程序员的范畴论

作者：鲍志远

版本：0.1

以中有足乐者，不知口体之奉不若人也. —— 宋濂

第 1 章 范畴: 复合的本质

✍ **练习 1.1** Implement, as best as you can, the identity function in your favorite language (or the second favorite, if your favorite language happens to be Haskell).

解

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

✍ **练习 1.2** Implement the composition function in your favorite language. It takes two functions as arguments and returns a function that is their composition.

解

```
compose: ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c
let compose (g: 'b -> 'c) (f: 'a -> 'b) = fun x -> g (f x)
```

✍ **练习 1.3** Write a program that tries to test that your composition function respects identity.

解

代码:

```
copy: int -> string
copy_id: int -> string
id_copy: int -> string
let copy x = string_of_int x ^ string_of_int x
let copy_id = compose copy id
let id_copy = compose id copy
```

结果:

```
    Ocaml
# copy_id 42 = id_copy 42;;
- : bool = true
```

范畴由两部分组成: 对象 (object) 与态射 (morphism). 其中态射是有向的, 且有复合的性质, 或者说是传递性.

$$\forall x, y, z \in \mathcal{O} \quad (Mxy \wedge Myz) \rightarrow Mxz$$

✍ **练习 1.4** Is the world-wide web a category in any sense? Are links morphisms?

解

不是. web 的 link 更像是无向的连接, 并且也没有复合的性质.

✍ **练习 1.5** Is Facebook a category, with people as objects and friendships as morphisms?

解

不是. 关注关系虽然是有向的, 但没有没复合的性质. A 关注 B, B 关注 C, 并不代表 A 关注了 C.

✍ **练习 1.6** When is a directed graph a category?

解

用 V 表示节点的集合, E 表示有向边的集合, $e(x, y)$ 表示 E 中有一条起点为 x 终点为 y 的边.
有向图是范畴, 当:

$$\forall x, y, z \in V \quad (e(x, y) \wedge e(y, z)) \rightarrow e(x, z)$$

有向图是全范畴 (full category), 当:

$$[\forall x, y, z \in V \quad (e(x, y) \wedge e(y, z)) \rightarrow e(x, z)] \wedge [\forall v \in V \quad e(v, v)]$$

第 2 章 类型和函数

✍ **练习 2.1** Define a higher-order function (or a function object) memoize in your favorite language. This function takes a pure function f as an argument and returns a function that behaves almost exactly like f, except that it only calls the original function once for every argument, stores the result internally, and subsequently returns this stored result every time it' s called with the same argument. You can tell the memoized function from the original by watching its performance. For instance, try to memoize a function that takes a long time to evaluate. You' ll have to wait for the result the first time you call it, but on subsequent calls, with the same argument, you should get the result immediately.

解

代码:

```ocaml
let sleep_id n =
  Unix.sleep n;
  n

let memoize f =
  let cache = ref [] in
    fun x ->
      try List.assoc x !cache with
        Not_found ->
          let res = f x in
            cache := (x, res) :: !cache;
            res

let m_sleep_id = memoize sleep_id
```

结果:

```
    Ocaml
# m_sleep_id 3;;
- : int = 3 (after three seconds)
# m_sleep_id 3;;
- : int = 3 (immediately)
```

✍ **练习 2.2** Try to memoize a function from your standard library that you normally use to produce random numbers. Does it work?

解

No.

代码:

```ocaml
let m_int = memoize Random.int
```

结果:

```
    Ocaml
# m_int 100;;
- : int = 44
```

```
# m_int 100;;
- : int = 44
# m_int 100;;
- : int = 44
```

第一次调用 m_int 100 后产生的随机数被存在函数内部, 导致随后的调用都使用之前产生的结果, 也就是没有产生新的随机数.

✍ **练习 2.3** Most random number generators can be initialized with a seed. Implement a function that takes a seed, calls the random number generator with that seed, and returns the result. Memoize that function. Does it work?

解

Yes.

代码:

```
let seed_int seed = Random.init seed; Random.int
let m_seed_int = memoize seed_int
```

结果:

```
    Ocaml
# m_seed_int 3 100;;
- : int = 86
# m_seed_int 3 100;;
- : int = 70
# m_seed_int 3 100;;
- : int = 18
```

这一情况下, cache 中存的是函数, 是 Randm.init 3 产生的函数. 所以重复调用 m_seed_int 3 100 时, 跳过了根据种子产生随机函数这一步, 而直接使用之前产生的随机函数.

✍ **练习 2.4** Which of these C++ functions are pure? Try to memoize them and observe what happens when you call them multiple times: memoized and not.

1. The factorial function from the example in the text.
2. `std::getchar()`

3.
   ```
   bool f() {
       std::cout << "Hello!" << std::endl;
       return true;
   }
   ```

4.
   ```
   int f(int x) {
       static int y = 0;
       y += x;
       return y;
   }
   ```

解

只有 1 是纯函数.

memoize 它们之后, 只有 1 每次会返回正确的结果.

第一次调用, 每个函数的行为都与期望一致. 随后, 2 每次都返回第一次得到的字符, 而不会从输入中获得字符; 3 每次都直接返回 true, 而不会打印 Hello!; 4 只有在参数与之前不同时才会累加结果, 参数相同时直接返回之前的结果.

✍ **练习 2.5** How many different functions are there from `Bool` to `Bool`? Can you implement them all?

解

有四个不同的函数:

```
let same :bool -> bool = fun x ->
  match x with
    true -> true
  | false -> false

let neg :bool -> bool = fun x ->
  match x with
    true -> false
  | false -> true

let always_true: bool -> bool = fun x -> true

let always_false: bool -> bool = fun x -> false
```

✍ **练习 2.6** Draw a picture of a category whose only objects are the types `Void`, `()` (unit), and `Bool`; with arrows corresponding to all possible functions between these types. Label the arrows with the names of the functions.

解

$$\begin{tikzcd}
& Bool \arrow[loop above, "false"] \arrow[loop left, "true"] \arrow[loop right, "neg"] \arrow[loop below, "same"] \arrow[dr, "unit"] & \\
Void \arrow[ur, "absord"] \arrow[rr, "absord"] \arrow[loop below, "absord"] & & Unit \arrow[loop below, "id"] \arrow[ul, bend right, "true"'] \arrow[ul, bend left, "false"]
\end{tikzcd}$$

第 3 章 范畴大与小

从一个初始的有向图 $G = (V, E)$ 开始, 把节点看做对象, 有向边看做态射. 向图中添加态射, 直到使态射满足自反性, 传递性, 和结合性. 如果在这个过程中, 我们加入了尽量少的态射, 那么我们就构造了这个图的自由范畴 (free category).

注意到一个态射和自己复合完全可以产生新的态射. 比如: $+2 \circ +2 = +4$.

字符串拼接是一个极好的不满足交换律的运算的例子.

✍ **练习 3.1** Generate a free category from:

1. A graph with one node and no edges
2. A graph with one node and one (directed) edge (hint: this edge can be composed with itself)
3. A graph with two nodes and a single arrow between them
4. A graph with a single node and 26 arrows marked with the letters of the alphabet: a, b, c ... z.

解

1. $$\begin{tikzcd} A \arrow[loop above, "id"] \end{tikzcd}$$
   只有一个态射, id 和自己复合仍然是 id.
2. $$\begin{tikzcd} A \arrow[loop above, "id"] \arrow[loop above, distance=4em, "\vdots"] \end{tikzcd}$$
   有无数个 $A \to A$ 这样的态射, 其中有一个是 id, 其他的是复合得到的.
3. $$\begin{tikzcd} A \arrow[r] \arrow[r, bend left, "\vdots"] \arrow[r, bend right, "\vdots"'] \arrow[loop below, "id"] & B \arrow[loop below, "id"] \end{tikzcd}$$
   有无数个 $A \to B$ 的态射, 但是没有 $B \to A$ 的态射.
4. 有无数个 $A \to A$ 这样的态射, 但是态射的数量应该和 2 相同.

关于态射的顺序性质:

- 前序 (preorder): 有单位态射, 态射之间可以复合, 复合满足结合律.
- 偏序 (partial order): 前序加上态射的反对称性.
- 全序 (total order): 偏序加上态射的全连接性.

✍ **练习 3.2** What kind of order is this?

1. A set of sets with the inclusion relation: A is included in B if every element of A is also an element of B.
2. C++ types with the following subtyping relation: `T1` is a subtype of `T2` if a pointer to `T1` can be passed to a function that expects a pointer to `T2` without triggering a compilation error.

解

1. 偏序.
   将包含看做态射. 有单位态射, 每个集合都包含自身; 态射是可以复合的, A 包含 B, B 包含 C, 那么 A 包含 C; 复合是满足结合律的; (why?) 态射是反对称的, 如果 A 包含 B 且 B 包含 A, 那么 $A = B$; 态射不是全连接的, 两个集合间可以没有包含关系.
2. 前序.
   将子类型关系看做态射. 有单位态射, 每个需要`T1`指针的地方都可以传`T1`; 态射是可复合的, 需要`T1`指针的地方可以传`T1`的子类型, 也可以传`T1`子类型的子类型; 复合是满足结合律的; (why) 态射不是反对称的, 互为子类型不代表相等, (互相继承); 态射不是全连接的, 不是每一个类型都和别的类型有子类型关系.

一个幺半群 (monoid), 要求一个集合有一个二元操作, 该操作有单位元, 该操作满足结合律.

为什么一定是结合律? 我没有理解这个定义, 还停留在背诵阶段.

练习 3.3 Considering that `Bool` is a set of two values `True` and `False`, show that it forms two (set-theoretical) monoids with respect to, respectively, operator `&&` (AND) and `||` (OR).

解

首先`&&` 和 `||`都满足结合律, 这是定义保证的. 接下来我们把单位元找到: `True`是 `&&` 的单位元, `False`是 `||` 的单位元.

在范畴论中, 我们尝试摆脱集合和集合的元素, 转而关注对象和态射.

练习 3.4 Represent the `Bool` monoid with the AND operator as a category: List the morphisms and their rules of composition.

解

$$\begin{tikzcd} Bool \arrow[loop above, "(AND)\ False"] \arrow[loop below, "(AND)\ True"] \end{tikzcd}$$

在 hom-set $\mathcal{B}(Bool, Bool)$ 中, 我们有两个态射, 其中 $(AND)\ True$ 是单位态射. 设 $f = (AND)\ True$, $g = (AND)\ False$, 则复合规则为: $f \circ g = g$, $g \circ f = g$.

练习 3.5 Represent addition modulo 3 as a monoid category.

解

回忆 addition modulo m: $a +_m b = (a + b) \bmod m$, 显然我们在正整数范围内讨论比较方便.

$$\begin{tikzcd} Nat \arrow[loop above, "+_3 0"] \arrow[loop left, "+_3 1"] \arrow[loop below, "+_3 2"] \end{tikzcd}$$

在 hom-set $\mathcal{N}(Nat, Nat)$ 中, 我们有三个态射, 其中 $+_3 0$ 是单位态射. 将单位态射的复合省略后, 复合规则为:

- $+_3 1 \circ +_3 1 = +_3 2$
- $+_3 2 \circ +_3 2 = +_3 1$
- $+_3 1 \circ +_3 2 = +_3 2 \circ +_3 1 = +_3 0$

第 4 章 克莱斯利范畴

这一章, 我们在态射上做文章. 当考虑从对象 A 到对象 B 的态射时, 我们并不把该态射简单地认为是一个类型为 $A \to B$ 的函数, 而是一个类型为 $A \to (B, M)$ 的函数, 其中 (B, M) 表示一个二元组 (pair), M 是一个幺半群. 具有这样态射的范畴我们称其为克莱斯利范畴, 一个基于一元 (monad) 的范畴.

✍ **练习 4.1** Construct the Kleisli category for partial functions (define composition and identity).

解

部分函数的定义用 bool 替换了前文的 string. 这是合理的, 因为 bool 也是一个幺半群.

```
type 'a optional = {is_valid: bool; value: 'a}

let compose m2 m1 = fun x ->
  let {is_valid = v1; value = val1} = m1 x in
  let {is_valid = v2; value = val2} = m2 val1 in
    {is_valid = v1 && v2; value = val2}

let id x = {is_valid = true; value = x}
```

✍ **练习 4.2** Implement the embellished function `safe_reciprocal` that returns a valid reciprocal of its argument, if it’ s different from zero.

解

```
let safe_root x =
  if x >= 0.
  then {is_valid = true; value = sqrt x}
  else {is_valid = false; value = x}

let safe_reciprocal x =
  if x <> 0.
  then {is_valid = true; value = 1. /. x}
  else {is_valid = false; value = x}
```

✍ **练习 4.3** Compose the functions `safe_root` and `safe_reciprocal` to implement `safe_root_reciprocal` that calculates `sqrt(1/x)` whenever possible.

解 代码:

```
let safe_root_reciprocal = compose safe_root safe_reciprocal
```

结果:

```
    Ocaml
# safe_root_reciprocal 0.25;;
- : float optional = {is_valid = true; value = 2.}
# safe_root_reciprocal 4.;;
- : float optional = {is_valid = true; value = 0.5}
# safe_root_reciprocal 1.;;
- : float optional = {is_valid = true; value = 1.}
```

```
# safe_root_reciprocal (-1.);;
- : float optional = {is_valid = false; value = -1.}
```