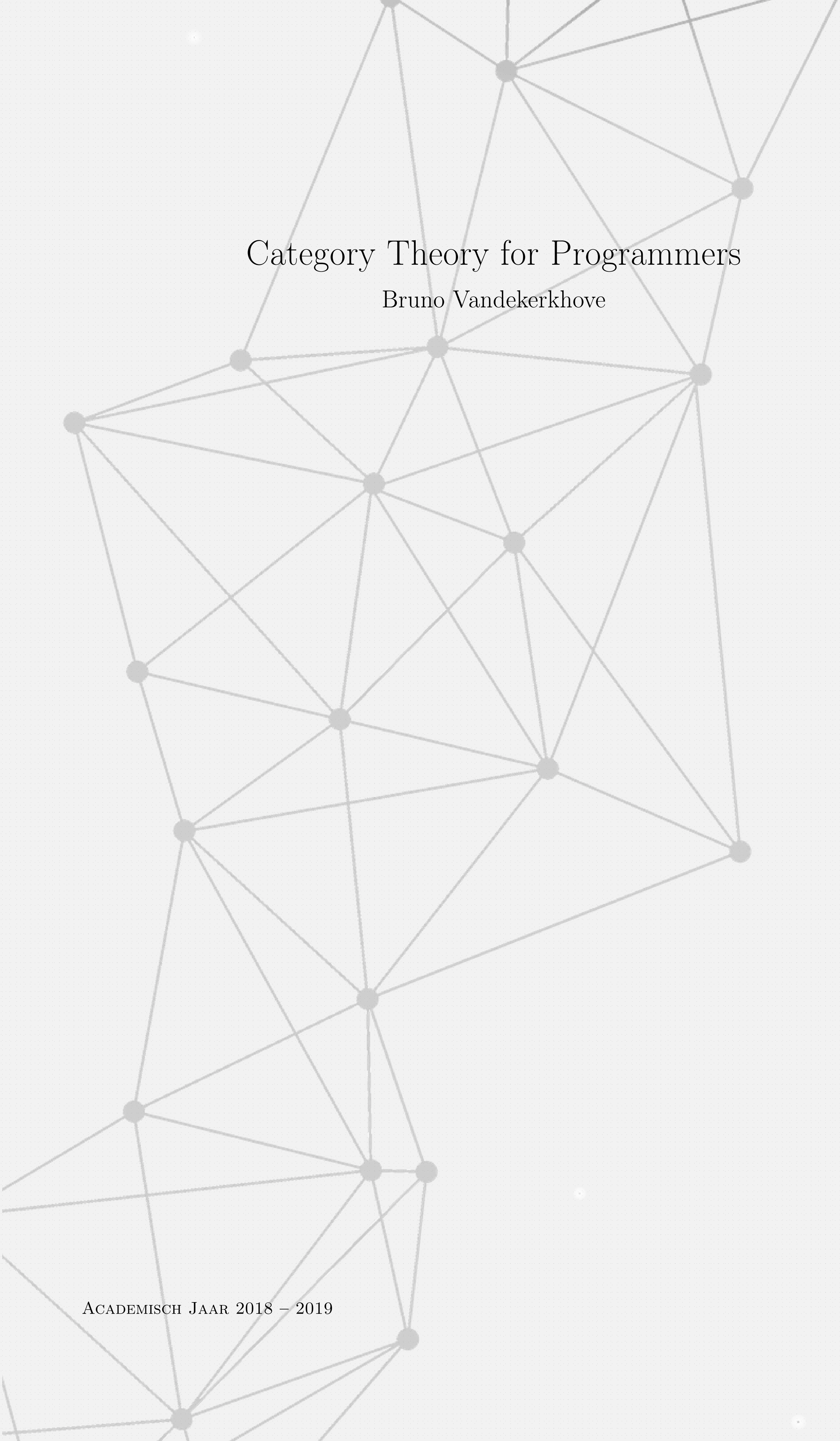

Category Theory for Programmers

Bruno Vandekerkhove

Academisch Jaar 2018 – 2019

Category Theory for Programmers

Homework (Session 1)

Bruno Vandekerkhove

Contents

Definition of a Category

Formal Definition

Milewski summarises his informal definition of a category as follows :

A category consists of objects and arrows (morphisms). Arrows can be composed, and the composition is associative. Every object has an identity arrow that serves as a unit under composition.

Let's compare the statement with the original definition of Mac Lane and his colleague Eilenberg [3, 2]:

A category $C = \{A, \alpha\}$ is an aggregate of abstract elements A called the *objects* of the category, and abstract elements α, called *mappings* of the category. Certain pairs of mappings α_1, α_2 uniquely determine a product mapping $\alpha_1\alpha_2$ subject to the first three axioms listed below. Each object A is associated with a unique mapping e_A for which axioms 4-5 have to hold.

1. The triple product $\alpha_3(\alpha_2\alpha_1)$ is defined if and only if $(\alpha_3\alpha_2)\alpha_1$ is defined. When either is defined, the associative law $\alpha_3(\alpha_2\alpha_1) = (\alpha_3\alpha_2)\alpha_1$ holds. This triple product will be written as $\alpha_3\alpha_2\alpha_1$.

2. The triple product $\alpha_3\alpha_2\alpha_1$ is defined whenever both products $\alpha_3\alpha_2$ and $\alpha_2\alpha_1$ are defined.

   Definition : A mapping $e \in C$ will be called an *identity* of C if and only if the existence of any product $e\alpha$ or βe implies that $e\alpha = \alpha$ and $\beta e = \beta$.

3. For each mapping $a \in C$ there is at least one identity $e_1 \in C$ such that αe_1 is defined, and at least one identity $e_2 \in C$ such that $e_2\alpha$ is defined.

4. The mapping e_A corresponding to each object A is an identity.

5. For each identity e of C there is a unique object A of C such that e_A = e.

Comparison

Aside from the naming, the definition also differs in that it is more abstract. The first of the axioms corresponds to associativity of the composition function. The fact that mappings can be composed if the range of one mapping is equal to the domain of the other is stated by a lemma that follows from the axioms. The fact that each object is associated with just one identity follows from axiom 3-5.

A few decades after the formal definition given here, Mac Lane published a book in which an other definition is given [1]. There he starts by defining a (meta)graph, builds on that to define metacategories (for which he does use the term morphism for denoting the arrows). Finally he introduces the term category as an interpretation of the axioms of a metacategory within set theory. Because a category is defined as a special kind of directed graph, this automatically answers the last challenge of chapter 1. Bartosz mentions the book in his foreword making it likely that he based his informal definition on this one.

In both of Mac Lane's referenced works he points out that since there's a one-to-one correspondence between objects and identities, one could disregard objects altogether and deal with arrows (morphisms or mappings) only. The resulting axioms are equivalent.

Challenges

Memoization

In `C` macros can be used to define memoized functions, but things quickly get complicated, so I limit myself to functions with one integer argument :

```c
#include <stdio.h>

#define MEMOIZE(return_type, function, argument)                  \
    return_type function_internal(int argument);                  \
    return_type function(int argument) {                          \
        static return_type *results[50];                          \
        if (results[argument] != NULL)                            \
            return *results[argument];                            \
        results[argument] = malloc(sizeof(return_type));          \
        *results[argument] = function_internal(argument);         \
        return *results[argument];                                \
    }                                                             \
    return_type function_internal(argument)

int fib(int n) {
    return (n <= 2 ? 1 : fib(n-2) + fib(n-1));
}

MEMOIZE(int, memoized_fib, n) {
    return (n <= 2 ? 1 : memoized_fib(n-2) + memoized_fib(n-1));
}

int main(int argc, const char * argv[]) {
    printf("Hello, World!\n");
    printf("%i\n", fib(45));
    printf("%i\n", memoized_fib(45));
    return 0;
}
```

In languages that provide generics and higher-order functions it's easier to do it (here in Swift, not the preferred language for this) :

```swift
func memoize<I:Hashable,O>(_ function: @escaping ((I) -> O, I) -> O) -> (I) -> O {
    var map = Dictionary<I,O>()
    func internal_function(input: I) -> O {
        print(map)
        if  let value = map[input] {
            return value
        }
        let result = function(internal_function, input)
        map[input] = result
        return result
    }
    return internal_function
}

func fib(_ n: Int) -> Int {
    return (n <= 2 ? 1 : fib(n-1) + fib(n-2))
}

let memoized_fib = memoize {
    memoized_fib, n in
    return (n <= 2 ? 1 : memoized_fib(n-1) + memoized_fib(n-2))
}

print(fib(25)) // 75025, slow
print(memoized_fib(25)) // 75025, fast

import Foundation

func random(_ seeded: Bool) -> Double {
    if  seeded {
        srand48(0)
    }
    return drand48()
}

let memoized_rand = memoize { _, flag in return random(flag) }
memoized_rand(true)
memoized_rand(false) // 0.8404853694114252

print(random(false)) // 0.09637165562356742
print(memoized_rand(false)) // 0.8404853694114252
print(random(true)) // 0.17082803610628972
print(memoized_rand(true)) // 0.17082803610628972
```

Memoizing a random number generator doesn't work because such a generator is a dirty function (its output varies when the input value doesn't). Making use of a seed which is the basis of the random number generation results in a pure function, for which memoization *does* work.

Pure Functions

The factorial function produces no side effects and returns the same output for a given output at all times. `std::getchar` will return a different output depending on the user's behaviour. `std::cout` prints to the screen which is a side effect. Finally, the use of a static variable means that memory shared between function calls is altered (which is a side effect) and the output happens to depend on the value of the very variable, not just on the input.
In other words, only the factorial function is pure.

Void, Unit & Bool

The category with objects $Void$, $Unit$ and $Bool$ could be displayed as follows :

$$\begin{tikzcd}
& \mathbf{Void} \arrow[loop above, "absurd"] \arrow[dl, "absurd"'] \arrow[dr, "absurd"] & \\
\mathbf{Bool} \arrow[loop, in=200, out=250, looseness=4, "id"] \arrow[loop, in=190, out=260, looseness=6, "not"] \arrow[loop, in=180, out=270, looseness=8, "true"] \arrow[loop, in=170, out=280, looseness=10, "false"] \arrow[rr, bend right, "const"'] & & () \arrow[ll, shift right, "true"'] \arrow[ll, "false"] \arrow[loop right, "id"]
\end{tikzcd}$$

Void cannot be returned by any function. It is an initial object so there's only one morphism from **Void** to every object. **Unit** is a terminal object so there's exactly one morphism for every object that points to it.

The Haskell types include the bottom $\perp$ (even **Void**), and the corresponding functions are missing from the diagram. For example, there are 11 functions going from **Bool** to **Bool** (you cannot pattern match the bottom but the `True` and `False` functions may take it as an argument).

The absurd function cannot be called. You can return the bottom with $undefined$.

References

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

[2] Stanford Encyclopedia of Philosophy. Category theory, August 2019. stanford.edu [First published Fri Dec 6, 1996; substantive revision Thu Aug 29, 2019].

[3] Saunders Mac Lane Samuel Eilenberg. General theory of natural equivalences. *Transactions of the American Mathematical Society, Vol. 58, No. 2 (Sep., 1945), pp. 231- 294*, 1945.

Category Theory for Programmers

Homework (Session 2)

Bruno Vandekerkhove

Contents

Challenges

Chapter 3

What Kind of Order?

For a set of sets with an *inclusion* relation it holds that $A \leq B, B \leq A \Rightarrow A = B$. If two sets do not share any elements, then they are not related. Which means we're dealing with a partial order.

In `C++` you can use template functions to define functions that take a pointer to any type. A template function `min` that takes two pointers to scalar types, for example. You could pass that function a pointer to an `int` or a `long`, the compiler won't complain.

```cpp
#include <iostream>

template <class T, class U>
auto min(T *a, U *b) {
    return (*a < *b ? *a : *b);
}

int main(int argc, const char * argv[]) {
    int a = 1;
    long b = 2;
    printf("%i", (int)min(&a, &b));
    return 0;
}
```

This does not work in general ; because of the strong typing system, any function that explicitly asks for a pointer to an `int` will not accept a pointer to a `long` unless you typecast. `void` is one exception. If you consider `void` a type, then any other type is one of its subtypes. Inheritance also makes it possible to substitute one type for another, presuming that the substituted class is a parent class and its substitute one of its subclasses (or subtypes). But in no case can you go the other way around ; you cannot hand a pointer to `void` to a function expecting a pointer to an `int`, for example.

All of this implies that - if we go by *naming equivalence* - then we appear to be dealing with a partial order. Naming equivalence means that two `structs` with the exact same structure are considered different types simply because of having a different name (in contrast with *structural equivalence*).

Addition Modulo 3

Adding 0 corresponds to the identity function, adding 1 or 2 are other possible morphisms, and any composition of these leads to the same set of morphisms (for example, composing the addition of 1 and 2 gives the identity function).

$$\begin{tikzcd} \mathbb{N} \arrow[loop, out=180, in=270, distance=2em, "id"'] \arrow[loop, out=180, in=270, distance=5em, "add1mod3"'] \arrow[loop, out=180, in=270, distance=8em, "add2mod3"'] \end{tikzcd}$$

Chapter 4

It is asked to construct and implement the Kleiski category for partial functions.

Construction

The objects of the category are the types of the programming language.

The (partial) functions take a value of a given class and return an optional. Composition of functions works as follows. The first function is called on the input. If the result is valid (i.e. a valid optional is returned) then the second function is called on the value of the returned optional. Finally, the result of this second function call is returned. If the result of the first function call wasn't valid, an invalid optional is returned. The output class of the first function has to be the same as the input class of the second one.

The identity morphism for an optional of a given class returns the optional itself.

Implementation

Below you can find an implementation in Haskell.

```haskell
data Optional a = Valid a | Invalid deriving Show

(>=>) :: (a -> Optional b) -> (b -> Optional c) -> (a -> Optional c)
(>=>) f g = \x ->
    let y = f x
    in  case y of
        Valid z -> g z
        _ -> Invalid

return :: a -> Optional a
return x = Valid x
```

Composition

In the implementation hereafter, calling `safe_root_reciprocal` will result in an `Invalid` value if the input argument is either negative or equal to zero.

```haskell
safe_root :: Double -> Optional Double
safe_root x = if (x >= 0) then Valid (sqrt(x)) else Invalid

safe_reciprocal :: Double -> Optional Double
safe_reciprocal x = if (x /= 0) then Valid (1/x) else Invalid

safe_root_reciprocal = safe_reciprocal >=> safe_root
```

Chapter 5

Product & Coproduct in Posets

We start from any given poset with a relation denoted with 'less than'. While a preorder is a 'thin' category (there's at most one morphism from any object to an other one), a partial order also imposes the absence of cycles. Let's consider two objects A and B. We want a product C which has projections to A and B (this means it is related to them or 'less than' them). Let's say there is an other object C' which also has such projections.

$$\begin{tikzcd}
 & C' \arrow[d, "m"'] \arrow[ddl, bend right] \arrow[ddr, bend left] & \\
 & C \arrow[dl] \arrow[dr] & \\
A & & B
\end{tikzcd}$$

Then by the definition of a product there must be a unique morphism from C' to C (C' is 'less than' C). This functions as a factorising function m. Meaning that C is a 'better' fit for a product than C'. Or in general, the product of two objects in a poset is the 'largest' object that is smaller than both of these objects. Because the product in categories is unique up to isomorphism, the product in a poset must be unique.

If we were to deal with a total order (in which any two objects including A and B are related), then the product of two objects would be the 'smallest' of the two.

$$\begin{tikzcd}
 & C' \arrow[dl, "m"'] \arrow[drr] & & \\
A \arrow[loop left, "id"] \arrow[rrr] & & & B
\end{tikzcd}$$

The same reasoning can be applied to the coproduct, where the resulting object is the 'smallest' of all objects that is 'larger' than the two given objects. Or the supremum, if you will.

Better Coproducts (`Either` versus `int`)

It's easy to use Milewski's own factoriser function to show that you can factorise the injections to `int` :

```haskell
-- Create factorizer

i :: Int -> Int
i = id

j :: Bool -> Int
j b = if b then 1 else 0

factorizer :: (a -> c) -> (b -> c) -> Either a b -> c
factorizer i j (Left a) = i a
factorizer i j (Right b) = j b

m = factorizer i j

-- GHCI
-- m (Left 1) --> 1
-- m (Right True) --> 1
-- m (Right False) --> 0
```

Because of the existence of such a factoriser it's clear that `Either` is a better coproduct. You cannot do the same the other way round, because 0 and 1 represent both integers and booleans (you can't factorise). I'd argue that since `int` has limited range, adding 2 to positive integers wouldn't solve this problem ; it may lead to an overflow so the injections aren't actually functions.

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 × 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.].

Category Theory for Programmers

Homework (Session 4)

Bruno Vandekerkhove

Contents

Alternative definition of a natural transformation

Saunders MacLane defines natural transformations in terms of what's basically a commutative diagram, the naturality square (here with two functors F and G, $C \to D$) :

$$\begin{tikzcd}
F(c_1) \arrow[r, "F(f)"] \arrow[d, "\alpha_x"'] & F(c_2) \arrow[d, "\alpha_y"] \\
G(c_1) \arrow[r, "G(f)"'] & G(c_2)
\end{tikzcd}$$

The rest of this whole paragraph is loosely equivalent with *nLab's* definition of a natural transformation in terms of the cartesian closed monoidal structure [1]. A cartesian monoidal category C is closed when there's an *internal hom* functor $[X, -] : C \to C$ such that :

$$Hom(a \times b, c) \cong Hom(a, [b, c] = b^c)$$

Where the rightward map is called currying. The exponential notation is used because the category is cartesian.

Since `Cat` is a cartesian monoidal category the question is if there is an *internal hom* functor $[C, -] :$ `Cat` $\to$ `Cat` which would make it closed such that for $A, C, D \in$ `Cat` :

$$Funct(A \times C, D) \cong Funct(A, [C, D])$$

This is true when $[C, D]$ represents the category of functors from C to D. The morphisms in that category are natural transformations, and to see what they are we consider a functor $I \to E$ with E any category and I the *interval* category (a category with just two objects and one morphism between them, denoted as $\{a \to b\}$). This functor corresponds to a choice of morphism in E. So we can find out what a morphism is in $[C, D]$ by taking a close look at the functors $I \to [C, D]$. The following holds :

$$Funct(I, [C, D]) \cong Funct(I \times C, D)$$

This means we can consider functors $I \times C \to D$ instead. The morphisms of the category $I \times C$ are pairs of morphisms in I and C subject to composition laws such that we end up with a commuting diagram :

$$\begin{tikzcd}
(c_1, a) \arrow[r, "{(f, id)}"] \arrow[d, swap, "{(id, a \to b)}"] & (c_2, a) \arrow[d, "{(id, a \to b)}"] \\
(c_1, b) \arrow[r, swap, "{(f, id)}"] & (c_2, b)
\end{tikzcd}$$

We can therefor conclude that a natural transformation between functors $C \to D$ is given by the image of this square in D. For a natural transformation $\alpha : F \to G$ one can trace back the *hom*-isomorphism to end up with a commuting diagram which corresponds to the naturality square depicted at the start of the paragraph (also depicted in Bartosz's book).

The bifunctor `Pair a b`

Here's an implementation of the `Pair` bifunctor :

```haskell
import Data.Bifunctor

data Pair a b = Pair a b

instance Bifunctor Pair where
    bimap g h (Pair x y) = Pair (g x) (h y)
    first g (Pair x y) = Pair (g x) y
    second h (Pair x y) = Pair x (h y)
```

`Pair a b` can be proven to be a bifunctor with equational reasoning (it's enough to show that it is functorial in each argument separately) :

```haskell
bimap id id (Pair x y) = Pair (id x) (id y)
        = Pair x y
        = id (Pair x y)
bimap (f . g) id (Pair x y) = Pair (f . g x) (id y)
        = Pair (f . g x) y
        = bimap g . (Pair (f x) y)
        = bimap g . bimap f (Pair x y)
-- bimap id (f . g) (Pair x y) boils down to the same thing
```

This is not a surprise given that `Pair` is isomorphic with the `(,)` bifunctor. We now show that the above implementation is equivalent to using the default ones :

```haskell
bimap g h (Pair x y) = first g . second h = first g . (Pair x (h y)) = Pair (g x) (h y)
first g = bimap g id = Pair (g x) (id y) = Pair (g x) y
second h = bimap id h = Pair (id x) (h y) = Pair x (h y)
```

Isomorphism of `Maybe` and `Maybe'`

It can be shown that `Maybe'` and `Maybe` are isomorphic by constructing two mappings and using equality reasoning to show that they're each other's inverse :

```haskell
import Data.Functor.Identity
import Data.Functor.Const

type Maybe' a = Either (Const () a) (Identity a)

toMaybe' :: Maybe a -> Maybe' a
toMaybe' Nothing = Left (Const ())
toMaybe' (Just a) = Right (Identity a)

toMaybe :: Maybe' a -> Maybe a
toMaybe (Left (Const ())) = Nothing
toMaybe (Right (Identity a)) = Just a

toMaybe' . toMaybe (Left (Const ())) = toMaybe' Nothing = Left (Const ())
toMaybe' . toMaybe (Right (Identity a)) = toMaybe' (Just a) = Right (Identity a)

toMaybe . toMaybe' Nothing = toMaybe (Left (Const ())) = Nothing
toMaybe . toMaybe' (Just a) = toMaybe (Right (Identity a)) = Just a
```

The `PreList` bifunctor

The `PreList` bifunctor can be implemented as follows :

```haskell
data PreList a b = Nil | Cons a b

instance BiFunctor (PreList a b) where
        bimap g h Nil = Nil
        bimap g h (Cons a b) = Cons (g a) (h b)
```

Aside from proving that it is functorial in both arguments (which is analogous to what we did for `Pair`) it can be noted that it is equivalent to `Either (Const () a) (a,b)` which is a bifunctor because `Either` and `(,)` are bifunctors and `Const ()` is a functor. The reasoning is similar to what happens with `Maybe = Either (Const () a) (Identity a)` except that in that case you really end up with a functor because the type parameter is the same for both parts of the sum.

`K2`, `Fst` and `Snd`

Implementation of `bimap` follows :

```haskell
instance BiFunctor (K2 c a b) where
        bimap _ _ = id
instance BiFunctor (Fst a b) where
        bimap g _ (Fst a) = Fst (g a)
instance BiFunctor (Snd a b) where
        bimap _ h (Snd b) = Fst (h b)
```

In the referenced study `K2` is described as the bifunctorial 'cousin' of `K1` which is a constant. They're part of a repertoire of components called *'polynomial (bi)functors'* which are used by the author to deal with data types in a generic way.

Natural transformation of `Maybe` to `List`

Here's a natural transformation from `Maybe` to `List` and a proof that it satisfies the naturality condition :

```haskell
alpha :: Maybe a -> [a]
alpha Nothing = []
alpha (Just x) = [x]

fmap f (alpha Nothing) = fmap f [] = []
alpha (fmap f Nothing) = alpha Nothing = []

fmap f (alpha (Just x)) = fmap f [x] = [f x]
alpha (fmap f (Just x)) = alpha (Just (f x)) = [f x]
```

Many more are possible, which is illustrated in the next challenge. It can be noted that Haskell's parametric polymorphism made the proof unnecessary, which was hinted at in the previous lesson where the 'magic trick' showed that polymorphic functions do nothing to the elements themselves so the order of application isn't important.

Natural transformation from `Reader` to `List` or `Maybe`

There's an infinite amount of natural transformations from `Reader ()` to `List` because you can transform to the empty list or to a series of applications of g to () :

```haskell
-- :: Reader () a -> [a]
alpha0 (Reader g) = []
alpha1 (Reader g) = [g ()]
alpha2 (Reader g) = [g (), g ()]
...
```

This doesn't hold in the case of `Reader Bool` and `Maybe` where you can only transform in three ways (returning `Nothing`, `Just (g True)` and `Just (g False)` respectively).

References

[1] nLab. natural transformation in nlab, April 2019. ncatlab.org [Last revised on April 3, 2019 at 03:15:34.].

Category Theory for Programmers

Homework (Session 5)

Bruno Vandekerkhove

Contents

Pushout in `C++`

We talked about products and co-products in posets before, where the term supremum and infimum applied. In this more general case applied to `C++` types a diagram of a pushout D is as follows :

$$\begin{tikzcd}
 & E & \\
 & D & \\
B \arrow[ur] \arrow[uur] & A \arrow[l] \arrow[r] & C \arrow[ul] \arrow[uul]
\end{tikzcd}$$

Any superclass E of B and C is a superclass of D. Which means that D corresponds to a superclass that encompasses all the functionality that the subclasses have in common[1].

Limit of the identity functor

The identity functor `Id` maps a category to itself. The initial object is the object for which there's exactly one morphism to every object. When constructing the limit for `Id` any apex will correspond to an initial object (or there wouldn't be a natural transformation from Δ_c to `Id`). If there are many apexes (or initial objects) there will be inversible transformations between them. Indeed, initial objects are unique up to isomorphism, which we knew already.

[1]One has to consider all possible superclasses of B and C. Then D inherits from all of those. Some superclasses may only declare a part of the functionality that's present in both the classes B and C, yet together they declare all of it and nothing more or some wouldn't be superclasses.

Pullback, pushout, ... in `Set`

Starting from the diagram depicted on the previous page and applying analogous reasoning we can see that a pushout D is the smallest superset or the union of B and C. In the opposite category we get :

$$\begin{tikzcd}
 & E \arrow[ddl] \arrow[ddr] & \\
 & D \arrow[dl] \arrow[dr] & \\
B \arrow[r] & A & C \arrow[l]
\end{tikzcd}$$

Here, the pullback D is the largest subset or the intersection (every other set that's a subset of B and C ought to be a subset of D). A trellis visualises this and the terms supremum and infimum are in order.

As for the initial - and terminal object, they (trivially) correspond to the empty set and the set on which the category is based.

Co-equalizer

Visually :

$$\begin{tikzcd}
 & C & \\
B \arrow[ur, "p"] & & A \arrow[ul, "q"'] \arrow[ll, shift right, "f"'] \arrow[ll, shift left, "g"]
\end{tikzcd}$$

Once again q is fully defined by p and one of the morphisms from $A \to B$. We have

$$q = p \cdot f \qquad \textit{and} \qquad q = p \cdot g \qquad \Rightarrow \qquad p \cdot f = p \cdot g$$

where for every other co-equalizer C' there should be a unique factoriser. Aside from this universal construction, there's a more concrete explanation of what it means in `Set`. There we end up with a partitioning of B that corresponds to an equivalence relationship $f(x) \sim g(x)$. It should be the smallest of such relationships which corresponds to the finest partitioning[2].

Pullback towards terminal object, pushout from initial object

A pullback is a special kind of product because A imposes some additional structure. When you're dealing with A as a terminal object and you're looking for a product, then that structure (morphisms to A from B, C and D) will necessarily be there. In other words, every such product is a pullback towards the terminal object (and vice versa, every such pullback is a product).

In the same vein, when you're dealing with A as an initial object and are looking for a coproduct then there will be unique morphisms from A to B, C and D which necessarily make it a pushout from the initial object.

[2]An example ; let $A = B = \{1, 2, 3\}$, $f = id$, $g(1) = 2$, $g(2) = 1$ and $g(3) = 3$. Then we find a co-equalizer p where $p(1) = p(2) = [1]$ and $p(3) = [3]$. This is a '*better*' co-equalizer than p' with $p'(x) = [1]$ because we find the factoriser h with $h(x) = [1]$ for which $p' = h \cdot p$.