} {F G H : 𝒞 ⇒ 𝒟}
    (τ₁ : G ≫ H) (τ₂ : F ≫ G) : F ≫ H :=
    {
      α :=  X, 𝒟.compose (τ₁ . X) (τ₂ . X),
      naturality_condition := _,
    }
```

Horizontal Composition

Another way to compose transformations:




```
  def bimap {𝒞 𝒟 ℰ: category}
      {F F' : 𝒞 ⇒ 𝒟} {G G' : 𝒟 ⇒ ℰ}
      (bb : G ≫ G') (α : F ≫ F')
    : (G · F) ≫ (G' · F') :=
    {
      α :=  X,
        ℰ.compose (bb.α (F'.map_obj X))
  (G.map_hom (α.α X)),
      naturality_condition := _,
    }
```

Proof for naturality:

$$\begin{tikzcd}
X \arrow[d, "f"] & G \cdot F(X) \arrow[r, "G(\alpha_X)"] \arrow[d, "G \cdot F(f)"] & G \cdot F'(X) \arrow[r, "\beta_{F'(X)}"] \arrow[d, "G \cdot F'(f)"] & G' \cdot F'(X) \arrow[d, "G' \cdot F'(f)"] \\
Y & G \cdot F(Y) \arrow[r, "G(\alpha_Y)"] & G \cdot F'(Y) \arrow[r, "\beta_{F'(Y)}"] & G' \cdot F'(Y)
\end{tikzcd}$$

Natural Isomorphisms

While it may be easy to see from the definitions that:

$$F \cdot (G \cdot H) = (F \cdot G) \cdot H$$
$$Id \cdot F = F = F \cdot Id$$

The type checker sees those as different types. To fix this we can show that there is a natural transformation from one side to the other, and the other way round. This transformation is then called a **natural isomorphism**. To prove they are a natural isomorphism we need to show that all assigned arrows in the transformation are isomorphisms. This is easy as the assigned arrows are the identity arrow.

Monad

The monad is defined by two natural transformations and some laws that must apply to said transformations:

- Given a category $\mathcal{C}$, and an endofunctor $T : \mathcal{C} \to \mathcal{C}$.
- A natural transformation μ, from $T^2 = T \cdot T$ to T.
- A natural transformation η, from Id to T.
- With laws:

$$\mu \circledcirc \mu \times ID_T = \mu \circledcirc ID_T \times \mu$$
$$\mu \circledcirc ID_T \times \eta = ID_T = \mu \circledcirc \eta \times ID_T$$

Here ID_T is the identity natural transformation from T to T.

```
structure Monad
    {𝒞 : category}
    (T : 𝒞 ⇒ 𝒞)
    :=

    (μ : (T · T) ≫ T)

    (η : (Id 𝒞) ≫ T)

    (assoc :
      μ ⊚ μ × (ID T) = t
      μ ⊚ (ID T) × μ ⊚ (assoc_nt T T T)
    )

    (lu : μ ⊚ ID T × η = ID T ⊚ right_unit_nt T)

    (ru : μ ⊚ η × ID T = ID T ⊚ left_unit_nt T)
```

The laws visually:

$$\begin{tikzcd}
T \arrow[r, "T \times \eta"] \arrow[dr, "T"'] & T^2 \arrow[d, "\mu"] & T^2 \arrow[d, "\mu"] & T \arrow[l, "\eta \times T"'] \arrow[dl, "T"] \\
& T & T &
\end{tikzcd}$$

$$\begin{tikzcd}
T^3 \arrow[r, "T \times \mu"] \arrow[d, "\mu \times T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"] & T
\end{tikzcd}$$

`Maybe`

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



-- μ
join            :: Maybe (Maybe a) → Maybe a
join (Just x) = x
join Nothing  = Nothing


-- η
return :: a → Maybe a
return = Just
```

```
-- Definition
inductive Maybe (α : Type*)
  | none : Maybe
  | some : α → Maybe

-- μ
def Maybe.join {α : Type*}
  : Maybe (Maybe α) → Maybe α
  | Maybe.none := Maybe.none
  | (Maybe.some x)  := x

-- η
def Maybe.return {α : Type*} (x : α) : Maybe α :=
  Maybe.some x
```

$$\begin{tikzcd}
Ma \arrow[r, "\texttt{fmap return}"] \arrow[dr, "\texttt{id}"'] & M(Ma) \arrow[d, "\texttt{join}"] & M(Ma) \arrow[d, "\texttt{join}"] & Ma \arrow[l, "\texttt{return}"'] \arrow[dl, "\texttt{id}"] \\
& Ma & Ma &
\end{tikzcd}$$

$$\begin{tikzcd}
M(M(Ma)) \arrow[r, "\texttt{fmap join}"] \arrow[d, "\texttt{join}"'] & M(Ma) \arrow[d, "\texttt{join}"] \\
M(Ma) \arrow[r, "\texttt{join}"] & Ma
\end{tikzcd}$$

`List`

```
-- Definition
data List a = [] | a : List a








-- μ
join    :: [[a]] → [a]
join xs = [y | x ← xs, y ← x]


-- η
return   :: a → [a]
return x = [x]
```

```
inductive List (α: Type) : Type
  | nil : List
  | cons (head: α) (tail: List) : List

def List.merge {α : Type}
  : List α → List α → List α
  | List.nil ys := ys
  | (List.cons x xs) ys :=
    List.cons x (List.merge xs ys)

-- μ
def List.join {α} : List (List α) → List α
  | List.nil := List.nil
  | (List.cons l ls) := List.merge l (List.join ls)

-- η
def List.return {α} (a : α) : List α :=
  List.cons a List.nil
```

$$\begin{tikzcd}
\texttt{[a]} \arrow[r, "\texttt{fmap return}"] \arrow[dr, "\texttt{id}"'] & \texttt{[[a]]} \arrow[d, "\texttt{join}"] & \texttt{[[a]]} \arrow[d, "\texttt{join}"] & \texttt{[a]} \arrow[l, "\texttt{return}"'] \arrow[dl, "\texttt{id}"] \\
& \texttt{[a]} & \texttt{[a]} &
\end{tikzcd}$$

$$\begin{tikzcd}
\texttt{[[[a]]]} \arrow[r, "\texttt{fmap join}"] \arrow[d, "\texttt{join}"'] & \texttt{[[a]]} \arrow[d, "\texttt{join}"] \\
\texttt{[[a]]} \arrow[r, "\texttt{join}"] & \texttt{[a]}
\end{tikzcd}$$

References

[1] B. C. Pierce, "A taste of category theory for computer scientists,", Feb. 2011. DOI: 10.1184/R1/6602756.v1. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/A_taste_of_category_theory_for_computer_scientists/6602756.