Category Theory for Programmers

Homework (Session 3)

Bruno Vandekerkhove

Contents

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 $T : C \to B$ 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 $f : c \to c'$ of C an arrow $Tf : Tc \to Tc'$ of B, in such a way that
>
> $$T(1_c) = 1_{Tc}, \qquad T(g \circ f) = Tg \circ Tf,$$
>
> 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 :

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

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

Circles and rectangles

Let's start from a base implementation :

```cpp
#include <iostream>
#include <math.h>

class Shape {
    virtual float area() = 0;
};

class Circle: public Shape {
    float _radius;
public:
    Circle(float radius) { _radius = radius; }
    float area() { return M_PI * _radius * _radius; }
};

class Rectangle: public Shape {
    float _width, _height;
public:
    Rectangle(float width, float height) { _width = width; _height = height; }
    float area() { return _width * _height; }
};
```

We can add an operation to calculate the circumference :

```cpp
...

class Shape {
    virtual float area() = 0;
    virtual float circ() = 0;
};

class Circle: public Shape {
...
    float circ() { return 2 * M_PI * _radius; }
};

class Rectangle: public Shape {
...
    float circ() { return 2 * (_width + _height); }
};
```

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 :

```
...

class Square: public Shape {
    float _width;
public:
    Square(float width) { _width = width; }
    float area() { return _width * _width; }
    float circ() { return 4 * _width; }
};
```

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 :

```haskell
func :: Either a a -> (Bool, a)
func (Left x) = (True, x)
func (Right x) = (False, x)

func_inv :: (Bool, a) -> Either a a
func_inv (True, x) = Left x
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 :

```haskell
fmap _ _ = Nothing
```

Then the identity function is not always preserved :

```haskell
fmap id (Just x) = Nothing = id Nothing /= id (Just x)
```

The laws do hold for the reader functor :

```haskell
fmap id f = (.) id f = id f = f
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 :

```haskell
fmap id Nil = Nil = id Nil
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 :

```
fmap (f . g) Nil = Nil = fmap g Nil = fmap f (fmap g Nil)
fmap (f . g) (Cons x t)
          = Cons ((f . g) x) (fmap (f . g) t)
          = Cons ((f . g) x) (fmap f (fmap g t))
          = fmap f (Cons (g x) (fmap g t))
          = fmap f (fmap g (Cons x t))
```

Implementing the `Reader` functor

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

```
object Reader extends App {
  print(new Reader[String].fmap((i: Int) => 10 + i)(s => s.toInt * 2)("10")) // 30
}

class Reader[R] extends Functor[({type Func[X] = (R => X)})#Func] {
  override def fmap[A, B](f: A => B)(g: R => A): R => B = g andThen f
}

trait Functor[F[_]] {
  def fmap[A, B](f: (A => B))(g: F[A]) : F[B]
}
```

References

[1] Saunders Mac Lane. Categories for the working mathematician, 1978.

[2] nLab. Functor in nlab, February 2019. ncatlab.org [Last revised on February 5, 2019 at 13:23:04.].