

(Terrible) Solutions to Category Theory for Programmers by Bartosz Milewski

Syed Khalid

Nov, 2023

1 Ch-1 Category

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).

Answer:

```
function id(f) { return f }
```

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

Answer:

```
const compose = (funcA, funcB) => x => funcA(funcB(x));
```

3. Write a program that tries to test that your composition function respects identity.

Answer:

```
function square(x) { return x ** 2 }  
function sqrt(x) { return x ** (1 / 2) }  
  
// Test composition and identity  
// ... (your test code here)
```

4. Is the world-wide web a category in any sense? Are links morphisms?

Answer:

WWW is a category if and only if all sites link to each other and each site links to itself and since it actually doesn't, it's not.

- a. Objects - web pages
- b. Morphisms - linking two web pages
- c. Composition - clicking two links one after the other same as clicking direct link
- i) Associativity - since all sites are interlinked routes don't matter
- ii) Identity - going from H to A then A to A

5. Is Facebook a category, with people as objects and friendships as morphisms?

Answer:

Facebook

- a. Objects - people
- b. Morphisms - friendships
- c. Composition - if A is a friend of B and if B is a friend of C does not necessarily entail that A & C are friends

Hence no FB is not a category.

6. When is a directed graph a category?

Answer:

A directed graph is a category if and only if

- i) each vertex is linked to / has an adjacency to itself.
- ii) all vertices are interconnected with at least a one-way adjacency

2 Ch-2 Types

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.

Answer:

```
function memoize(f) { }
```

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

Answer:

```
% Your response here
```

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?

Answer:

```
% Your response here
```

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.

(a) The factorial function from the example in the text.

(b) `std::getchar()`

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

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

Answer:

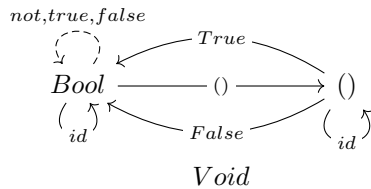
```
% Your response here
```

5. How many different functions are there from `Bool` to `Bool`? Can you implement them all?

Answer:

$$\begin{aligned} f &: \text{Bool} \rightarrow \text{Bool} \\ \text{id} &= (T,T), (F,F) \\ \text{not} &= (T,F), (F,T) \\ \text{tru} &= (T,T), (F,T) \\ \text{fal} &= (T,F), (F,F) \end{aligned}$$

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.



And also counter intuitively, you have single morphisms from *Void* to *Bool* and *()* called **absurd**. And also id_{Void} .

3 Ch-3 Categories Great and Small

1. **Generate a free category from:**

- (a) A graph with one node and no edges.



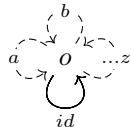
- (b) A graph with one node and one (directed) edge (hint: this edge can be composed with itself).



- (c) A graph with two nodes and a single arrow between them.



- (d) A graph with a single node and 26 arrows marked with the letters of the alphabet: a, b, c ... z.



2. **What kind of order is this?**

- (a) **A set of sets with the inclusion relation: *A* is included in *B* if every element of *A* is also an element of *B*.**

Same as saying, $A \subseteq B$. Now,

- If $A \subseteq B$ and $B \subseteq C$ then $A \subseteq C$
- If $A \subseteq B$ and $B \subseteq A$ then $A = B$.
- There can exist sets A, B such that $A \not\subseteq B$ and $B \not\subseteq A$.

Therefore this is a *partially ordered* set.

- (b) **What kind of order is this? 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.

TODO

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, the operator `&&` (AND) and `||` (OR).

$$\text{AND } False \begin{array}{c} \curvearrowright \\ \text{Bool} \\ \curvearrowleft \end{array} \text{AND } True=id$$

$$\text{OR } False=id \begin{array}{c} \curvearrowright \\ \text{Bool} \\ \curvearrowleft \end{array} \text{OR } True$$

In haskell:

```
empty1 = True
mappend1 = AND
empty2 = False
mappend2 = OR
```

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

Morphisms - AND True, AND False

Rule of composition - AND True \rightarrow AND False = AND False

(i.e AND True is id)

5. Represent addition modulo 3 as a monoid category.

$$id=+0 \begin{array}{c} \curvearrowright \\ \{0, 1, 2\} \\ \curvearrowleft \\ +1 \end{array} +2$$

4 Ch-4 Kleisli Categories

1. Construct the Kleisli category for partial functions (define composition and identity).

```
(>->) :: (a -> Maybe b) -> (b -> Maybe c) -> (a -> Maybe c)
f >-> g = \x -> case f x of
    Nothing -> Nothing
    Just b   -> g b
kid :: (a -> Maybe a)
kid = Just
```

2. Implement the embellished function *safe_reciprocal* that returns a valid reciprocal of its argument, if it's different from zero.

```
safe_reciprocal :: Double -> Maybe Double
safe_reciprocal 0 = Nothing
safe_reciprocal n = Just (1/n)
```

```
safe_root :: Double -> Maybe Double
safe_root n | n >= 0    = Just $ sqrt n
             | otherwise = Nothing
```

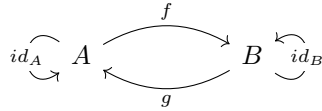
3. Compose the functions *safe_root* and *safe_reciprocal* to implement *safe_root_reciprocal* that calculates $\text{sqrt}(1/x)$ whenever possible.

```
safe_root_reciprocal :: Double -> Maybe Double
safe_root_reciprocal = safe_root >-> safe_reciprocal
```

5 Ch5 Products & Coproducts

1. Show that the terminal object is unique up to unique isomorphism.

Isomorphism between two objects is present when



$$g \circ f = id_A \quad (1)$$

$$f \circ g = id_B \quad (2)$$

The terminal object is the object with *one and only one* morphism coming to it from any object in the category.

Consider any two terminal objects T_1 & T_2 .

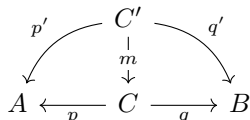
For T_2 we have a unique morphism from T_1 , let's name it : $f : T_1 \rightarrow T_2$. Similarly for T_1 we have $g : T_2 \rightarrow T_1$.

Also since both are objects in a category we have $id_{T_1} : T_1 \rightarrow T_1$ and $id_{T_2} : T_2 \rightarrow T_2$

Now $g \circ f : T_1 \rightarrow T_1$ and $f \circ g : T_2 \rightarrow T_2$ but there can since T_1 and T_2 are terminal objects the only morphisms which can go from itself to itself is its id. Therefore (like (1) and (2)) f, g are unique isomorphisms.

2. **What is a product of two objects in a poset?** Hint: Use the universal construction.

A product of A and B is C equipped with two projections ($p : C \rightarrow A$ and $q : C \rightarrow B$) such that for any other C' equipped with two projections ($p' : C' \rightarrow A$ and $q' : C' \rightarrow B$) there is a unique morphism $m : C' \rightarrow C$ such that $p' = p \circ m$ and $q' = q \circ m$.



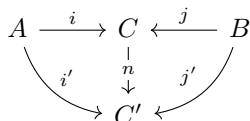
A poset is a set with some ordering (\leq) such that if $a \leq b$ and $b \leq a$ then $a = b$. Also not all elements have to have some ordering between them (that condition would make it a *total* order.).

Thus a product of two objects in a poset is the *greatest* element in the poset which is *lesser* than both the objects.

Example: Consider the poset of \mathbb{Z} . For $A = 0, B = 1$ the least element greater than the two is $C = 0$. All other $C' = \{n < 0\}$ have a morphism (m) to C that is \leq .

3. **What is a coproduct of two objects in a poset?**

A coproduct of A and B is C equipped with two injections ($i : A \rightarrow C$ and $j : B \rightarrow C$) such that for any other C' equipped with two injections ($i' : A \rightarrow C'$ and $j' : B \rightarrow C'$) there is an unique morphism $n : C \rightarrow C'$ such that $i' = n \circ i$ and $j' = n \circ j$.



Thus a coproduct of two objects in a poset is the *least* element in the poset which is *greater* than both the objects.

4. **Implement the equivalent of Haskell *Either* as a generic type in your favorite language (other than Haskell).**
5. **Show that *Either* is a "better" coproduct than *int* equipped with two injections:**

```

int i(int n) { return n; }
int j(bool b) { return b ? 0 : 1; }

```

Hint: Define a function

```

int m(Either const & e);

```

that factorizes i and j .

6. Continuing the previous problem: How would you argue that *int* with the two injections *i* and *j* cannot be "better" than *Either*?
7. Still continuing: What about these injections?

```
int i(int n) {
    if (n < 0) return n;
    return n + 2;
}

int j(bool b) { return b ? 0 : 1; }
```

8. Come up with an inferior candidate for a coproduct of *int* and *bool* that cannot be better than *Either* because it allows multiple acceptable morphisms from it to *Either*.

6 Ch6 Algebraic Data Types

1. Show the isomorphism between *Maybe a* and *Either () a*.

```
may2eit :: Maybe a -> Either () a
may2eit Nothing = Left ()
may2eit (Just x) = Right x

eit2may :: Either () a -> Maybe a
eit2may (Left ()) = Nothing
eit2may (Right x) = Just x
```

2. Here's a sum type defined in Haskell:

```
data Shape = Circle Float | Rect Float Float
```

When we want to define a function like *area* that acts on a *Shape*, we do it by pattern matching on the two constructors:

```
area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rect d h) = d * h
```

Implement *Shape* in C++ or Java as an interface and create two classes: *Circle* and *Rect*. Implement *area* as a virtual function.

3. Continuing with the previous example: We can easily add a new function *circ* that calculates the circumference of a *Shape*. We can do it without touching the definition of *Shape*:


```

circ :: Shape -> Float
circ (Circle r) = 2.0 * pi * r
circ (Rect d h) = 2.0 * (d + h)

```

Add *circ* to your Cpp or Java implementation. What parts of the original code did you have to touch?

4. Continuing further: Add a new shape, *Square*, to *Shape* and make all the necessary updates. What code did you have to touch in Haskell vs. Cpp or Java?
5. Show that $a + a = 2 \times a$ holds for types (up to isomorphism). Remember that 2 corresponds to *Bool*, according to our translation table.

$$a + a = 2 \times a$$

Either $a \cong (Bool, a)$

```

e2b :: Either a a -> (Bool, a)
e2b (Right x) = (True, x)
e2b (Left x) = (False, x)

b2e :: (Bool, a) -> Either a a
b2e (True, x) = Right x
b2e (False, x) = Left x

```

Thus they are isomorphic.

7 Ch7 Functors

1. Can we turn the *Maybe* type constructor into a functor by defining:

```
fmap _ _ = Nothing
```

which ignores both of its arguments?

We have the functor laws :

- (a) Preservation of identity :-

```

fmap id Nothing
    = Nothing -- applying our definition of fmap
    = id Nothing -- refactoring x to id x

fmap id (Just x)
    = Nothing -- applying our definition of fmap

```

```

    = id Nothing -- refactoring x to id x
    /= id Just x -- id has NOT been preserved

```

Which is not the same as k

(b) Preservation of composition :- It violates this too.

Thus **No**.

2. Prove functor laws for the reader functor.

The fmap definition for the *Reader* functor is,

```

fmap :: (a -> b) -> ((r -> a) -> (r -> b))
fmap = (.)

```

Say we have a function

```

fr :: (r -> a) -> (r -> b)
fr = undefined

```

(a) Preservation of identity :-

(b) Preservation of composition :-

3. Implement the reader functor in your second favorite language.

4. Prove the functor laws for the list functor. Assume that the laws are true for the tail part of the list you're applying it to (use induction.)