



# Category Theory for Programmers

## Homework (Session 3)

Bruno Vandekerkhove

## Contents

<b>Definition of a Functor</b>	<b>1</b>
<b>Challenges</b>	<b>1</b>
2.1 Chapter 6	1
Isomorphism of <code>Maybe a</code> and <code>Either () a</code>	1
Circles and rectangles	2
Show that <code>a + a = 2 × a</code>	3
2.2 Chapter 7	3
Implementing the <code>Reader</code> functor	4
<b>References</b>	<b>4</b>

## Definition of a Functor

A definition of functors from Mac Lane's book[1] :

A *functor* is a morphism of categories. In detail, for categories  $C$  and  $B$  a functor  $T : C \rightarrow B$  with codomain  $C$  and codomain  $B$  consists of two suitably related functions: The *object function*  $T$ , which assigns to each object  $c$  of  $C$  an object  $Tc$  of  $B$  and the *arrow function* (also written  $T$ ) which assigns to each arrow  $f : c \rightarrow c'$  of  $C$  an arrow  $Tf : Tc \rightarrow Tc'$  of  $B$ , in such a way that

$$T(1_c) = 1_{Tc}, \quad T(g \circ f) = Tg \circ Tf,$$

the latter whenever the composite  $g \circ f$  is defined in  $C$ .

The similarities are obvious. A functor is a mapping of both objects and their morphisms from one category to an other such that identity and composition ('the structure') are preserved. An interesting simplification is *nLab's 'a functor preserves commuting diagrams.'*[2]. A commuting diagram is a diagram where any directed path from a given start to a given end point leads to the same results. Both identities and compositions correspond to such diagrams.

If we consider arrow-only categories (objects are represented by their identities) then an arrow-only definition of a functor follows.

## Challenges

### Chapter 6

#### Isomorphism of `Maybe a` and `Either () a`

Isomorphism implies the existence of an invertible morphism, which in this case can be defined as follows :

```

1 func :: Maybe a -> Either () a
2 func Nothing = Left ()
3 func (Just x) = Right x
4
5 func_inv :: Either () a -> Maybe a
6 func_inv (Left ()) = Nothing
7 func_inv (Right x) = Just x

```

## Circles and rectangles

Let's start from a base implementation :

```

1  #include <iostream>
2  #include <math.h>
3
4  class Shape {
5      virtual float area() = 0;
6  };
7
8  class Circle: public Shape {
9      float _radius;
10 public:
11     Circle(float radius) { _radius = radius; }
12     float area() { return M_PI * _radius * _radius; }
13 };
14
15 class Rectangle: public Shape {
16     float _width, _height;
17 public:
18     Rectangle(float width, float height) { _width = width; _height = height; }
19     float area() { return _width * _height; }
20 };

```

We can add an operation to calculate the circumference :

```

1  ...
2
3  class Shape {
4      virtual float area() = 0;
5      virtual float circ() = 0;
6  };
7
8  class Circle: public Shape {
9      ...
10     float circ() { return 2 * M_PI * _radius; }
11 };
12
13 class Rectangle: public Shape {
14     ...
15     float circ() { return 2 * (_width + _height); }
16 };

```

Clearly, a new virtual function had to be added to the `Shape` class and implementations had to be coded for each subtype. Finally, let's add a new type of shape, a square :

```

1  ...
2
3  class Square: public Shape {
4      float _width;
5  public:
6      Square(float width) { _width = width; }
7      float area() { return _width * _width; }
8      float circ() { return 4 * _width; }
9  };

```

This time only some subtype `Square` had to be added and the virtual functions had to be implemented. In Haskell one could alter the definition of `Shape` and change each operation's implementation (adding a case for both `area` and `circ`).

This is a reference to the *expression problem*; in functional programming languages it's easier to add operations to existing data types, but it's more difficult to add new data types. The opposite holds for object-oriented languages, where the visitor pattern helps to turn the problem on its head but does not solve it.

**Show that  $a + a = 2 \times a$**

We can simply substitute  $1 + 1$  for  $2$  and use the distributivity property :

$$2 \times a = (1 + 1) \times a = 1 \times a + 1 \times a = a + a$$

Or in Haskell :

```

1  func :: Either a a -> (Bool, a)
2  func (Left x) = (True, x)
3  func (Right x) = (False, x)
4
5  func_inv :: (Bool, a) -> Either a a
6  func_inv (True, x) = Left x
7  func_inv (False, x) = Right x

```

## Chapter 7

Proving the functor laws means that we have to show that `fmap` preserves identity and composition. When we ignore both arguments when lifting :

```

1  fmap _ _ = Nothing

```

Then the identity function is not always preserved :

```

1  fmap id (Just x) = Nothing = id Nothing /= id (Just x)

```

The laws do hold for the reader functor :

```

1  fmap id f = (.) id f = id f = f
2  fmap (h . g) f = (h . g) . f = h . (g . f) = h . (fmap g f) = fmap h (fmap g f)

```

For the list functor they hold as well. We start with proving that identity is preserved :

```

1  fmap id Nil = Nil = id Nil
2  fmap id (Cons x t) = Cons (id x) (fmap id t) = Cons x t

```

The last line assumes that the law holds for the tail of the list. Composition is also preserved :

```

1  fmap (f . g) Nil = Nil = fmap g Nil = fmap f (fmap g Nil)
2  fmap (f . g) (Cons x t)
3      = Cons ((f . g) x) (fmap (f . g) t)
4      = Cons ((f . g) x) (fmap f (fmap g t))
5      = fmap f (Cons (g x) (fmap g t))
6      = fmap f (fmap g (Cons x t))

```

### Implementing the Reader functor

The Reader functor takes a mapping  $a \rightarrow b$  and transforms it to a mapping  $(r \rightarrow a) \rightarrow (r \rightarrow b)$ . In Scala :

```

1  object Reader extends App {
2      print(new Reader[String].fmap((i: Int) => 10 + i)(s => s.toInt * 2)("10")) // 30
3  }
4
5  class Reader[R] extends Functor[({type Func[X] = (R => X)})#Func] {
6      override def fmap[A, B](f: A => B)(g: R => A): R => B = g andThen f
7  }
8
9  trait Functor[F[_]] {
10     def fmap[A, B](f: (A => B))(g: F[A]) : F[B]
11 }

```

### References

- [1] Saunders Mac Lane. Categories for the working mathematician, 1978.
- [2] nLab. Functor in nlab, February 2019. [ncatlab.org](https://ncatlab.org) [Last revised on February 5, 2019 at 13:23:04.].