Category theory for Programmers Solutions

littlekuo

Contents

Chapter One

Category: The Essence of Composition

Exercise 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).

Solution.

```
auto id = [](auto x){ return x; }
```

□

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

Solution.

```
auto compose = [](auto f, auto g) {
 return [=](auto&& ... x) {
   return f(g(x...));
 };
};
```

□

Exercise 1.3.
Write a program that tries to test that your composition function respects identity.

Solution.

```
bool test_compose(){
  auto f = [](int str){
    return std::to_string(str);
  };
  auto f_id = compose(f, id);
  auto id_f = compose(id, f);
  for(int i = 0; i < 1000; i++){
    auto expect = f(i);
    auto r1 = f_id(i), r2 = id_f(i);
    if(r1 != expect || r2 != expect)
       return false;
  }
    return true;
}
```

□

Exercise 1.4.
Is the world-wide web a category in any sense? Are links morphisms?

Solution. If assume $A \to B$ is a morphism iff page B is reachable from page A in 0 or more steps, the world-wide web is a category.
Links are not morphisms, because if there is a link on page A to page B, and a link on page B to page C, there not necessarily a link on page A to page C. □

Exercise 1.5.
Is the world-wide web a category in any sense? Are links morphisms?

Solution. If assume $A \to B$ is a morphism iff page B is reachable from page A in 0 or more steps, the world-wide web is a category.
Links are not morphisms, because if there is a link on page A to page B, and a link on page B to page C, there not necessarily a link on page A to page C. □

Exercise 1.6.
Is Facebook a category, with people as objects and friendships as morphisms?

Solution. It's not a category. Because if A and B are friends, and B and C are friends, A and C may not know each other. □

Chapter Two

Types and Functions

Exercise 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.

Solution.

```
template <typename T>
struct Memoize;

template<typename ResultType, typename... ArgTypes>
class Memoize<ResultType(ArgTypes...)>
{
private:
   using ArgsType = std::tuple<ArgTypes...>;
public:
   Memoize(std::function<ResultType(ArgTypes...)> f) : _f(f) {}

   ResultType operator()(ArgTypes... args) {
     const auto argsAsTuple = std::make_tuple(args...);
     auto memoized = _table.find(argsAsTuple);
     if(memoized == _table.end()) {
       auto const r = _f(args...);
       _table.insert({ {args...}, r});
       return r;
     } else {
       std::cout << "memoized: ->" << memoized->second << std::endl;
       return memoized->second;
     }
   }
private:
   std::map<ArgsType, ResultType> _table;
   std::function<ResultType(ArgTypes...)> _f;
};
```

□

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

Solution.

```
auto memoizedRand = Memoize<int(void)>(rand);
std :: cout << memoizedRand() << std::endl;
std :: cout << memoizedRand() << std::endl;
```

It always returns the same result. From the point of produce random numbers, it does not work. ☐

Exercise 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?

Solution.

```
int randNum(int seed){
   srand(seed);
   return rand();
}
auto memoizedRand = Memoize<int(int)>(randNum);
```

this will always return the same random number for the same seed. ☐

Exercise 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.
(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;
}
```

Solution.
(a) It is. there are only local variables, and return the same value when argument is the same.

(b) It is not pure.
(c) This function is not pure, since it outputs "Hello"(have side effects).
(d) It is not pure, because for the same argment, the return value may different. □

Exercise 2.5. How many different functions are there from Bool to Bool? Can you implement them all?

Solution.

```
bool id(bool x){
  return x;
}

bool altrue(bool x){
  return true;
}

bool alfalse(bool x){
  return false;
}

bool neg(bool x){
  return !x;
}
```

□

Exercise 2.6. Draw a picture of a category whose only objects are the types $Void, ()(unit), andBool;$ with arrows corresponding to all possible functions between these types. Label the arrows with the names of the functions

Solution.

$$\begin{tikzcd}
Void \arrow[loop above, "\text{id}"] \arrow[r, "\text{absurd}"] \arrow[dr, "\text{absurd}"'] & () \arrow[loop above, "\text{id}"] \arrow[d, bend right, "\text{fTrue}"'] \arrow[d, "\text{fFalse}"] \\
 & Bool \arrow[u, bend right=60, "\text{unit}"'] \arrow[loop right, "\text{id}"] \arrow[loop below, "\text{alfalse}"] \arrow[loop left, "\text{neg}"] \arrow[loop, out=-30, in=-60, distance=2em, "\text{altrue}"]
\end{tikzcd}$$

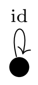
□

Chapter Three

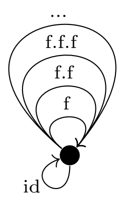
Categories Great and Small

Exercise 3.1. 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: $a, b, c...z$.

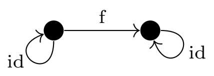
Solution.
(a) only one object and one morphism



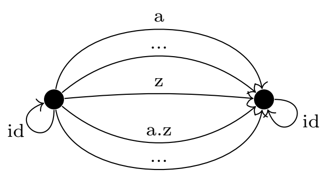
(b) one object ; since the edge can be composed with itself, we may have infinitely many composed morphisms.



(c) only one object and three morphisms.



(d) one object, similar to (b), we may have infinitely many composed morphisms.



□

Exercise 3.2. 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.
(b) 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.

Solution.
(a) Define $A \leq B$ as $A \subseteq B$. For every set A, $A \subseteq A$. $\subseteq$ is composable, $A \subseteq B$ and $B \subseteq C$ implies $A \subseteq C$. This means $\subseteq$ is a preorder. If $A \subseteq B$ and $B \subseteq A$ then $A = B$ which means it is a partial order. It is not a total order because $\{1\} \nsubseteq \{2\}$ and $\{2\} \nsubseteq \{1\}$.
(b) Define $A \leq B$ as A is subtype of B. Similar to (a), it is at least a preorder. □

Exercise 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).

Solution.
For operator &&, a && $(b$ && $c) = (a$ && $b)$ && c (check eight cases), so it is associative. The special element is $True$.
For operator ||, it is also associative. The special element is $False$.
□

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

Solution.
the morphsims are $True, False$, the compositon of f, g is $f\&\&g$. □

Exercise 3.5. Represent addition modulo 3 as a monoid category.

Solution.
the morphsims are $[0], [1], [2]$, the compositon of $[a]$ and $[b]$ is $[(a + b)\ mod\ 3]$. □

Chapter Four

Kleisli Categories

Exercise 4.1. Construct the Kleisli category for partial functions (define composition and identity).

Solution.

```
template<class A, class B, class C>
std::function<optional<C>(A)> compose(std::function<optional<C>(B)> m2,
     std::function<optional<B>(A)> m1) {
     return [m1, m2](A x) {
       auto p1 = m1(x);
       if (!p1.isValid()) return optional<C>{};
       auto p2 = m2(p1.value());
       return p2;
     };
}
template<class A> optional<A> identity(A x) {
    return optional<A>{x};
}
```

□

Exercise 4.2. Implement the embellished function **safe_reciprocal** that returns a valid reciprocal of its argument, if it's different from zero.

Solution.

```
optional<double> safe_reciprocal(double x) {
  if (x != 0) return optional<double>{1 / x};
  else return optional<double>{};
}
```

□

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

Solution.

```
optional<double> safe_root_reciprocal(double x) {
   return compose(safe_root, safe_reciprocal)(x);
}
```

□

Chapter Five

Products and Coproducts

Exercise 5.1. Show that the terminal object is unique up to unique isomorphism

Solution. Suppose A and B are terminal objects, then there is exactly one morphism $f : A \to B$ and exactly one morphism $g : B \to A$(By definition of terminal object), So $fg : B \to B$ and $gf : A \to A$, while $id_B : B \to B$ and $id_A : A \to A$, we get $fg = id_B \ and \ gf = id_A$ (By definition of terminal object), thus any two terminal objects are isomorphic. □

Exercise 5.2. What is a product of two objects in a poset? Hint: Use the universal construction

Solution. Suppose the product of two objecs a, b is c , we must have two projections:

$$c \leq a \ and \ c \leq b$$

and for any other object c' equipped with two projections:

$$c' \leq a \ and \ c' \leq b$$

then $c' \leq c$, so the product of two objects in a poset is their greatest lower bound. □

Exercise 5.3. What is a coproduct of two objects in a poset?

Solution. Suppose the product of two objecs a, b is c , we must have two injections:

$$a \leq c \ and \ b \leq c$$

and for any other object c' equipped with two projections:

$$a \leq c' \ and \ b \leq c'$$

then $c \leq c'$, so the coproduct of two objects in a poset is their least upper bound. □

Exercise 5.4. Implement the equivalent of Haskell *Either* as a generic type in your favorite language (other than Haskell).

Solution.

□

Exercise 5.5. 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.

Solution.

```
int m(Either const & e) {
 if (e.isLeft()) {
   return e.left;
 }
 return e.right?: 0; 1;
}
```

□

Exercise 5.6. Continuing the previous problem: How would you argue that *int* with the two injections i and j cannot be "better" than *Either*?

Solution. Suppose function

```
Either m(int e);
```

factorizes function Left and Right. then

```
Left = m.i
Right = m.j
```

So

```
Left(0) = m(0);
Right(true) = m(0);
```

it means m(0) have different values which is impossible. □

Exercise 5.7. 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; }
```

Solution. Define function

```
Either m(int e) {
  if(e == 0){
     return Right(True);
  }
  if(e == 1){
     return Right(False);
  }
  if(e < 0){
     return Left(e);
  }
  else {
     return Left(e-2);
  }
}
```

we can check $m.i = Left$ and $m.j = Right$. □

Exercise 5.8. 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*.

Solution.

```
data Tup = IntPair Int Int | BoolPair Bool Bool

-- two injection
intToTup :: Int -> Tup;
intToTup x = IntPair x x

boolToTup :: Bool -> Tup;
boolToTup x = BoolPair x x
```

Obviously there have many morphisms from it to Either

```
m :: Tup -> Either Int Bool;
m (IntPair x y) = if x == y then (Left x) else (Left 0)
m (BoolPair x y) = if x == y then (Right x) else (Right False)

m1 :: Tup -> Either Int Bool;
m1 (IntPair x y) = if x == y then (Left x) else (Left 1)
m1 (BoolPair x y) = if x == y then (Right x) else (Right False)

m2 :: Tup -> Either Int Bool;
m2 (IntPair x y) = if x == y then (Left x) else (Left 3)
m2 (BoolPair x y) = if x == y then (Right x) else (Right True)
```

□