计算概论A—实验班

函数式程序设计

Functional Programming

胡振江，张 伟

北京大学 计算机学院

2024年09～12月

范畴论、Functor、Monad

Category Theory, Functor, and Monad

范畴论、Functor、Monad

- Haskell中有很多成分直接抄袭/借鉴自范畴论（Category Theory, CT）
  - 例如，Functor、Monad就是两个来源于范畴论的概念
- 很不幸的是，CT中的绝大部分概念都极为抽象，如同阳春白雪，超出了下里巴人的审美能力
  - 维特根斯坦认为：“凡能够说的，都能够说清楚；凡不能谈论的，就应该保持沉默（对人类而言）”
  - 我不确认：CT是否是人类能或不能谈论的
  - 但是，可以确认的是：一些数学家在几十年前就开始谈论CT了
  - 即然如此，我们就大胆地去谈论吧（如果天真的塌了，先砸到的也是这些高傲的数学家）

不谈行不行？

其实也可以不谈；但是，这意味着你永远无法看透Haskell。

不看透又如何？

不看透，就无法超越。

范畴论是一种什么理论?

- 范畴论是一种元数学语言，用于描述不同细分领域之间存在的抽象共性成分

> Category theory provides a cross-disciplinary language for mathematics designed to delineate general phenomena, which enables the transfer of ideas from one area of study to another.

- 范畴论是一种关于类比（analogy）的理论。
  - 法国数学家庞加莱曾经说过：“数学是一种为不同事物进行相同命名的艺术，而诗歌则是一种为同一事物进行不同命名的艺术。”

    > Mathematics is the art of giving the same name to different things. Poetry is the art of giving different names to the same thing.

  - 数学与诗歌这两个表面上看起来风牛马不相及的领域，在 **类比** 的意义上得到了统一。
  - 范畴论大概做了类似的事情：祂为不同数学细分领域之间提供了一种统一的类比机制，使得人类能够在形式上观察到这些细分领域背后蕴含的某种共性结构。
- 范畴论是如何做到这些事情的？ 有这样一句歌词：“远离地面，快接近三万英尺的距离”。

内容来源

本幻灯片的内容主要来源于如下材料：

1. Saunders Mac Lane. Categories for the Working Mathematician (Second Edition). Springer, 1998.
2. Steve Awodey. Category Theory (Second Edition). Oxford University Press, 2010.
3. Emily Riehl. Category Theory in Context. Dover Publications, 2016.
4. Bartosz Milewski. Category Theory for Programmers (Version v1.3.0-0-g6bb0bc0). August 12, 2019.

Category / 范畴

Definition： Category / 范畴

A **category** C consists of the following data :

1. A class of **objects** / 对象: $a, b, c, \ldots$, denoted by C_0
2. A class of **morphisms** / 态射: $f, g, h, \ldots$, denoted by C_1
3. Two operations on morphisms: **domain** and **codomain**.
   - The *former* assigns to each morphism f in C_1 an object in C_0, denoted by $\mathsf{dom} f$.
   - The *latter* assigns to each morphism f in C_1 an object in C_0, denoted by $\mathsf{cod} f$.

     > We use notation $f : a \to b$ or $f_{a\to b}$ to indicate a morphism f in C_1 with $\mathsf{dom} f = a$ and $\mathsf{cod} f = b$
4. An **identity** operation that assigns to each object a in C_0 a morphism $f_{a\to a}$ in C_1, denoted by 1_a
5. A **composition** operation that assigns to each pair $(g_{b\to c}, f_{a\to b})$ of morphisms in C_1 a morphism $h_{a\to c}$ in C_1, denoted by $g \circ f$.

These data are subject to the following two **axioms**:

1. **Unit** : $f \circ 1_a = f = 1_b \circ f$, for any $f_{a\to b}$ in C_1.
2. **Associativity** : $h \circ (g \circ f) = (h \circ g) \circ f$, for any $f_{a\to b}$, $g_{b\to c}$, $h_{c\to d}$ in C_1.

Definition：Category / 范畴 ---------- (中文版本)

一个 **范畴** C 由如下数据构成：

1. 一类对象：$a, b, c, \ldots$，记为 C_0
2. 一类态射：$f, g, h, \ldots$，记为 C_1
3. 态射上的两个操作：**domain** 和 **codomain**.
   - *前者*为 C_1 中的每一个态射 f 赋予 C_0 中的一个对象，记为 $\mathsf{dom} f$.
   - *后者*为 C_1 中的每一个态射 f 赋予 C_0 中的一个对象，记为 $\mathsf{cod} f$.

     > 我们使用符号 $f : a \to b$ 或 $f_{a\to b}$ 表示 C_1 中的一个态射 f，其满足 $\mathsf{dom} f = a$ 且 $\mathsf{cod} f = b$
4. 一个 **identity** 操作：其为 C_0 中的每一个对象 a 赋予 C_1 中的一个态射 $f_{a\to a}$，记为1_a
5. 一个 **composition** 操作：其为 C_1 中的任何一对态射 $(g_{b\to c}, f_{a\to b})$ 赋予 C_1 中的一个态射 $h_{a\to c}$，记为 $g \circ f$.

这些数据需要满足如下两个公理：

1. **Unit** : $f \circ 1_a = f = 1_b \circ f$，对于 C_1 中的任何一个态射 $f_{a\to b}$.
2. **Associativity** : $h \circ (g \circ f) = (h \circ g) \circ f$，对于 C_1 中的任何三个态射 $f_{a\to b}$，$g_{b\to c}$，$h_{c\to d}$.

任何一个满足上述定义的事物都是一个范畴

除此之外，无需对范畴产生其他任何幻想

下面，我们就拿着这个定义/模具，去构造几个简单到无聊的范畴吧！

范畴示例之：Zero

这里有一个范畴，你看见了吗?

你…在…说…什…么…，太…空…旷..了…，有…回…声…，听…不…清…

年纪轻轻就耳背了。注意锻炼身体啊！

Zero：一个不包含任何对象和态射的范畴

- 既然空集 ∅ 是一个集合，那么，空范畴 Zero 自然也是一个范畴

范畴示例之：One、Two、Thr

- 范畴 One:
  - $\mathsf{One}_0 \doteq \{ * \}, \quad \mathsf{One}_1 \doteq \{ 1_* \}$
- 范畴 Two
  - $\mathsf{Two}_0 \doteq \{ *, \star \}, \quad \mathsf{Two}_1 \doteq \{1_*, 1_\star, f_{*\to\star}\}$
- 范畴 Thr
  - $\mathsf{Thr}_0 \doteq \{ *, \star, \bullet \}, \quad \mathsf{Thr}_1 \doteq \{ 1_*, 1_\star, 1_\bullet, f_{*\to\star}, g_{\star\to\bullet}, h_{*\to\bullet} \}, \quad h = g \circ f$

💡 **Finite Category / 有限范畴:** 称范畴 C 为一个有限范畴，当且仅当 C_1 是一个有限集合。

💡 对于任一有限范畴 C，C_0 一定是一个有限集合。

给你一个集合，如何用最小成本将其发展为一个范畴？

- 给定集合 A，可定义一个范畴 A：
  - $\mathsf{A}_0 \doteq \{\, a \mid a \in A \,\} = A$
  - $\mathsf{A}_1 \doteq \{1_a \mid a \in \mathsf{A}_0\}$
  - 定义态射的组合，确保对于任何 $a \in \mathsf{A}_0$，$1_a \circ 1_a = 1_a$

你要是再不介绍几个有趣的范畴，我们就要准备退课了！

年轻人，有话好好说嘛！千万别生气，身体是革命的本钱，千万要锻炼好身体哈。

我认为：下面即将出现的这个范畴，也许会永远驻留在你的心中。

真 TM 的啰嗦，快要 emo 了…

范畴示例之: Fn

The category Fn consists of the following data :

1. $\mathsf{Fn}_0 \doteq \{a \mid a \text{ is a set}\}$
2. $\mathsf{Fn}_1 \doteq \{f \mid f \text{ is a function between two sets in } \mathsf{Fn}_0\}$
3. Two operations on functions: **domain** and **codomain**
   - For each function f in Fn_1, $\mathsf{dom} f \doteq f$'s domain set, and $\mathsf{cod} f \doteq f$'s codomain set.
4. The **identity** operation
   - For each set a in Fn_0, $1_a \doteq \{x \mapsto x \mid x \in a\}$.
5. The **composition** operation
   - For each pair $(g_{b\to c}, f_{a\to b})$ of functions in Fn_1, $(g \circ f)_{a\to c} \doteq \{x \mapsto g(f(x)) \mid x \in a\}$.

These data are subject to the following two axioms:

1. **Unit** : $f \circ 1_a = f = 1_b \circ f$, for any $f_{a\to b}$ in Fn_1.
2. **Associativity** : $h \circ (g \circ f) = (h \circ g) \circ f$, for any $f_{a\to b}$, $g_{b\to c}$, $h_{c\to d}$ in Fn_1.

范畴示例之: Fn // Concise Version

一个关于集合（as objects）与函数（as morphisms）的范畴

The category Fn is defined by:

1. $\mathsf{Fn}_0 \doteq \{a \mid a \text{ is a set}\}$
2. $\mathsf{Fn}_1 \doteq \{f \mid f \text{ is a function between two sets in } \mathsf{Fn}_0\}$
3. $\mathsf{dom} f \doteq f$'s domain set, and $\mathsf{cod} f \doteq f$'s codomain set.
4. $1_a \doteq \{x \mapsto x \mid x \in a\}$.
5. $(g_{b\to c} \circ f_{a\to b})_{a\to c} \doteq \{x \mapsto g(f(x)) \mid x \in a\}$.

These data are subject to the following two axioms:

1. **Unit** : $f \circ 1 = f = 1 \circ f$

   > Proof: $(f_{a\to b} \circ 1_a)\ x = f_{a\to b}\ (1_a\ x) = f_{a\to b}\ x = 1_b\ (f_{a\to b}\ x) = (1_b \circ f_{a\to b})\ x$

2. **Associativity** : $h \circ (g \circ f) = (h \circ g) \circ f$

   > Proof: $(h \circ (g \circ f))\ x = h\ ((g \circ f)\ x) = h\ (g\ (f\ x)) = (h \circ g)\ (f\ x) = ((h \circ g) \circ f)\ x$

Fn 这个范畴，有什么了不起呢？

如果把 Haskell 类比为 孙悟空，那么，Fn 大概就是 如来佛的手掌了。

Haskell 再怎么折腾，大概也只是在 Fn 这个手掌内转圈。

听起来好神秘的样子...

关于范畴定义中一个重要细节

- 在前面范畴的定义中，我们说：一个范畴中存在 a **class** of objects 和 a **class** of morphisms。
- 能否把这种陈述修改为 a **set** of objects 或 a **set** of morphisms 吗？// Set 对应的中文是“集合”

- 或者更直接一些：$\mathsf{Fn}_0 \doteq \{a \mid a \text{ is a set}\}$ 这个东西是一个集合吗？
- 上述问题的等价问题是：存在一个包含所有集合的集合吗？
- 对于这个问题，确定的答案是：不存在！
  - 因为：对于任意集合a，我们都可以构造出一个集合b，满足 $b \notin a$

Proposition: 对于任意集合 a 以及集合 $b \doteq \{x \in a \mid x \notin x\}$，可知：$b \notin a$.

▶ Proof: 反证法

$$
\begin{aligned}
(b \in a) &\Rightarrow (b \in b) \vee (b \notin b) \\
&\Rightarrow ((b \in b) \wedge (b \notin b)) \ \vee \ ((b \notin b) \wedge (b \in b)) \\
&\Leftrightarrow \text{false} \vee \text{false} \\
&\Leftrightarrow \text{false}
\end{aligned}
$$

范畴示例之: Rel

一个关于集合（as objects）与集合之间的二元关系（as morphisms）的范畴

The category Rel is defined by:

1. $\mathsf{Rel}_0 \doteq \{a \mid a \text{ is a set}\}$
2. $\mathsf{Rel}_1 \doteq \{r \mid r \text{ is a binary relation between two sets in } \mathsf{Rel}_0\}$
3. $\mathsf{dom}\ r \doteq r$'s left set, and $\mathsf{cod}\ r \doteq r$'s right set.
4. $1_a \doteq \{(x, x) \mid x \in a\}$.
5. $(s_{b\to c} \circ r_{a\to b})_{a\to c} \doteq \{(x, z) \mid \exists y \in b : (x, y) \in r \wedge (y, z) \in s\}$.

These data are subject to the following two axioms:

1. **Unit** : $r \circ 1 = r = 1 \circ r$
2. **Associativity** : $t \circ (s \circ r) = (t \circ s) \circ r$

> See next two slides for the two axioms' proof.

范畴示例之: Rel

1. **Unit** : $r \circ 1 = r = 1 \circ r$

Proof:

$$
\begin{aligned}
& (x, z) \in r_{a \to b} \circ 1_a \\
\equiv \quad & \exists y \in a : (x, y) \in 1_a, (y, z) \in r \\
\equiv \quad & \exists y \in a : x \in a, x = y, (y, z) \in r \\
\equiv \quad & \exists y \in a : x \in a, x = y, (x, z) \in r \\
\equiv \quad & (x, z) \in r \wedge (\exists y \in a : x \in a, x = y) \\
\equiv \quad & (x, z) \in r \wedge \text{true} \\
\equiv \quad & (x, z) \in r
\end{aligned}
$$

According to the **axiom of extensionality** of the ZFC set theory, we have $r_{a \to b} \circ 1_a = r$.

$$
\begin{aligned}
& (x, z) \in 1_b \circ r_{a \to b} \\
\equiv \quad & \exists y \in b : (x, y) \in r, (y, z) \in 1_b \\
\equiv \quad & \exists y \in b : (x, y) \in r, z \in b, y = z \\
\equiv \quad & \exists y \in b : (x, z) \in r, z \in b, y = z \\
\equiv \quad & (x, z) \in r \wedge (\exists y \in b : z \in b, y = z) \\
\equiv \quad & (x, z) \in r \wedge \text{true} \\
\equiv \quad & (x, z) \in r
\end{aligned}
$$

According to the **axiom of extensionality** of the ZFC set theory, we have $r = 1_b \circ r_{a \to b}$.

范畴示例之: Rel

2. **Associativity** : $t \circ (s \circ r) = (t \circ s) \circ r$

Proof:

$$
\begin{aligned}
& (x, w) \in t_{c \to d} \circ (s_{b \to c} \circ r_{a \to b}) \\
\equiv \quad & \exists z \in c : (x, z) \in (s_{b \to c} \circ r_{a \to b}), (z, w) \in t \\
\equiv \quad & \exists z \in c : (\exists y \in b : (x, y) \in r, (y, z) \in s), (z, w) \in t \\
\equiv \quad & \exists z \in c : \exists y \in b : (x, y) \in r, (y, z) \in s, (z, w) \in t \\
\equiv \quad & \exists y \in b : \exists z \in c : (x, y) \in r, (y, z) \in s, (z, w) \in t
\end{aligned}
$$

$$
\begin{aligned}
& (x, w) \in (t_{c \to d} \circ s_{b \to c}) \circ r_{a \to b} \\
\equiv \quad & \exists y \in b : (x, y) \in r, (y, w) \in (t_{c \to d} \circ s_{b \to c}) \\
\equiv \quad & \exists y \in b : (x, y) \in r, (\exists z \in c : (y, z) \in s, (z, w) \in t) \\
\equiv \quad & \exists y \in b : \exists z \in c : (x, y) \in r, (y, z) \in s, (z, w) \in t \\
\equiv \quad & (x, w) \in t_{c \to d} \circ (s_{b \to c} \circ r_{a \to b})
\end{aligned}
$$

According to the **axiom of extensionality** of the ZFC set theory, we have $t \circ (s \circ r) = (t \circ s) \circ r$.

范畴示例之: Pos

- **Definition**: A **partially ordered set** (**poset**) is a tuple $(a, \leq_a)$, where:
  - a is a set
  - $\leq_a \subseteq\ a \times a$ is a binary relation that satisfies the following three conditions:
    1. **Reflexivity**: $x \leq_a x$, for all $x \in a$.
    2. **Transitivity**: $(x \leq_a y) \wedge (y \leq_a z) \Rightarrow (x \leq_a z)$, for all $x, y, z \in a$.
    3. **Antisymmetry**: $(x \leq_a y) \wedge (y \leq_a x) \Rightarrow (x = y)$, for all $x, y \in a$.
- **Definition**: A **monotone** function between two posets $(a, \leq_a)$ and $(b, \leq_b)$ is a function $m : a \to b$ such that the following condition hold:

$$x \leq_a y \quad \Rightarrow \quad m\, x \leq_b m\, y, \qquad \text{for all } x, y \in a.$$

范畴示例之: Pos

The category Pos is defined by:

1. $\mathsf{Pos}_0 \doteq \{a \mid a \text{ is a poset}\}$
2. $\mathsf{Pos}_1 \doteq \{m \mid m \text{ is a monotone function between posets in } \mathsf{Pos}_0\}$
3. $\mathsf{dom}\ m \doteq m\text{'s source poset}$, and $\mathsf{cod}\ m \doteq m\text{'s target poset}$
4. $1_a \doteq \{x \mapsto x \mid x \in a_0\}$

   ▶ Proof: identity is a monotone function.
5. $(n_{b\to c} \circ m_{a\to b})_{a\to c} \doteq \{x \mapsto n(m(x)) \mid x \in a_0\}$

   ▶ Proof: the compostion of composable monotone functions is a monotone function.

These data are subjects to the following two axioms:

1. **Unit** : $m \circ 1 = m = 1 \circ m$
2. **Associativity** : $r \circ (n \circ m) = (r \circ n) \circ m$

Small Categories, Locally Small Categories

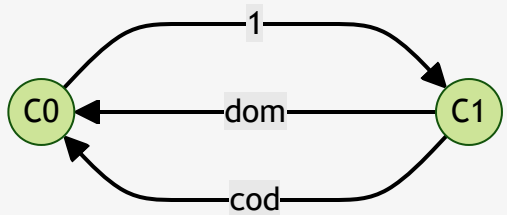
Definition :

A category C is **small**, iff C_1 is a set.

If C is a small category, then C_0 must be a set.

- For each $a \in \mathsf{C}_0$, there is a distinct morphism $1_a \in \mathsf{C}_1$.

- A small category C has three functions:



Definition : Given a category C, for any two objects a, b in C_0,

$$\mathsf{C}(a,b) \doteq \{f \in \mathsf{C}_1 \mid \mathsf{dom} f = a,\ \mathsf{cod} f = b\}$$

That is, $\mathsf{C}(a,b)$ is a **class** that consists of morphisms from a to b in C_1.

Definition: A category C is **locally small**, iff for any two objects a, b in C_0, $\mathsf{C}(a,b)$ is a set.

Isomorphism / 同构态射

Definition: A morphism $f_{x \to y}$ in a category is an **isomorphism**, iff :

$$\text{There exists a morphism } g_{y \to x}, \text{ such that } g \circ f = 1_x \text{ and } f \circ g = 1_y.$$

- Here: g, denoted by f^{-1}, is called the *inverse* isomorphism of f; and vise versa.
  - ▶ Proof: Any isomorphism has only one inverse isomorphism.

Definition: Two objects x, y in a category are **isomorphic**, denoted by $x \cong y$, iff:

$$\text{There exists an isomorphism between } x \text{ and } y.$$

💡 An **endomorphism** / 自同态态射: A morphism f with $\mathsf{dom} f = \mathsf{cod} f$.

💡 An **automorphism** / 自同构态射: A morphism that is both *endomorphism* and *isomorphism*.

Isomorphism / 同构态射

Definition: A morphism $f_{x \to y}$ in a category is an **isomorphism**, iff :

$$\text{There exists a morphism } g_{y \to x}, \text{ such that } g \circ f = 1_x \text{ and } f \circ g = 1_y.$$

- Here: g, denoted by f^{-1}, is called the *inverse* isomorphism of f; and vise versa.
  - Proof: Any isomorphism has only one inverse isomorphism.

- Given an isomorphism $f_{x \to y}$,

suppose its has two inverse isomorphisms $g_{y \to x}, h_{y \to x}$.

$$\begin{aligned} & (g \circ f) \circ h = g \circ (f \circ h) \\ \Leftrightarrow \quad & 1_x \circ h = g \circ 1_y \\ \Leftrightarrow \quad & h = g \end{aligned}$$

Definition: Two objects x, y in a category are **isomorphic**, denoted by $x \cong y$, iff:

$$\text{There exists an isomorphism between } x \text{ and } y.$$

💡 An **endomorphism** / 自同态态射: A morphism f with $\mathsf{dom} f = \mathsf{cod} f$.

💡 An **automorphism** / 自同构态射: A morphism that is both *endomorphism* and *isomorphism*.

Isomorphism / 同构态射：补充说明

Definition: A morphism $f_{x \to y}$ in a category is an **isomorphism**, iff :

$$\text{There exists a morphism } g_{y \to x}, \text{ such that } g \circ f = 1_x \text{ and } f \circ g = 1_y.$$

1. **同构态射总是成对出现**
   - 若 f 为一个同构态射，则必然存在唯一一个 f 的逆同构态射，记为 f^{-1}
2. **关于“两个对象是否同构”的判断，需要建立在特定的范畴下**
   - 脱离特定的范畴，在比喻的意义上谈论同构，是不严谨的
   - 具体而言，需要指出：a. 当前关注的对象类（the class of objects）；b. 当前关注的对象之间的态射
   - 在此之前，需要确认这种态射是否满足两条公理（Unit，Associativity）
3. **对任何范畴 C 中的任何对象 a，1_a 是一个自同构态射**；也即，任何范畴中的任何对象都与自身同构
   - 原因：$1_a \circ 1_a = 1_a$ 且 $1_a \circ 1_a = 1_a$
   - 同时可知：$1_a^{-1} = 1_a$

Groupoid / 广群, Group / 群, Monoid / 单位半群

Definition: A category C is a **groupoid**, iff $\forall f \in \mathsf{C}_1 : f$ is an isomorphism.

Definition: A category C is a **group**, iff C is a groupoid with exactly one object in C_0.

群的另一种等价定义方式：一个满足两个条件的范畴：1. 仅具有一个对象；2. 所有态射都是同构态射。

Definition: A category C is a **monoid**, iff there exists exactly one object in C_0.

您是不是以为我们没有学过数学?

好孩子，何出此言呢？你是不是遇到了什么不开心的事情了?

我们学过 群 和 单位半群/幺半群。它们不是这样定义的！

你确信你是站在三万英尺的高空吗?

Monoid

Definition: A monoid is a triple $(m, u, \otimes)$:

1. a set m,
2. a distinguished *unit* element $u \in m$,
3. a binary operation $\otimes : m \times m \to m$

such that the following two conditions hold:

1. $u \otimes x = x = x \otimes u, \ \forall x \in m$
2. $x \otimes (y \otimes z) = (x \otimes y) \otimes z, \ \forall x, y, z \in m$

Some examples of monoids:

- $(\mathbb{N},\ 0,\ +)$, $(\mathbb{Z},\ 0,\ +)$, $(\mathbb{Q},\ 0,\ +)$, $(\mathbb{R},\ 0,\ +)$
- $(\mathbb{N},\ 1,\ \times)$, $(\mathbb{Z},\ 1,\ \times)$, $(\mathbb{Q},\ 1,\ \times)$, $(\mathbb{R},\ 1,\ \times)$

Any monoid is a category

A monoid $(m, u, \otimes)$ is a category M, where:

1. $\mathsf{M}_0 \doteq \{\ m\ \}$
2. $\mathsf{M}_1 \doteq \{f \mid f \in m\} = m$
3. $\mathsf{dom} f \doteq m$, and $\mathsf{cod} f \doteq m$
4. $1_m \doteq u_{m \to m}$
5. $(g_{m \to m} \circ f_{m \to m})_{m \to m} \doteq (g \otimes f)_{m \to m}$

These data are subject to the following two axioms:

1. **Unit** : $f \circ 1 = f = 1 \circ f$
2. **Associativity** : $h \circ (g \circ f) = (h \circ g) \circ f$

Group

Definition: A group is a triple $(g, u, \otimes)$:

1. a set g,
2. an unit element $u \in g$,
3. a binary operation $\otimes : g \times g \to g$,

such that the following four conditions hold:

1. $f \otimes u = f = u \otimes f,\ \forall f \in g$
2. $(f \otimes h) \otimes k = f \otimes (h \otimes k),\ \forall f, h, k \in g$
3. $\forall f \in g : \exists!\ f^{-1} \in g : f \otimes f^{-1} = f = f^{-1} \otimes f$

Any group is a category

A group $(g, u, \otimes)$ is a category G, where:

1. $\mathsf{G}_0 \doteq \{g\}$
2. $\mathsf{G}_1 \doteq \{f \mid f \in g\} = g$
3. $\mathsf{dom} f \doteq g,\ \text{and}\ \mathsf{cod} f \doteq g$
4. $1_g \doteq u_{g \to g}$
5. $h_{g \to g} \circ f_{g \to g} \doteq (h \otimes f)_{g \to g}$

These data are subject to the following two axioms:

1. **Unit** : $f \circ 1 = f = 1 \circ f$
2. **Associativity** : $k \circ (h \circ f) = (k \circ h) \circ f.$

In addition, the following property holds:

- Every $f \in \mathsf{G}_1$ is an isomorphism.

Functor / 函子

Functor / 函子

A **functor** F beteen two categories $\mathsf{C}, \mathsf{D},$ denoted by $F : \mathsf{C} \to \mathsf{D}$ or $F_{\mathsf{C}\to\mathsf{D}},$ consists of the following data:

1. An **object-mapping** operation.
   - It maps each object c in C_0 to an object in D_0, denoted by Fc.
2. An **morphism-mapping** operation.
   - It maps each morphism $f_{c\to c'}$ in C_1 to a morphism $g_{Fc\to Fc'}$ in D_1, denoted by Ff.

These data are required to satisfy the following two **functoriality axioms**:

1. $F\ 1_c = 1_{Fc},$ for each object c in C_0.
2. $Fg \circ Ff = F\ (g \circ f),$ for each two morphisms $f_{c\to c'}, g_{c'\to c''}$ in C_1.

Endofunctor / 自函子

A functor $F_{\mathsf{C}\to\mathsf{D}}$ that satisfies $\mathsf{C} = \mathsf{D}$ is called an **endofunctor**.

函子示例之：1_{C}

- 给定任一范畴 C，函子 $1_{\mathsf{C}} : \mathsf{C} \to \mathsf{C}$ 定义如下：
  - 对 C_0 中的任何一个对象 c，$1_{\mathsf{C}}\ c = c$
  - 对 C_1 中的任何一个态射 f，$1_{\mathsf{C}}\ f = f$
- 可知：1_{C} 满足函子的两个公理

1. $1_{\mathsf{C}}\ 1_c = 1_{1_{\mathsf{C}}\ c}$，for each object c in C_0.
2. $1_{\mathsf{C}}\ g \circ 1_{\mathsf{C}}\ f = 1_{\mathsf{C}}\ (g \circ f)$，for each two morphisms $f_{c \to c'}, g_{c' \to c''}$ in C_1.

函子示例之：$P_{\mathsf{Fn}\to\mathsf{Fn}}$

The functor is defined by:

- $P\ a \doteq \{x \mid x \subseteq a\}$.
- $P\ f_{a\to b} \doteq \{x \mapsto \{f\ i \mid i \in x\} \mid x \in P\ a\}$.

These data are required to satisfy the following two **functoriality axioms**:

1. $P\ 1_a = 1_{Pa}$

   Proof: $P\ 1_a = \{x \mapsto \{1_a\ i \mid i \in x\} \mid x \in P\ a\} = \{x \mapsto x \mid x \in P\ a\} = 1_{Pa}$

2. $Pg \circ Pf = P\ (g \circ f)$

   Proof:

   $$\begin{array}{lcl} P\ (g_{b\to c} \circ f_{a\to b}) & = & \{x \mapsto \{(g \circ f)\ i \mid i \in x\} \mid x \in P\ a\} \\ P\ (g_{b\to c} \circ f_{a\to b})\ x & = & \{(g \circ f)\ i \mid i \in x\} \\ (Pg \circ Pf)\ x & = & Pg\ ((Pf)\ x) = Pg\ \{f\ i \mid i \in x\} = \{g\ j \mid j \in \{f\ i \mid i \in x\}\} = \{g\ j \mid j = f\ i, i \in x\} \\ & = & \{g\ (f\ i) \mid i \in x\} = \{(g \circ f)\ i \mid i \in x\} = P\ (g_{b\to c} \circ f_{a\to b})\ x \end{array}$$

   According to the **axiom of extensionality** of functions, we have $Pg \circ Pf = P\ (g \circ f)$.

Haskell 中对 Functor 的定义

- Functor 这个概念被表示为一个 type class

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

-- 且满足如下两个性质：
-- Identity:     fmap id == id
-- Composition: fmap (f . g) == fmap f . fmap g
```

  - 也即：在 Haskell 语言中，可以将一个仅具有一个类型参数的 type constructor f 声明为是一个 Functor，只要你能提供一个合适的 fmap 函数。

- “Haskell 中的 Functor 概念” 符合 “CT 中的 Functor 概念”；这一点你能意识到吗?

  - f :: $\mathsf{Fn}_0 \to \mathsf{Fn}_0$; 定义了 Functor 中的 **object-mapping**
  - fmap :: $\mathsf{Fn}_1 \to \mathsf{Fn}_1$; 定义了 Functor 中的 **morphism-mapping**
  - f 和 fmap 两者组合在一起，定义了一个Functor $F : \mathsf{Fn} \to \mathsf{Fn}$
    - 显然可知：F 是一个 Endofunctor（自函子）

Kleisli Category and Monad

Model *side effects / non-pure functions* in CT

- An example: functions that log or trace their execution.
- The following gives such a function in C++ language.

```
string logger;

bool negate(bool b) {    // This function is not a pure function,
  logger += "Not so! ";  // since it mutates a global state.
  return !b;
}
```

- In modern programming, we try to stay away from global mutable state as much as possible. *You would never put code like this in a library*.
- It's possible to make this function pure: you just have to pass the log explicitly, in and out.

```
pair<bool, string> negate(bool b, string logger) {
  return make_pair(!b, logger + "Not so! ");
}
```

- It's also not a very good interface for a library function.
  - The callers are free to ignore the string in the return type, so that's not a huge burden;
  - But they are forced to pass a string as input, which might be inconvenient.

Is there a way to do the same thing better?

- Idea 1: Seperate the *aggregating-log-message* concern from the function.

```
pair<bool, string> negate(bool b) {
  return make_pair(!b, "Not so! ");
}
```

- Idea 2: Log-messages generated by functions are aggregated between function calls.
- To see how this can be done, let's switch to a slightly more realistic example:
  - A function from string to string that turns lower case characters to upper case.

```
string toUpper(string s) {
  string result;
  // ... code omitted
  return result;
}
```

  - A function that splits a string into a vector of strings, breaking it on whitespace boundaries

```
vector<string> toWords(string s) {
  vector<string> result;
  // ... code omitted
  return result;
}
```

- Then, we modify the two functions to *embellish* their regular return values with a log-message.

```
template<class A> using Writer = pair<A, string>;

Writer<string> toUpper(string s) {
  string result;
  // ... code omitted
  return make_pair(result, "toUpper ");
}
```

```
Writer<vector<string>> toWords(string s) {
  vector<string> result;
  // ... code omitted
  return make_pair(result, "toWords ");
}
```

- Then, we compose them into another embellished function that uppercases a string and splits it into words.

```
Writer<vector<string>> process(string s) {
  auto p1 = toUpper(s);
  auto p2 = toWords(p1.first);
  return make_pair(p2.first, p1.second + p2.second);
}
```

- We have accomplished our goal:
  - The aggregation of the log is no longer the concern of individual functions.
  - Functions produce their own messages, which are then, externally, concatenated into a larger log.

A still existing drawback

- Now imagine a whole program written in this style.
- It’s a nightmare of repetitive, error-prone code.
- But we are programmers. We know how to deal with repetitive code: we abstract it!
- Before we write more code, let’s **analyze the problem from the categorical point of view**.

Kleisli Category and Monad

The Writer Category

The Writer Category

- The idea of "*embellishing the return types of functions in order to add some additional functionality*" turns out to be very fruitful; *We'll see many more examples of it*.
- We will implement this idea by construct a **Writer** category:
  - The starting point is the Fn category.
  - We'll leave objects unchanged, but **redefine our morphisms to be the embellished functions**.
- For instance, we embellish the function **isEven** that goes from **int** to **bool**.

```
pair<bool, string> isEven(int n) {
  return make_pair(n % 2 == 0, "isEven ");
}
```

- **The important point**:
  - This function is still considered a morphism from **int** and **bool**, even though it returns a pair.
- By the laws of a category, we should be able to compose this morphism with another morphism that goes from **bool** to whatever object; for example, compose it with our earlier **negate** function:

```
pair<bool, string> negate(bool b) {
  return make_pair(!b, "Not so! ");
}
```

The Writer Category

```
pair<bool, string> isEven(int n) {
  return make_pair(n % 2 == 0, "isEven ");
}
```

```
pair<bool, string> negate(bool b) {
  return make_pair(!b, "Not so! ");
}
```

- The two morphisms cannot be composed as function composition, because the input and output mismatch.

  Their composition should look more like this:

```
pair<bool, string> isOdd(int n) {
  pair<bool, string> p1 = isEven(n); pair<bool, string> p2 = negate(p1.first);
  return make_pair(p2.first, p1.second + p2.second);
}
```

- This composition can be abstract as a higher-order function in C++

```
template<class A, class B, class C>
function<Writer<C>(A)> compose(function<Writer<B>(A)> m1, function<Writer<C>(B)> m2) {
  return [m1, m2](A x) {
    auto p1 = m1(x); auto p2 = m2(p1.first); return make_pair(p2.first, p1.second + p2.second);
  };
}
```

- The next thing: **The identity morphisms** in our new category. [*Laws of categories should be checked.*]

```
template<class A> Writer<A> identity(A x) { return make_pair(x, ""); }
```

The Writer Category in Haskell

- Define the **Writer** type:

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

- The morphisms are functions from arbitrary type to some **Writer** type:

```
a -> Writer b
```

- The composition is defined as a **fish** operator:

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

- The identity morphisms are defined as a **return** function

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

The Writer Category in Haskell

- The Haskell versions of the embellished functions **upCase** and **toWords**:

```
upCase :: String -> Writer String
upCase s = (map toUpper s, "upCase ")

toWords :: String -> Writer [String]
toWords s = (words s, "toWords ")
```

- The composition of the two functions is accomplished with the help of the fish operator:

```
process :: String -> Writer [String]
process = upCase >=> toWords
```

Kleisli Category

- This **Writer** category is an example of the so called **Kleisli** category – a category based on a **monad**.
  - Later we will see this monad.
- For our limited purposes, in a **Kleisli** category:
  - Objects are the types of the underlying programming language.
  - Morphisms from type A to type B are functions that go from A to a type derived from B using the particular embellishment.
    - Later we'll see that "embellishment" corresponds to the notion of an *endofunctor* in a category.
- Each Kleisli category defines its own way of composing such morphisms, as well as the identity morphisms with respect to that composition.
-
- Kleisli categories gives us more flexibilities to play with the Fn category.

Monad

- Programmers have developed a whole mythology around the monad.
  - For many programmers, the moment when they understand the monad is like a mystical experience.
- The whole mysticism around the monad is the result of a **misunderstanding**.
  - The monad is a very simple concept.
  - It's the diversity of applications of the monad that causes the confusion.
    - The monad abstracts the essence of so many diverse constructions that we simply don't have a good analogy for it in everyday life.
    - We are like those blind men touching different parts of the elephant and exclaiming triumphantly: "It's a rope," "It's a tree trunk," or "It's a burrito!"

Kleisli Category

- We have previously arrived at the **writer** monad by embellishing regular functions.
- The particular embellishment was done by pairing their return values with strings or, more generally, with elements of a monoid.
- We can now recognize that such an embellishment is a functor:

```
newtype Writer w a = Writer (a, w)

instance Functor (Writer w) where
  fmap f (Writer (a, w)) = Writer (f a, w)
```

- We have subsequently found a way of composing embellished functions, which are functions of the form:

```
a -> Writer w b
```

- It was inside the composition that we implemented the accumulation of the log messages.

Kleisli Category: A more general definition

Given a category C and an endofunctor $m : \mathsf{C} \to \mathsf{C}$, the corresponding Kleisli category K is defined by:

1. $\mathsf{K}_0 \doteq \mathsf{C}_0$
2. $\mathsf{K}_1 \doteq \{f_{a \to b} \mid f_{a \to b} \text{ is a morphism } f'_{a \to m\, b} \in \mathsf{C}_1\}$
3. $\mathsf{dom} f_{a \to b} \doteq a$, and $\mathsf{cod} f_{a \to b} \doteq b$
4. $1_a \doteq \mathsf{return}_{a \to m\, a}$, where $\mathsf{return}_{a \to m\, a} \in \mathsf{C}_1$
5. $(g_{b \to c} \circ f_{a \to b})_{a \to c} \doteq f' \rightarrowtail g'$, where $\rightarrowtail$ is a composition operation that assigns to each pair $(f'_{a \to m\, b},\ g'_{b \to m\, c})$ of morphisms in C_1 a morphism $h'_{a \to m\, c}$ in C_1, denoted by $f' \rightarrowtail g'$.

These data are subject to the following two axioms:

1. **Unit** : $f \circ 1 = f = 1 \circ f$

   > That is: $\mathsf{return}_{a \to m\, a} \rightarrowtail f'_{a \to m\, b} \;=\; f'_{a \to m\, b} \;=\; f'_{a \to m\, b} \rightarrowtail \mathsf{return}_{b \to m\, b}$

2. **Associativity** : $h \circ (g \circ f) = (h \circ g) \circ f$

   > That is: $(f' \rightarrowtail g') \rightarrowtail h' \;=\; f' \rightarrowtail (g' \rightarrowtail h')$

Then, the functor m is called a **monad**.

Kleisli Category in Haskell

- In Haskell
  - Kleisli composition is defined using the fish operator **>=>**, and
  - the identity morphism is a polymorphic function called **return**.
- Here's the definition of a monad using Kleisli composition:

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

  - Keep in mind that there are many equivalent ways of defining a monad, and that this is not the primary one in the Haskell ecosystem.
- In this formulation, monad laws are very easy to express

```
(f >=> g) >=> h  ==  f >=> (g >=> h)      -- associativity
return >=> f  ==  f                       -- left unit
f >=> return  ==  f                       -- right unit
```

Kleisli Category in Haskell

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

-----------------------------------------------------------

(f >=> g) >=> h  ==  f >=> (g >=> h)      -- associativity
return >=> f  ==  f                      -- left unit
f >=> return  ==  f                      -- right unit
```

- This kind of a definition also expresses what a monad really is:
  - It's a way of composing embellished functions.
    - It's not about side effects or state. It's about composition.
- As we'll see later, embellished functions may be used to express a variety of effects or state, but that's not what the monad is for.

The Writer Category in Haskell

- The logging functions (the Kleisli morphisms for the **Writer** functor) form a category because **Writer** is a monad:

```haskell
instance Monoid w => Monad (Writer w) where

  f >=> g = \a -> let Writer (b, s ) = f a
                      Writer (c, s') = g b
                  in Writer (c, s `mappend` s')

  return a = Writer (a, mempty)
```

  - Monad laws for **Writer w** are satisfied as long as monoid laws for **w** are satisfied.

Fish Anatomy

When implementing the fish operator for different monads you quickly realize that a lot of code is repeated and can be easily factored out.

- To begin with, the Kleisli composition of two functions must return a function, so its implementation may as well start with a lambda taking an argument of type **a**:

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \a -> ...
```

- The only thing we can do with this argument is to pass it to f:

```
f >=> g = \a -> let mb = f a
                in ...
```

- At this point we have to: produce the result of type **m c**, from an object of type **m b** and **g :: b -> m c**
- Let's define a function that does that for us.
  - This function is called **bind** and is usually written in the form of an infix operator:

```
(>>=) :: m a -> (a -> m b) -> m b
```

- For every monad, instead of defining the **fish** operator, we may instead define **bind**.

- In fact, the standard Haskell definition of a monad uses **bind**:

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

- Here's the definition of bind for the Writer monad:

```
(Writer (a, w)) >>= f = let Writer (b, w') = f a
                        in  Writer (b, w `mappend` w')
```

  - It is indeed shorter than the definition of the fish operator.

Bind Anatomy

It's possible to further dissect bind, taking advantage of the fact that **m** is a functor.

```
(>>=) :: m a -> (a -> m b) -> m b
```

- We can use **fmap** to apply the function **a -> m b** to the contents of **m a**.
- The result of the application is therefore of type **m (m b)**; *this is not exactly what we want*.
- All we need is a function that collapses the double application of **m**. Such a function is called **join**:

```
join :: m (m a) -> m a
```

- Using **join**, we can rewrite **bind** as:

```
ma >>= f = join (fmap f ma)
```

- That leads us to the third option for defining a monad:

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

The do Notation

- The do notation is just syntactic sugar for monadic composition.
  - On the surface, it looks a lot like imperative code,
  - but it translates directly to a sequence of binds and lambda expressions.
- For instance, take the example we used previously:

```
process :: String -> Writer String [String]
process = upCase >=> toWords
```

-

- In the do notation it would look like this:

```
process s = do
  upStr <- upCase s
  toWords upStr
```

- This do block is desugared by the compiler to:

```
process s =
  upCase s >>= \upStr ->
  toWords upStr
```

Monads and Effects

Monads and Effects

- Now we know what the monad is for — it lets us *compose embellished functions*.
- **Why embellished functions are so important in functional programming**.
- We've already seen one example, the **Writer** monad, where
  - Embellishment let us create and accumulate a log across multiple function calls.
  - A problem that would otherwise be solved using impure functions.
- In the following, we will see more examples of monads:
  - **What's really amazing**: the same pattern of embellishing the function return types works for a large variety of problems that normally would require abandoning purity.

Example 1: Nondeterminism – The List Monad

- If a function can return many different results, it may as well return them all at once.
- Semantically, **a non-deterministic function** is equivalent to *a function that returns a list of results*.
- This makes a lot of sense in a lazy language. For instance:
  - If all you need is one value, you can just take the head of the list, and the tail will never be evaluated.
  - If you need a random value, use a random number generator to pick the n-th element of the list.
  - Laziness even allows you to return an infinite list of results.
- In the List monad, **join** is implemented as **concat**.

```
instance Monad [] where
  --join :: [[a]] -> [a]
    join = concat
  --return :: a -> [a]
    return x = [x]
  --(>>=) :: [a] -> (a -> [b]) -> [b]
    ma >>= f = join (fmap f ma)
```

An Example of The List Monad

- The program that generates Pythagorean triples

```haskell
triples = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  guard (x^2 + y^2 == z^2)
  return (x, y, z)

guard :: Bool -> [()]
guard True = [()]
guard False = []
```

- The problem that normally would require a set of three nested loops has been dramatically simplified with the help of the List monad and the do notation.
- Haskell let's you simplify this code even further using list comprehension:

```haskell
triples = [(x, y, z) | z <- [1..], x <- [1..z], y <- [x..z], x^2 + y^2 == z^2]
```

  - This is just further syntactic sugar for the List monad.

Example 2: Read-Only State – The Reader Monad

- **A function that has read-only access to some external state**, can be always replaced by *a function that takes that external state as an additional argument*.
- A pure function **(a, e) -> b** (where **e** is the type of the external state) doesn't look like a Kleisli morphism.
- But as soon as we curry it to **a -> (e -> b)** we recognize the embellishment:

```
newtype Reader e a = Reader (e -> a)
```

- You may interpret a function returning a Reader as producing a mini-executable:
  - An action that given an external state produces the desired result.
- There is a helper function **runReader** to execute such an action:

```
runReader :: Reader e a -> e -> a
runReader (Reader f) e = f e
```

- The Reader e Monad:

```
instance Monad (Reader e) where
  -- (>>=) :: Reader e a -> (a -> Reader e b) -> Reader e b
    ra >>= k = Reader (\e -> runReader (k (runReader ra e)) e)
  --return :: a -> Reader e a
    return x = Reader (\e -> x)
```

Example 3: Write-Only State – The Writer Monad

- This is just our initial logging example. The embellishment is given by the Writer functor:

```
newtype Writer w a = Writer (a, w)
```

- For completeness, there's also a trivial helper runWriter that unpacks the data constructor:

```
runWriter :: Writer w a -> (a, w)
runWriter (Writer (a, w)) = (a, w)
```

- As we've seen before, in order to make Writer composable, w has to be a Monoid.
- Here's the monad instance for Writer written in terms of the bind operator:

```
instance (Monoid w) => Monad (Writer w) where
  -- (>>=) :: Writer w a -> (a -> Writer w b) -> Writer w b
    (Writer (a, w)) >>= k = let (a', w') = runWriter (k a)
                            in Writer (a', w `mappend` w')
  --return :: a -> Writer w a
    return a = Writer (a, mempty)
```

Example 4 : The State Monad

See the details in the other slides.

Example 5 : Exceptions

- An imperative function that throws an exception is really a *partial* function.
  - It's a function that's not defined for some values of its arguments.
- The simplest implementation of exceptions in terms of pure total functions uses the `Maybe` functor.
  - A partial function is extended to a total function that returns `Just a` whenever it makes sense, and `Nothing` when it doesn't.

```
instance Monad Maybe where
  Nothing >>= k = Nothing
  Just a >>= k = k a
  return a = Just a
```

- Monadic composition for `Maybe` correctly short-circuits the computation when an error is detected.
  - That's the behavior we expect from exceptions.

Example 6: Continuations

- It's the "*Don't call us, we'll call you!*" situation you may experience after a job interview.
- You are supposed to provide a **handler**, which is a function to be called with the *result*.
- This style of programming is especially useful when the *result* is not known at the time of the call, because, for instance, it's being evaluated by another thread or delivered from a remote web site.
- A Kleisli morphism in this case returns a function that accepts a handler **a -> r**; this handler represents **the rest of the computation** when the *result* **a** is known:

```
data Cont r a = Cont ((a -> r) -> r)
```

  - The handler **a -> r**, when it's eventually called, produces the result of type **r**, and this result is returned at the end.
- There is also a helper function for executing the action returned by the Kleisli morphism.

```
runCont :: Cont r a -> (a -> r) -> r
runCont (Cont k) h = k h
```

```haskell
data Cont r a = Cont ((a -> r) -> r)

runCont :: Cont r a -> (a -> r) -> r
runCont (Cont k) h = k h
```

- The signatures of the **bind** operator and **return** function.

```haskell
(>>=) ::      m a  ->  (a  ->       m b)  ->       m b
(>>=) :: Cont r a  ->  (a  ->  Cont r b)  ->  Cont r b

return :: a ->      m a
return :: a -> Cont r a
```

- The Cont r Monad:

```haskell
instance Monad (Cont r) where
  --(>>=) :: Cont r a  ->  (a  ->  Cont r b)  ->  Cont r b
  --                                                    :: Cont r b
  --                                                    vvvvvv
    ka >>= a2kb = Cont $ \hb        ->  runCont ka (\a -> runCont (a2kb a) hb)
  --                     ^^            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  --                     :: b -> r     :: r        ^^^^^^^^^^^^^^^^^^^^^^^^^
  --                                               :: a -> r

  --return :: a -> Cont r a
    return a = Cont $ \ha    ->   ha a
  --                  ^^          ^^^^
  --                  :: a -> r   :: r
```

THE END

Natural Transformation

自然变换

Natural Transformation / 自然変換

Given two categories C and D, and two functors $F_{\mathsf{C}\to\mathsf{D}}$ and $G_{\mathsf{C}\to\mathsf{D}}$, a **natural transformation** α from F to G, denoted by $\alpha : F \to G$, consists of the following data:

- for each object $c \in C_0$, an morphism $\alpha_c : F\ c \to G\ c$ in D_1.
  - Each α_c is called a **component** of the natural transformation.

These data are required to satisfy the following **axiom**:

- For any morphism $f : c \to c'$ in C_1

$$G\ f \circ \alpha_c = \alpha_{c'} \circ F\ f$$

- That is, the following square of morphisms in D *commutes*.