

My solutions to Bartosz Milewski's "Category Theory for Programmers"

Posted on November 10, 2018

I recently worked through Bartosz Milewski's excellent free book "Category Theory for Programmers." The book is available online here (<https://github.com/hmemcpy/milewski-ctfp-pdf/>) and here (<https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/>).

I had an awesome time reading the book and learning about Category Theory so I figured I'd post my solutions to the book problems online to make it easier for other people to have a similar experience. You can find my solutions below:

Section 1

p1.1

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

```
def identity(x):  
    return x
```

p1.2

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

Solution

```
def compose(f1, f2):  
    return lambda x: f2(f1(x))
```

p1.3

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

Solution

```
assert compose(lambda x: x + 4, identity)(5) == 9
assert compose(identity, lambda x: x + 4)(5) == 9
```

p1.4

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

Solution The world wide web is indeed a category if we consider the objects to be webpages and for there to be an “arrow” between A and B if there is a way to get to B from A by clicking on links

p1.5

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

Solution No, because just because $A \rightarrow B$ and $B \rightarrow C$ does not imply $A \rightarrow C$

p1.6

When is a directed graph a category?

Solution Whenever every node has an edge that points back to it and for every two nodes A, B such that there is a path from A to B , there is also an edge connecting A to B .

Section 2

p2.1 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

```
def memoize(f):
    calls = {}
    def memoized(x):
        if x not in calls:
            calls[x] = f(x)
        return calls[x]
    return memoized
```

p2.2 Try to memoize a function from your standard library that you normally use to produce random numbers. Does it work?

Solution This will not work

p2.3 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

```
def seed_to_random(seed):
    np.random.seed(seed)
    return np.random.random()
memoized_random = memoize(seed_to_random)

assert np.isclose(memoized_random(0), memoized_random(0))
assert memoized_random(0) != memoized_random(1)
```

p2.3 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.

Solution factorial is a pure function

b: std::getchar()

Solution getchar is not a pure function, since it relies on the state of stdin

c:

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

Solution `f` is not a pure function, since it has the side effect of printing

d:

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

Solution `f` is not a pure function, since it both has the side effect of incrementing `y` and relies on the state of static variable `y`

p2.5 How many different functions are there from `Bool` to `Bool` ? Can you implement them all?

Solution

```
same :: Bool -> Bool  
same trueorfalse = trueorfalse  
  
opposite :: Bool -> Bool  
opposite trueorfalse = not trueorfalse  
  
alwaystrue :: Bool -> Bool  
alwaystrue _ = True  
  
alwaysfalse :: Bool -> Bool  
alwaysfalse _ = False
```

Section 3

p3.1 Generate a free category from:

- **A graph with one node and no edges** *Solution* Add an identity arrow.
- **A graph with one node and one (directed) edge (hint: this edge can be composed with itself)** *Solution* Add infinite arrows to represent every number of applications of the directed edge.

- **A graph with two nodes and a single arrow between them** *Solution* Add identity arrows.
- **A graph with a single node and 26 arrows marked with the letters of the alphabet: a, b, c ... z.** *Solution* Add an identity arrow, and then add infinite arrows, one for every combination of a-z of any length.

p3.2 What kind of order is this?

- **A set of sets with the inclusion relation: A is included in B if every element of A is also an element of B .** *Solution* This is a partial order.
 - For any (a, b) there is at most one $a \rightarrow b$ and if $a \rightarrow b$ and $b \rightarrow a$ then a and b have the same elements and are the same set.
 - Since there might be some (a, b) where $a \cap b$ is empty, this is not a total order
- **C++ types with the following subtyping relation: τ_1 is a subtype of τ_2 if a pointer to τ_1 can be passed to a function that expects a pointer to τ_2 without triggering a compilation error.** *Solution* This is a partial order.
 - For any (τ_1, τ_2) there is at most one $\tau_1 \rightarrow \tau_2$, and if $\tau_1 \rightarrow \tau_2$ and $\tau_2 \rightarrow \tau_1$ then τ_1 and τ_2 are the same type.
 - There are types not connected by a subtype relation, so this is not a total order.

p3.3 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 **AND * Closure:** The output of `AND` is boolean * **Identity:** The identity is `True` * **Associative:** Easy to show by enumeration

OR * Closure: The output of `OR` is boolean * **Identity:** The identity is `False` * **Associative:** Easy to show by enumeration

p3.4 Represent the `Bool` monoid with the `AND` operator as a category: List the morphisms and their rules of composition.

Solution The single element in this category is the `Bool` type. The morphisms are `AND True` (identity) and `AND False`. The composition of these two is `AND False`.

p3.5 Represent addition modulo 3 as a monoid category.

Solution The single element in this category is the type `[Int < 3, >= 0]`. The morphisms are `* A: add 3n (identity) * B: add 1 + 3n * C: add 2 + 3n` The morphisms in this category are closed under association because `B . B` is `C` and both `B . C`, `C . B` are `A`

Section 4

p4.1 Construct the Kleisli category for partial functions (define composition and identity).

Solution

```
class Optional:

    def __init__(self, value):
        self._value = value

    def is_valid(self):
        return self._value is not None

    def get(self):
        assert self.is_valid()
        return self._value

def compose(f1, f2):
    def composed(x):
        flout = f1(x)
        return f2(flout.get()) if flout.is_valid() else Optional(None)
    return composed

def identity(x):
    return Optional(x)
```

p4.2 Implement the embellished function `safe_reciprocal` that returns a valid reciprocal of its argument, if it's different from zero.

Solution

```
def safe_root(x):
    return Optional(np.sqrt(x)) if x >= 0 else Optional(None)

def safe_reciprocal(x):
    return Optional(1 / float(x)) if x != 0 else Optional(None)

assert not safe_root(-1).is_valid()
assert np.isclose(safe_root(4).get(), 2.0)

assert not safe_reciprocal(0).is_valid()
assert np.isclose(safe_reciprocal(4).get(), 0.25)
```

p4.3 Compose `safe_root` and `safe_reciprocal` to implement `safe_root_reciprocal` that calculates $\sqrt{1/x}$ whenever possible.

Solution

```
safe_root_reciprocal = compose(safe_reciprocal, safe_root)

assert not safe_root_reciprocal(0).is_valid()
assert not safe_root_reciprocal(-5).is_valid()
assert np.isclose(safe_root_reciprocal(0.25).get(), 2)
```

Section 5

p5.1 Show that the terminal object is unique up to unique isomorphism.

Solution Consider two terminal objects A , B . There is exactly one morphism m_1 from $A \rightarrow B$ since B is terminal and exactly one morphism m_2 from $B \rightarrow A$ since A is terminal. Then $m_1 \circ m_2$ is a morphism from $B \rightarrow B$ and $m_2 \circ m_1$ is a morphism from $A \rightarrow A$. Since B is terminal, there is exactly one morphism from $B \rightarrow B$, so $m_1 \circ m_2$ is the identity.

Therefore m_1 , m_2 form an isomorphism between A and B . Since there are no other morphisms between A and B , m_1 , m_2 is a unique isomorphism.

p5.2 What is a product of two objects in a poset? Hint: Use the universal construction.

Solution The product of two objects A , B in a poset is the object C that is less

than both A and B (i.e. exists: $p: C \rightarrow A$ and $q: C \rightarrow B$) and for any other object D that is also less than A and B , exists $D \rightarrow C$. This object does not always exist.

Sufficiency Say we have such an A, B, C . Now consider some object D such that $p_2: D \rightarrow A$, $q_2: D \rightarrow B$. Then we have some $m: D \rightarrow C$ so $p_1 \circ m: D \rightarrow A$ and $q_1 \circ m: D \rightarrow B$. Now since there is at most one morphism between any pair of objects in a poset, it's true that $p_2 = p_1 \circ m$ and $q_2 = q_1 \circ m$, so m factorizes p and q .

Necessity If there were some object D such that $D \rightarrow A$, $D \rightarrow B$ but not $D \rightarrow C$, then there is no morphism m such that $p_1 = p_2 \circ m$ since m must be a morphism from $D \rightarrow C$. Therefore C is not the product of A, B .

p5.3 What is a coproduct of two objects in a poset?

Solution We just reverse the arrows in **p5.2**. The coproduct of two objects A, B in a poset is the object C that is greater than both A and B (i.e. exists: $p: A \rightarrow C$ and $q: B \rightarrow C$) and for any other object D that is also greater than A and B , exists $C \rightarrow D$. This object does not always exist.

p5.4 Implement the equivalent of Haskell `Either` as a generic type in your favorite language (other than Haskell).

Solution


```

class EitherAbstract(object):
    pass

class RightEither(EitherAbstract):
    def __init__(self, right):
        self.right = right

class LeftEither(EitherAbstract):
    def __init__(self, left):
        self.left = left

# doing something like this in python is a recipe for disaster :)
def either_factory(left_type, right_type):
    def generate(left=None, right=None):
        assert ((left is None) ^ (right is None))
        if left is not None:
            assert isinstance(left, left_type)
            out = LeftEither(left=left)
        if right is not None:
            assert isinstance(right, right_type)
            out = RightEither(right=right)
        return out
    return generate

my_left = either_factory(int, str)(left=5)
assert my_left.left == 5

my_right = either_factory(int, str)(right="5")
assert(my_right.right == "5")

try:
    either_factory(int, str)()
    print("failed")
except AssertionError:
    pass

try:
    either_factory(int, str)(left=5, right="5")
    print("failed")
except AssertionError:

```

```
pass
```

p5.5 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

```
# Consider the following injections from int and bool into int
def int_to_int(int_param):
    assert isinstance(int_param, int)
    return int_param

def bool_to_int(bool_param):
    assert isinstance(bool_param, bool)
    return 1 if bool_param else 0

# We can define the following morphism from Either into int
def either_to_int(either_param):
    isinstance(either_param, EitherAbstract)
    if either_param.kind == "left":
        out = int_to_int(either_param.left)
    elif either_param.kind == "right":
        out = bool_to_int(either_param.right)
    return out

"""
Then
bool_to_int(x) = either_to_int (either_factory(int, bool)(left=x))
int_to_int(y) = either_to_int (either_factory(int, bool)(right=y))
so either_to_int factorizes bool_to_int and int_to_int
"""

assert either_to_int(either_factory(int, bool)(left=5)) == int_to_int(5)
assert either_to_int(either_factory(int, bool)(right=True)) == bool_to_int(True)
assert either_to_int(either_factory(int, bool)(right=False)) == bool_to_int(False)
```

p5.6 Continuing the previous problem: How would you argue that `int` with the two injections `i` and `j` cannot be “better” than `Either`?

Solution Say there exists some function `impossible_m` such that

```
either_factory(int, bool)(left=x) = impossible_m ( int_to_int (x))
either_factory(int, bool)(right=y) = impossible_m ( bool_to_int (y))
```

For all `int x` and `bool y`. Then it must be the case that:

```
impossible_m(1) = LeftEither(left=1)
impossible_m(1) = RightEither(right=1)
```

Which is not possible, because the output of a function for a given input argument must be unique.

p5.7 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 Say there exists some function `new_m` such that

```
def int_to_int_2(int_param):
    assert isinstance(int_param, int)
    return int_param if int_param < 0 else int_param + 2

either_factory(int, bool)(left=x) = new_m ( int_to_int_2 (x))
```

Then it must be the case that

```
new_m(-x) = LeftEither(-x) for all x
new_m(2) = LeftEither(0)
new_m(3) = LeftEither(1)
new_m(4) = LeftEither(2)
...
new_m(max_int) = Left(max_int - 2)
```

Since there are only a finite number of integers in python/C++, `Left(max_int - 1)` and `Left(max_int)` cannot be in the domain of `new_m`.

p5.8 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

Consider some type `SuperEither` defined as

```
data SuperEither = IntBoolTuple (Int, Int) | BoolBoolTuple (Bool, Bool)
```

We can define injections into this type from `Int` and `Bool` of the forms

```
intInt :: Int -> SuperEither;
intInt x = IntIntTuple (x, x)

boolBool :: Bool -> SuperEither;
boolBool x = BoolBoolTuple (x, x)
```

Then we can define multiple morphisms from `SuperEither` into `Either` that take either the first or second element.

Section 6

p6.1 Show the isomorphism between `Maybe a` **and** `Either () a`.

Solution We can define the following two functions, which serve as inverses

```
maybeToEither :: Maybe a -> Either () a
maybeToEither inputMaybe =
  case inputMaybe of
    Just a -> Right a
    Nothing -> Left ()

eitherToMaybe :: Either () a -> Maybe a
eitherToMaybe inputEither =
  case inputEither of
    Right a -> Just a
    Left () -> Nothing
```

p6.2 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.

Solution

```

# I'll use python again, just for fun

class AbstractShape(object):

    def area(self):
        assert NotImplementedError()

    def circ(self):
        assert NotImplementedError()

class Circle(AbstractShape):

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return self.radius**2 * np.pi

    def circ(self):
        return 2 * self.radius * np.pi

class Rect(AbstractShape):

    def __init__(self, height, width):
        self.height = height
        self.width = width

    def area(self):
        return self.height * self.width

    def circ(self):
        return 2 * self.height + 2 * self.width

rect = Rect(3, 5)
assert rect.circ() == 16
assert rect.area() == 15

```

p6.3 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 C++ or Java implementation. What parts of the original code did you have to touch?

Solution See above. We needed to add it to each class, including `AbstractShape`.

p6.4 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. C++ or Java? (Even if you're not a Haskell programmer, the modifications should be pretty obvious.)

Solution For Haskell we need to update the `Shape` definition and add another line to `circ` and `area` implementations. For Python we needed to write a new class with a new initializer, inheriting from `Rect`

Haskell:

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

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

circ :: Shape -> Float
circ (Circle r) = 2.0 * pi * r
circ (Rect d h) = 2.0 * (d + h)
circ (Square h) = 4.0 * h
```

Python:

```

class Square(Rect):

    def __init__(self, length):
        self.height = length
        self.width = length

square = Square(5)
assert square.circ() == 20
assert square.area() == 25

```

p6.5 Show that $a + a = 2 * a$ holds for types (up to isomorphism). Remember that 2 corresponds to `Bool`, according to our translation table.

Solution $a + a$ is equivalent to `Either a a` and $2 * a$ is equivalent to `(Bool, a)`. We can define the following invertible functions between them.

```

aPlusAToTwoTimesA :: Either a a -> (Bool, a)
aPlusAToTwoTimesA eitherAA =
    case eitherAA of
        Left a -> (True, a)
        Right a -> (False, a)

twoTimesAToAPlusA :: (Bool, a) -> Either a a
twoTimesAToAPlusA twoTimesA =
    case twoTimesA of
        (True, a) -> Left a
        (False, a) -> Right a

```

Section 7: Functors

p7.1 Can we turn the `Maybe` type constructor into a functor by defining: `fmap _ _ = Nothing` which ignores both of its arguments? (Hint: Check the functor laws.)

Solution No, this mapping of morphisms does not preserve the identity. For some `Just a`, we see that:

```

(fmap id) Just a = Nothing
id Just a = Just a

```

p7.2 Prove functor laws for the `reader` functor. Hint: it's really simple.

Solution We need to use equational reasoning to prove that `fmap` maintains

identity and preserves composition

Identity

```
fmap id (a->b) =  
    (.) id (a->b) =  
    id (a->b)
```

Composition

```
fmap ((c->d) . (b->c)) (a->b) =  
    (c->d) . (b->c) . (a->b) =  
    (c->d) . fmap ((b->c) (a->b)) =  
    fmap (c->d) (fmap (b->c) (a->b))
```

p7.3 Implement the `reader` functor in your second favorite language (the first being Haskell, of course).

Solution

```
def reader_functor_fmap(f, r_to_a):  
    return lambda r: f(r_to_a(r))  
  
def r_to_0(r):  
    return 0  
  
def r_to_1(r):  
    return 1  
  
r_to_5 = reader_functor_fmap(lambda x: x + 5, r_to_0)  
  
assert r_to_0("r") == 0  
assert r_to_1("r") == 1  
assert r_to_5("r") == 5
```

p7.4 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 (in other words, use induction).

Solution

Base Case

- **Identity**

```
fmap id Nil = Nil = id Nil
```

- **Composition**

```
fmap (f . g) Nil = Nil = fmap f (Nil) = fmap f (fmap g Nil)
```

Inductive Step

- **Identity**

```
fmap id (Cons x t) =
  Cons (id x) (fmap id t) =
  Cons (id x) (id t) =
  Cons (x t) =
  id Cons (x t)
```

- **Composition**

```
fmap (f . g) (Cons x t)
= Cons ((f . g) x) (fmap (f . g) t) // definition of fmap
= Cons ((f . g) x) ((fmap f . fmap g) t) // induction
= fmap f (Cons (g x) (fmap g t)) // definition of fmap
= fmap f (fmap g (Cons (x t))) // definition of fmap
= (fmap f . fmap g) (Cons (x t))
```

Section 8: Functoriality

p8.1 Show that the data type: `data Pair a b = Pair a b` is a bifunctor. For additional credit implement all three methods of `Bifunctor` and use equational reasoning to show that these definitions are compatible with the default implementations whenever they can be applied.

Solution Say we keep one of the arguments constant, then the `fmap` for both sides is just:

```
fmap f Pair (a), (C) = Pair (f a) (C)
```

Identity

```
fmap id Pair a C = Pair id a C = Pair a C
```

Composition

```
fmap f*g Pair a C = Pair f*g(a) C = fmap f Pair g(a) C = fmap f fmap
```

The three methods of Bifunctor

```
data Pair a b = Pair a b
pairBimap :: (a -> c) -> (b -> d) -> Pair a b -> Pair c d
pairBimap g h (Pair a b) = Pair (g a) (h b)

pairFirst :: (a -> c) -> Pair a b -> Pair c b
pairFirst g (Pair a b) = Pair (g a) b

pairSecond :: (b -> d) -> Pair a b -> Pair a d
pairSecond f (Pair a b) = Pair a (f b)
```

Proof that these definitions are compatible with the default implementations whenever they can be applied.

Proof of `pairBimap`

```
(pairBimap g h) (Pair a b) =
  Pair (g a) (h b) // definition of pairBimap
  pairFirst g (Pair a (h b)) // definition of pairFirst
  (pairFirst g . pairSecond h) (Pair a b) // definition of pairSecond
```

Which means that

```
pairBimap g h = pairFirst g . pairSecond h
```

Proof of `pairFirst`

```
pairFirst g (Pair a b) =
  Pair (g a) b // definition of pairFirst
  Pair (g a) (id b) // definition of id
  pairBimap (g id) Pair a b // definition of pairBimap
```

Which means that

```
pairFirst g = pairBimap g id
```

Proof of `pairSecond`

```
pairSecond f (Pair a b) =  
  Pair a (f b) // definition of pairSecond  
  Pair (id a) (f b) // definition of id  
  pairBimap (id f) Pair a b // definition of pairBimap
```

Which means that

```
pairSecond = pairBimap id
```

p8.2 Show the isomorphism between the standard definition of `Maybe` and this desugaring:
`type Maybe' a = Either (Const () a) (Identity a)` **Hint: Define two mappings between the two implementations. For additional credit, show that they are the inverse of each other using equational reasoning.**

Solution

```
data MyIdentity a = MyIdentity a  
data MyConst c a = MyConst c  
type Maybe' a = Either (MyConst () a) (MyIdentity a)  
  
desugaredToMaybe :: Maybe' a -> Maybe a  
desugaredToMaybe (Left (MyConst ())) = Nothing  
desugaredToMaybe (Right (MyIdentity a)) = Just a  
  
maybeToDesugared :: Maybe a -> Maybe' a  
maybeToDesugared Nothing = Left (MyConst ())  
maybeToDesugared (Just a) = Right (MyIdentity a)
```

We show that they are the inverse of each other using equational reasoning

```

maybeToDesugared desugaredToMaybe (Left (MyConst ()))
  = maybeToDesugared Nothing
  = maybeToDesugared Left (MyConst ())

desugaredToMaybe maybeToDesugared Nothing
  = desugaredToMaybe (Left (MyConst ()))
  = Nothing

maybeToDesugared desugaredToMaybe (Right (MyIdentity a))
  = maybeToDesugared Just a
  = Right (MyIdentity a)

desugaredToMaybe maybeToDesugared Just a
  = desugaredToMaybe Right (MyIdentity a)
  = Just a

```

p8.3 Let's try another data structure. I call it a `PreList` because it's a precursor to a `List`. It replaces recursion with a type parameter `b`: `data PreList a b = Nil | Cons a b`. You could recover our earlier definition of a `List` by recursively applying `PreList` to itself (we'll see how it's done when we talk about fixed points). Show that `PreList` is an instance of `Bifunctor`.

Solution Lets form the following mapping

```

fmapFull :: (a -> c) -> (b -> d) -> (PreList a b) -> (PreList c d)
fmapFull f g Nil = Nil
fmapFull f g Cons a b = Cons (f a) (g b)

```

Say we keep `b` constant (WLOG). Then the `fmap` for `a` is

```

fmap f Nil = Nil
fmap f Cons a C = Cons (f a) C

```

This is a functor because **Identity**

```

fmap id Nil = Nil
fmap id Cons a C = Cons id a C = Cons a C

```

Composition

```
fmap f . g Nil = Nil
fmap f . g Cons a C =
  Cons f . g a C =
  fmap f Cons g a C =
  fmap f (fmap g Cons a C)
```

p8.4 Show that the following data types define bifunctors in `a` and `b`:

```
data K2 c a b = K2 c
data Fst a b = Fst a
data Snd a b = Snd b
```

For additional credit, check your solutions against Conor McBride's paper [Clowns to the Left of me, Jokers to the Right](#).

Solution

`K2` : Without loss of generality, if we hold `b` constant, then `K2` becomes `Const`, which is a functor

`Fst` : If we hold `a` constant, then `Fst` becomes `Const`, which is a functor If we hold `b` constant, then `Fst` becomes `Identity`, which is a functor

`Snd` : If we hold `a` constant, then `Snd` becomes `Identity`, which is a functor If we hold `b` constant, then `Snd` becomes `Const`, which is a functor

p8.5 Define a bifunctor in a language other than Haskell. Implement `bimap` for a generic pair in that language.

Solution

```

class Bifunctor(object):

    def apply_bimap(self, f, g):
        assert False

    @classmethod
    def first(cls, f):
        return lambda pair: pair.apply_bimap(f, lambda x: x)

    @classmethod
    def second(cls, g):
        return lambda pair: pair.apply_bimap(lambda x: x, g)

    @classmethod
    def bimap(cls, f, g):
        return lambda pair: pair.apply_bimap(f, g)

class Pair(Bifunctor):

    def __init__(self, aval, bval):
        self.aval = aval
        self.bval = bval

    def apply_bimap(self, f, g):
        return Pair(f(self.aval), g(self.bval))

p = Pair(5, "4")
first_mapped = Bifunctor.first(lambda x: x + 1)(p)
assert first_mapped.aval == 6
assert first_mapped.bval == "4"

second_mapped = Bifunctor.second(lambda x: x + "1")(p)
assert second_mapped.aval == 5
assert second_mapped.bval == "41"

bimapped = Bifunctor.bimap(lambda x: x + 1, lambda s: s + "1")(p)
assert bimapped.aval

```

p8.6 Should `std::map` be considered a bifunctor or a profunctor in the two template arguments `key` and `T`? How would you redesign this data type to make it so?

Solution `std::map` should be considered a profunctor in `Key` and `T`.

We can define it as a Profunctor as follows:

```
get :: a -> Maybe b

instance Profunctor get where
  dimap f g get = lmap f . rmap g
  lmap f get = \x -> get (f x)
  rmap g get = \x -> fmap g (get x)
```

Section 9: Function Types (No Challenges)

Section 10: Natural Transformations

p10.1 Define a natural transformation from the `Maybe` functor to the `list` functor. Prove the naturality condition for it.

Solution

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

The naturality condition is $G f \circ \alpha_a = \alpha_b \circ F f$, which translates to

`fmap_list f . natTrans = natTrans . fmap_maybe f`

Nothing Case:

```
fmap_list f . natTrans Nothing =
  fmap_list f [] =
  [] =
  natTrans Nothing =
  natTrans . fmap_maybe f Nothing
```

(Just x) Case:


```
fmap_list f . natTrans (Just x) =
  fmap_list f [x] =
  [f(x)] =
  natTrans (Just (f x)) =
  natTrans . fmap_maybe f (Just x)
```

p10.2 Define at least two different natural transformations between `Reader ()` and the `list` functor. How many different lists of `()` are there?

Solution

```
natTransRL1:: (() -> a) -> [a]
natTransRL1 _ = []

natTransRL2:: (() -> a) -> [a]
natTransRL2 g = [g ()]

natTransRL3:: (() -> a) -> [a]
natTransRL3 g = fmap g [(), ()]
```

Since there are an infinite number of lists of `[(), ...]`, there are an infinite number of these natural transformations.

p10.3 Continue the previous exercise with `Reader Bool` and `Maybe`.

Solution

There are three natural transformations from `Reader Bool -> Maybe`

```
natTransRB1:: (Bool -> a) -> Maybe a
natTransRB1 _ = Nothing

natTransRB2:: (Bool -> a) -> Maybe a
natTransRB2 g = Just (g True)

natTransRB3:: (Bool -> a) -> Maybe a
natTransRB3 g = Just (g False)
```

p10.4 Show that horizontal composition of natural transformation satisfies the naturality condition (hint: use components). It's a good exercise in diagram chasing.

Solution

We have the functors F , G and the natural transformations:

```
 $\alpha a :: F\ a \rightarrow F'\ a$   
 $\beta a :: G\ a \rightarrow G'\ a$ 
```

We need to show that $(G' \circ F') \circ f \circ (\beta \circ \alpha) a = (\beta \circ \alpha) b \circ (G \circ F) \circ f$

```
 $(\beta \circ \alpha) b \circ (G \circ F) \circ f =$   
   $(\beta F' b \circ G \alpha b) \circ G \circ F \circ f = //$  definition of horizontal composition  
   $\beta F' b \circ G \circ F' \circ f \circ \alpha a = //$   $G \alpha b :: G\ (F\ b) \rightarrow G\ (F' b)$   
   $(G' \circ F') \circ f \circ (\beta \circ \alpha) a = //$   $\beta F' b :: G\ (F' a) \rightarrow G'\ (F' a)$ 
```

p10.5 Write a short essay about how you may enjoy writing down the evident diagrams needed to prove the interchange law.

Solution

If it's the case that:

```
 $F \xrightarrow{\beta'} F'$   
 $F' \xrightarrow{\alpha'} F''$   
 $G \xrightarrow{\beta} G'$   
 $G' \xrightarrow{\alpha} G''$ 
```

Then by the definition of horizontal composition it's simple to see that:

```
 $FG \xrightarrow{(\beta' \circ \beta)} F'G' \xrightarrow{(\alpha' \circ \alpha)} F''G''$   
 $FG \xrightarrow{(\beta' \circ \alpha')} F''G' \xrightarrow{(\beta \circ \alpha)} F''G''$ 
```

Also, by horizontal composition:

```
 $FG \xrightarrow{(\beta' \circ \beta)} F'G'$   
 $F'G' \xrightarrow{(\alpha' \circ \alpha)} F''G''$   
 $FG \xrightarrow{(\beta' \circ \beta)} F'G' \xrightarrow{(\alpha' \circ \alpha)} F''G''$ 
```

so $(\beta' \circ \beta) \circ (\alpha' \circ \alpha)$ and $(\beta' \circ \alpha') \circ (\beta \circ \alpha)$ have the equivalent effect on FG

p10.6 Create a few test cases for the opposite naturality condition of transformations

between different `Op` functors. Here's one choice:

```
op :: Op Bool Int
op = Op (\x -> x > 0)
and
f :: String -> Int
f x = read x
```

Solution

```

newtype Op r a = Op (a -> r)
contramap f (Op g) = Op (g . f)

unwrap_op :: Op a b -> b -> a
unwrap_op (Op f) x = f x

-- test 1
op1 :: Op Bool Int
op1 = Op (\x -> (x > 0))

f1 :: Bool -> Int
f1 x = if x then 1 else 0

opBoolToOpChar :: Op Bool a -> Op Char a
opBoolToOpChar (Op aToBool) = Op (\x -> if aToBool x then 'a' else 'b')

boolchar_contra_f_op1 :: Op Char Bool
boolchar_contra_f_op1 = opBoolToOpChar ((contramap f1) op1)

contra_f_boolchar_op1 :: Op Char Bool
contra_f_boolchar_op1 = contramap f1 (opBoolToOpChar op1)

test1a = (unwrap_op boolchar_contra_f_op1 True) == (unwrap_op contra_
test1b = (unwrap_op boolchar_contra_f_op1 False) == (unwrap_op contra_

-- test 2
op2 :: Op String Double
op2 = Op (\x -> show x)

f2 :: Int -> Double
f2 x = sqrt (fromIntegral x)

opStringToOpInt :: Op String a -> Op Int a
opStringToOpInt (Op aToString) = Op (\x -> length (aToString x))

stringint_contra_f_op2 :: Op Int Int
stringint_contra_f_op2 = opStringToOpInt ((contramap f2) op2)

contra_f_stringint_op2 :: Op Int Int
contra_f_stringint_op2 = contramap f2 (opStringToOpInt op2)

test2a = (unwrap_op stringint_contra_f_op2 5) == (unwrap_op contra_f_

```

```
test2b = (unwrap_op stringint_contra_f_op2 2) == (unwrap_op contra_f_
```

Section 11: Declarative Programming (No Challenges)

Section 12: Limits and Colimits

p12.1 How would you describe a pushout in the category of C++ classes?

Solution We are working in the C++ types category with subclasses as morphisms. For the span $1 \leftarrow 2 \rightarrow 3$ we have a class 2 that inherits from 1 and 3, and since we are working with colimits, the apex is some 4 such that $1 \rightarrow 4 \leftarrow 3$. The pushout is the colimit of this diagram, which is the universal 4 such that the colimit is also the subclass of every other candidate. This is the class 4 that has the maximum amount of shared functionality such that it can still be a superclass of 1 and 3.

p12.2 Show that the limit of the identity functor $\text{Id} :: C \rightarrow C$ is the initial object.

Solution The identity functor forms diagrams consisting of every item in C . The apex of each diagram must have morphisms to every other item, and the limit object must have unique morphisms to every other limit candidate, which is every other item. Therefore the limit must be the initial object.

p12.3 Subsets of a given set form a category. A morphism in that category is defined to be an arrow connecting two sets if the first is the subset of the second. What is a pullback of two sets in such a category? What's a pushout? What are the initial and terminal objects?

Solution The pushout is the intersection of the two sets (the largest set contained in them both) and the pullback is the union of those sets (the smallest set that contains them both). The initial object is the empty set and the terminal object is the "given set" that contains all of the elements and that all of the other elements are subsets of.

p12.4 Can you guess what a coequalizer is?

Solution The coequalizer is the equalizer in the opposite category. Given some 2 morphisms $f: b \rightarrow a$ and $g: b \rightarrow a$, the coequalizer is the colimit object c and associated morphism $p: a \rightarrow c$ such that $p \circ f = p \circ g$. That is, for

any other c' with morphism p' there exists some u such that $p' = p \cdot u$.

Over Set , the coequalizer defines a transformation of f and g 's codomains that makes them equal to each other.

p12.5 Show that, in a category with a terminal object, a pullback towards the terminal object is a product.

Solution Consider a diagram formed by the three object category \mathcal{I} of the form $1 \xrightarrow{f} t \xleftarrow{g} 2$ such that t is the terminal object. Since f and g are unique, the category of such diagrams is isomorphic to the category of diagrams formed by the two object discrete category consisting of only $1, 2$ without morphisms. The limit of this category is the product, so the pullback towards the terminal object is the product.

p12.6 Similarly, show that a pushout from an initial object (if one exists) is the coproduct.

Solution Consider a diagram formed by the three object category \mathcal{I} of the form $1 \xleftarrow{f} i \xrightarrow{g} 2$ such that i is the initial object. Since f and g are unique, the category of such diagrams is isomorphic to the category of diagrams formed by the two object discrete category consisting of only $1, 2$ without morphisms. The colimit of this category is the coproduct, so the pushout towards the initial object is the coproduct.

Section 13: Free Monoids

p13.1 You might think (as I did, originally) that the requirement that a homomorphism of monoids preserve the unit is redundant. After all, we know that for all a , $h(a * h(e)) = h(a * e) = h(a)$. So $h(e)$ acts like a right unit (and, by analogy, as a left unit). The problem is that $h(a)$, for all a might only cover a sub-monoid of the target monoid. There may be a "true" unit outside of the image of h . Show that an isomorphism between monoids that preserves multiplication must automatically preserve unit.

Solution Say $f: A \rightarrow A'$ is a monoid isomorphism. Then there exists some $g: A' \rightarrow A$ such that $g \circ f = \text{id}_A$. Given the unit u in A , for all a' in A' , we see $g(a' * f(u)) = g(a') * g(f(u)) = g(a') * u = g(a')$. Since g is injective, this means that $a' = a' * f(u)$, so $f(u)$ is the right unit for all a' in A' . We can do the same to show $f(u)$ is the left unit as well.

p13.2 Consider a monoid homomorphism from lists of integers with concatenation to integers with multiplication. What is the image of the empty list $[]$? Assume that all

singleton lists are mapped to the integers they contain, that is $[3]$ is mapped to 3 , etc. What's the image of $[1, 2, 3, 4]$? How many different lists map to the integer 12 ? Is there any other homomorphism between the two monoids?

Solution The image of the empty list is 1 , and the image of $[1, 2, 3, 4]$ is $1 * 2 * 3 * 4 = 24$. The lists $[12, 1]$ $[1, 12]$ $[6, 2]$ $[2, 6]$ $[3, 4]$ $[4, 3]$ $[2, 2, 3]$ $[2, 3, 2]$ $[3, 2, 2]$ all map to 12 .

The function that maps all lists of integers to 1 is also a homomorphism, because the unit $[1]$ is preserved and for any two lists l_1, l_2 we see that $h(l_1 ++ l_2) = 1 = 1 * 1 = h l_1 * h l_2$

p13.3 What is the free monoid generated by a one-element set? Can you see what it's isomorphic to?

Solution This monoid is lists of unit $()$ with concatenation. This is isomorphic to integers over addition.

```
forward :: List () -> Int
forward x = length x

inverse :: Int -> List ()
inverse x = replicate x [()]
```

Section 14: Representable Functors

p14.1 Show that the hom-functors map identity morphisms in \mathcal{C} to corresponding identity functions in \mathbf{Set} .

Solution When we apply the functor $\mathcal{C}(a, -)$ to some function f , we get a function that performs the action $\mathcal{C}(a, f) h = f \circ h$ on any morphism $h: a \rightarrow x$ in the homset $\mathbf{Hom}(a, x)$. If $f: x \rightarrow x$ is the identity morphism, $f \circ h = h$, so $\mathcal{C}(a, f) h = h$, and $\mathcal{C}(a, f)$ is the identity morphism as well.

p14.2 Show that `Maybe` is not representable.

Solution If `Maybe` were representable, then we would be able to implement a function of the form `beta :: Maybe x -> (a -> x)`. However, it is not possible to implement a function that accepts `None` and return `a -> x` for any arbitrary `x` type.

p14.3 Is the `Reader` functor representable?

Solution Yes, the `Reader` functor is the hom-functor over haskell types and it is isomorphic to itself.

p14.4 Using `Stream` representation, memoize a function that squares its argument.

Solution

```
data Stream x = Cons x (Stream x)
instance Representable Stream where
  type Rep Stream = Int
  tabulate f = Cons (f 0) (tabulate (f . (+1)))
  index (Cons b bs) n = if n == 0 then b else index bs (n - 1)

squareArg :: Int -> Int
squareArg x = x * x

memoizedSquares :: Stream Int
memoizedSquares = tabulate squareArg

zerothSquare :: Int
zerothSquare = index memoizedSquares 0
zerothSquareTrue = zerothSquare == 0

thirdSquare :: Int
thirdSquare = index memoizedSquares 3
thirdSquareTrue = thirdSquare == 9

fifthSquare :: Int
fifthSquare = index memoizedSquares 5
fifthSquareTrue = fifthSquare == 25
```

p14.5 Show that `tabulate` and `index` for `Stream` are indeed the inverse of each other. (Hint: use induction.)

Solution We want to prove that for all `n`, `index tabulate f n = f n` **Base Case**


```
index (tabulate f) 0 = // definition of tabulate
  index (Cons (f 0) (tabulate (f . (+1)))) 0 = // definition of index
f 0
```

Inductive Step

```
index (tabulate f) n = // definition of tabulate
  index (Cons (f 0) (tabulate (f . (+1)))) n = // definition of index
index (tabulate (f . (+1))) (n - 1) = // inductive assumption
f . (+1) . (n - 1) =
f n
```

Section 15: The Yoneda Lemma

p15.1 Show that the two functions `phi` and `psi` that form the Yoneda isomorphism in Haskell are inverses of each other.

```
phi :: (forall x . (a -> x) -> F x) -> F a
phi alpha = alpha id
psi :: F a -> (forall x . (a -> x) -> F x)
psi fa h = fmap h fa
```

Solution Note `psi` can be written as `psi fa = \h -> fmap h fa` **Forward**

```
(phi . psi) fa =
  phi (\h -> fmap h fa) =
  (\h -> fmap h fa) id =
  fmap id fa =
  fa
```

Backward

```
(psi . phi) alpha =
  psi (alpha id) =
  \h -> fmap h (alpha id) =
  \h -> (alpha h id) =
  \h -> alpha h =
  alpha
```

p15.2 A discrete category is one that has objects but no morphisms other than identity morphisms. How does the Yoneda lemma work for functors from such a category?

Solution Any homfunctor $C(a, -)$ from the discrete category maps a to the singleton set and all other objects to the empty set. For any functor F from the discrete category to Set , there are N morphisms (the item-selection morphisms) between the singleton set and $F a$, where N is the number of elements of $F a$. Since there is one morphism from the empty set to each other set, each of those N morphisms from singleton to $F a$ indicate a unique natural transformation from $C(a, -)$ to F , so there is one-to-one correspondence between these natural transformations and elements of $F a$.

p15.3 A list of units `[]` contains no other information but its length. So, as a data type, it can be considered an encoding of integers. An empty list encodes zero, a singleton `[]` (a value, not a type) encodes one, and so on. Construct another representation of this data type using the Yoneda lemma for the `list` functor.

Solution By the Yoneda lemma, the natural transformations from $C(a, -)$ (in this case $() \rightarrow x$) to F (in this case `List x`) are one-to-one with the elements of $F a$. So the data type $D (() \rightarrow x) \rightarrow List x$ is another representation of `List ()`.

It's pretty easy to see why this is the case - a function of the form $f: () \rightarrow x$ is essentially a container for a single value of x . So the elements of D are all of the form:

```
d1 f = [f ()]
d2 f = [f (), f ()]
...
```

Section 16: Yoneda Embedding

p16.1 Express the co-Yoneda embedding in Haskell.

Solution

```
forward :: (a -> b) -> ((x -> a) -> (x -> b))
forward atob = \f -> atob . f

backward :: ((x -> a) -> (x -> b)) -> (a -> b)
backward xToAToXToB = \a -> (xToAToXToB id) a
```

p16.3 Work out the Yoneda embedding for a monoid. What functor corresponds to the monoid's single object? What natural transformations correspond to monoid morphisms?

Solution In the single-element category view of monoid, we have a single element and the morphisms follow the monoid association rules. We will call this category \mathbf{M} . The Yoneda embedding maps the single object a to the functor $\mathbf{M}(a, -)$, which is the functor in $[\mathbf{M}, \mathbf{Set}]$ that maps the single element a to the set $\mathbf{M}(a, a)$. The Yoneda embedding maps each morphism in \mathbf{M} to the identity natural transformation that maps the functor $\mathbf{M}(a, -)$ to itself.

p16.4 What is the application of the covariant Yoneda embedding to preorders? (Question suggested by Gershon Bazerman.)

In a preorder category \mathbf{C} , if and only if a morphism $f: b \rightarrow a$ exists, we have exactly one natural transformation between $\mathbf{C}(a, -)$ and $\mathbf{C}(b, -)$. Since there are no functions that map non-empty sets into the empty set, we see that if $\mathbf{C}(a, x)$ is nonempty, then $\mathbf{C}(b, x)$ must be nonempty as well.

Therefore, we have the condition: $b \leq a$, if and only if for all x , $a \leq x$ implies $b \leq x$

p16.5 Yoneda embedding can be used to embed an arbitrary functor category $[\mathbf{C}, \mathbf{D}]$ in the functor category $[[\mathbf{C}, \mathbf{D}], \mathbf{Set}]$. Figure out how it works on morphisms (which in this case are natural transformations).

Solution For any natural transformation Nat_{AB} between the functors $F_b: \mathbf{C} \rightarrow \mathbf{D}$ and $F_a: \mathbf{C} \rightarrow \mathbf{D}$ such that $\text{Nat}_{AB}: F_b \rightarrow F_a$, the Yoneda embedding maps it to the natural transformation:

```
NYoneda: [C, D] (Fa, -) -> [C, D] (Fb, -)
```

Where

```
[C, D] (Fa, -): Fx -> NatAX
NatAX: Fa -> Fx
[C, D] (Fb, -): Fx -> NatBX
NatBX: Fb -> Fx
```

NYoneda operates on $[C, D] (Fa, -)$ by post-composing Nat_{AB} to the natural transformations in the output NatAX , which maps the output set to NatBX , which maps $([C, D] (Fa, -): Fx \rightarrow \text{NatAX}) \rightarrow ([C, D] (Fb, -): Fx \rightarrow$

Section 17: It's All About Morphisms

p17.1 Consider some degenerate cases of a naturality condition and draw the appropriate diagrams. For instance, what happens if either functor F or G map both objects a and b (the ends of $f :: a \rightarrow b$) to the same object, e.g., $F a = F b$ or $G a = G b$? (Notice that you get a cone or a co-cone this way.) Then consider cases where either $F a = G a$ or $F b = G b$. Finally, what if you start with a morphism that loops on itself — $f :: a \rightarrow a$?

Solution

For the following subproblems, let's assume we have some function $f :: a \rightarrow b$ and natural transformation α between functors F and G .

p17.1.1

Say $F a = F b$. Then $G a = G b$ because:

```
(α . Ff) Fa =
  α . Fb =
  α . Fa =
  Ga

(Gf * α) Fa =
  Gf Ga =
  Gb
```

p17.1.2

Say $G a = G b$. Then $\alpha_B Fb = \alpha_A Fa$, but we can't conclude that $Fb = Fa$, because it's possible that G is the constant functor.

p17.1.3

Say $F a = G a$. Then $Gf :: Fa \rightarrow Gb$ and since $\alpha_A Fa = Ga$, α_A is the identity.

p17.1.4

Say $F b = G b$. Then $Ff :: Fa \rightarrow Gb$ and since $\alpha_B Fb = Gb$, α_B is the identity.

p17.1.5

Say $f: a \rightarrow a$. Then $\alpha_A * Ff = Gf * \alpha_A$.

Section 18: Adjunctions

p18.1 Derive the naturality square for ψ , the transformation between the two (contravariant) functors:

```
a -> C(L a, b)
a -> D(a, R b)
```

Solution Say we have

```
f  :: a1 -> a2
F f :: C(L a1, b) -> C(L a2, b)
G f :: D(a1, R b) -> D(a2, R b)
```

Where L and R are functors

```
L :: D(a, R b) -> C(L a, b)
R :: C(L a, b) -> D(a, R b)
```

and define the natural transformation $\psi: G \rightarrow F$ such that

```
ψg1 :: D(a1, R b) -> C(L a1, b)
ψg2 :: D(a2, R b) -> C(L a2, b)
```

Now consider the morphisms $g1: a1 \rightarrow R$ and $g2: a2 \rightarrow R$. We want to prove that $F f * \psi g1 = \psi g2 * G f$

```
ψg2 * G f G a1 = // definition of f
ψg2 * G a2      = // definition of G
ψg2 * D(a2, R b) = // definition of ψg2
C(L a2, b)       = // definition of G
F a2             = // definition of F
F f * F a1       = // definition of f
F f * ψg1 G a1   = // definition of ψg1
```

p18.2 Derive the counit ϵ starting from the hom-sets isomorphism in the second definition of the adjunction.

Solution Assume that $C(L d, c) \cong D(d, R c)$ holds for any c in C and d in

D. We want to prove that there exists some natural transformation $\varepsilon :: L \circ R \rightarrow I_C$.

Say $d = R \circ c$, then $C((L \circ R) \circ c, c) \cong D(R \circ c, R \circ c)$. Since $D(R \circ c, R \circ c)$ contains at least the identity, our natural transformation from $D(R \circ c, R \circ c) \rightarrow C((L \circ R) \circ c, I_C)$ must map to a non-empty set. Therefore, we have some set of morphisms that map from $(L \circ R) \circ c \rightarrow I_C$ for any c . These morphisms form a natural transformation from $L * R \rightarrow I_C$, which is ε .

p18.3 Complete the proof of equivalence of the two definitions of the adjunction.

Solution In order to prove that the two definitions are equivalent, we need to prove the equivalence of the isomorphism $C(L \circ d, c) \cong D(d, R \circ c)$ and the existence of two natural transformations: the unit η and the counit ε .

In the text we proved that $C(L \circ d, c) \cong D(d, R \circ c)$ implies the existence of the η and that the existence of the η and ε implies the existence of a mapping from $C(L \circ d, c)$ to $D(d, R \circ c)$. In p18.2, we proved that $C(L \circ d, c) \cong D(d, R \circ c)$ implies the existence of the ε . Therefore, we still need to prove that the existence of the η and ε implies the existence of $\psi :: D(d, R \circ c) \rightarrow C(L \circ d, c)$.

For some morphism $f :: d \rightarrow R \circ c$, we can apply $\varepsilon_C * L$ to form the morphism:

```

\varepsilon_C . L f =
  L d -> \varepsilon_C L . R c =
  L d -> c =
  \psi f

```

p18.4/5 Show that the coproduct can be defined by an adjunction. Start with the definition of the factorizer for a coproduct. Show that the coproduct is the left adjoint of the diagonal functor.

Solution (In this solution, we assume C is `Set` or `Hask`) We want to prove that $C(\text{Either } a \ b, c) \cong (C \times C)(<a, b>, \Delta \ c)$. A homset in $C \times C$ is $(C \times C)(<a, b>, \Delta \ c)$, which consists of pairs of functions $a \rightarrow c$, $b \rightarrow c$ and a homset in C is $C(\text{Either } a \ b, c)$, which consists of functions $(\text{Either } a \ b \rightarrow c)$.

We can define a natural transformation between these two homsets with the `factorizer` and `inversefactorizer` functions.

```
factorizer :: (C×C) (<a, b>, Δ c) -> C(Either a b, c)
inversefactorizer :: C(Either a b, c) -> (C×C) (<a, b>, Δ c)
```

We can write these in pseudo-haskell as

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

inversefactorizer :: (Either a b -> c) -> ((a -> c), (b -> c))
inversefactorizer m = (\a -> m Left a), (\b -> m Right b)
```

Now note that because these are both polymorphic in a, b, c , both `factorizer` and `inversefactorizer` are natural, so $C(\text{Either } a \text{ } b, c) \cong (C \times C)(<a, b>, \Delta c)$.

p18.6 Define the adjunction between a product and a function object in Haskell.

Solution

```
producttofunction :: ((z, a) -> b) -> (z -> (a -> b))
producttofunction f = \z -> (\a -> f (z,a))

functiontoproduct :: (z -> (a -> b)) -> ((z, a) -> b)
functiontoproduct f = \z_a -> ((f (fst z_a)) (snd z_a))
```

Section 19: Free/Forgetful Adjunctions

p19.1 Consider a free monoid built from a singleton set as its generator. Show that there is a one-to-one correspondence between morphisms from this free monoid to any monoid m , and functions from the singleton set to the underlying set of m .

Solution

Forward Consider a morphism from the free monoid with the singleton set as

its generator to m . This morphism maps the generator element e to some m_1 in m . There exists exactly one function in the homset between the singleton set and the underlying set of m that maps $()$ to m_1 , so we can define a forward mapping.

Backward Consider a function from the singleton set to the underlying set of m . This function “chooses” a single element m_1 from the underlying set of m . We can define exactly one homomorphism between the singleton free monoid and m that maps the generator element e to m_1 , since any such homomorphism must satisfy the following:

```
1 -> unit
e -> m1
ee -> m1m1
eee -> m1m1m1
```

so there is exactly one such homomorphism and we can define a backward mapping.

Section 20: Monads: Programmer's Definition (No Challenges)

Section 21: Monads and Effects (No Challenges)

Section 22: Monads Categorically (No Challenges)

Section 24: F-Algebras

p24.1 Implement the evaluation function for a ring of polynomials of one variable. You can represent a polynomial as a list of coefficients in front of powers of x . For instance, $4x^2 - 1$ would be represented as (starting with the zero'th power) $[-1, 0, 4]$.

Solution


```
polyEval :: [Double] -> Double -> Double
polyEval coefficients value = foldr (\ (power, coeff) sumSoFar -> sum
isTrue = 99.0 == (polyEval [-1, 0, 4] 5)
```

p24.2 Generalize the previous construction to polynomials of many independent variables, like x^2y-3y^3z .

Solution

```
raiseAndProd :: [Double] -> [Double] -> Double
raiseAndProd values powers = foldr (\(value, power) prodSoFar -> pro

polyMultiEval :: [(Double, [Double])] -> [Double] -> Double
polyMultiEval coeffsExps values = foldr (\ (coeff, powers) sumSoFar -

isTrue1 = -2580.0 == (polyMultiEval [(1, [2, 1, 0]), (-3, [0, 3, 1])]
isTrue2 = 1.0 == (polyMultiEval [(1, [2, 1])] [1, 1])
```

p24.3 Implement the algebra for the ring of 2×2 matrices.

Solution

```

data MatExpr = RZero
  | ROne
  | RCompA
  | RCompB
  | RCompC
  | RCompD
  | RAdd MatExpr MatExpr
  | RMul MatExpr MatExpr
  | RNeg MatExpr

type MatrixTwoTwo = (Double, Double, Double, Double)

mCompA :: MatrixTwoTwo
mCompA = (1, 0, 0, 0)

mCompB :: MatrixTwoTwo
mCompB = (0, 1, 0, 0)

mCompC :: MatrixTwoTwo
mCompC = (0, 0, 1, 0)

mCompD :: MatrixTwoTwo
mCompD = (0, 0, 0, 1)

mZero :: MatrixTwoTwo
mZero = (0, 0, 0, 0)

mEye :: MatrixTwoTwo
mEye = (1, 0, 1, 0)

mAdd :: MatrixTwoTwo -> MatrixTwoTwo -> MatrixTwoTwo
mAdd (a1,b1,c1,d1) (a2,b2,c2,d2) = (a1 + a2, b1 + b2, c1 + c2, d1 + d2)

mMult :: MatrixTwoTwo -> MatrixTwoTwo -> MatrixTwoTwo
mMult (a1,b1,c1,d1) (a2,b2,c2,d2) = (a1 * a2 + b1 * c2, a1 * b2 + b1 * c2, a1 * c2 + b1 * d2, a1 * d2 + b1 * d2)

mNeg :: MatrixTwoTwo -> MatrixTwoTwo
mNeg (a1,b1,c1,d1) = (-a1,-b1,-c1,-d1)

evalZ :: MatExpr -> MatrixTwoTwo
evalZ RZero = mZero

```

```

evalZ ROne = mEye
evalZ RCompA = mCompA
evalZ RCompB = mCompB
evalZ RCompC = mCompC
evalZ RCompD = mCompD
evalZ (RAdd e1 e2) = mAdd (evalZ e1) (evalZ e2)
evalZ (RMul e1 e2) = mMult (evalZ e1) (evalZ e2)
evalZ (RNeg e) = mNeg (evalZ e)

matrixExpression :: MatExpr
matrixExpression = RMul (RAdd RCompA RCompB) (RAdd RCompC RCompD)
isTrue = (1.0, 1.0, 0.0, 0.0) == (evalZ matrixExpression)

```

p24.4 Define a coalgebra whose anamorphism produces a list of squares of natural numbers.

Solution

```

newtype Fix f = Fix (f (Fix f))
unFix :: Fix f -> f (Fix f)
unFix (Fix x) = x

cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix

ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg

data StreamF e a = StreamF e a deriving Functor

toListC :: Fix (StreamF e) -> [e]
toListC = cata al
  where al :: StreamF e [e] -> [e]
        al (StreamF e a) = e : a

nat :: [Int] -> StreamF Int [Int]
nat (p : ns) = StreamF (p^2) ns

squaresStream :: Fix (StreamF Int)
squaresStream = ana nat [0..]

squaresList :: [Int]
squaresList = toListC squaresStream

```

p24.5 Use `unfoldr` to generate a list of the first n primes.

Solution

```

listSieve :: [Int] -> Maybe (Int, [Int])
listSieve (p : ns) = Just (p, (filter (notdiv p) ns))
  where notdiv p n = n `mod` p /= 0

primeFilteredList :: [Int]
primeFilteredList = unfoldr listSieve [2..]

isTrue1 = primeFilteredList!!0 == 2
isTrue2 = primeFilteredList!!3 == 7

```

Section 25: Algebras for Monads

p25.1 What is the action of the free functor $F :: C \rightarrow C^T$ on morphisms. Hint: use the

naturality condition for monadic μ .

Solution First, note that $F\ a = (T\ a, \mu a)$. Since μ is natural, we see that $T\ f \cdot \mu a = \mu b \cdot (T \cdot T)\ f$. Therefore, for some $f :: a \rightarrow b$, the action of F on f is:

$$\text{fmap } f\ (T\ a, \mu a) = (\text{fmap } f\ T\ a, T\ f \cdot \mu a)$$

p25.2 Define the adjunction: $U^W \dashv F^W$

Solution First, we define the unit $\eta :: I \rightarrow F^W \cdot U^W$. Since it's the case that

$$\begin{aligned} F^W \cdot U^W\ (W\ a, f) &= \\ F^W\ (W\ a) &= \\ (W\ W\ a, \delta W a) \end{aligned}$$

η needs to map $(W\ a, f) \rightarrow (W\ W\ a, \delta W a)$. We can accomplish this by using f , the co-evaluator of the co-algebra, to define the component of η at $(W\ a, f)$

Next, we define the co-unit $\varepsilon :: U^W \cdot F^W \rightarrow I$. Since it's the case that:

$$\begin{aligned} U^W \cdot F^W\ a &= \\ U^W \cdot (W\ a, \delta a) &= \\ W\ a \end{aligned}$$

ε needs to map $W\ a \rightarrow a$ so we can use the `extract` method of the co-monad to define the component of ε at $W a$.

p25.3 Prove that the above adjunction reproduces the original comonad.

Solution First, we can use the co-unit of the adjunction ε as the co-monadic `extract`, since $\varepsilon W a\ W\ a = a$

Next, we can use the unit of the adjunction to define the co-monadic `duplicate` as the horizontal composition of three natural transformations $U^W \circ \eta \circ F^W$ where $U^W: U^W \rightarrow U^W$ and $F^W: F^W \rightarrow F^W$. Since F^W lifts a to $(W\ a, \delta a)$, η picks the co-evaluator δa which maps $W\ a \rightarrow W\ W\ a$ and

U^W has no action on morphisms, we see that $\text{duplicate} = U^W \circ \eta \circ F^W$.

Section 26: Ends and Coends (No Challenges)

Section 27: Kan Extensions (No Challenges)

Section 28: Enriched Categories (No Challenges)

Section 29: Topoi

p29.1 Show that the function f that is the pullback of true along the characteristic function must be injective.

Solution If $f: a \rightarrow b$ is the pullback of true along the characteristic function, then for any a^* , $f^*: a^* \rightarrow b$, there exists some unique $h: a^* \rightarrow a$ such that $f^* = f \circ h$.

Consider the case where a^* is the image of f and f^* is the identity. If f is not injective, then for the $e_1, e_2, e_1 \neq e_2$ in a such that $f(e_1) = f(e_2)$, h can map $f(e_1) = f(e_2)$ to either e_1 or e_2 . Therefore h would not be unique, which implies that f must be injective.

Section 30: Lawvere Theories

p30.1 Enumerate all morphisms between 2 and 3 in F (the skeleton of FinSet).

Solution $(0 \rightarrow 0, 1 \rightarrow 0), (0 \rightarrow 0, 1 \rightarrow 1), (0 \rightarrow 0, 1 \rightarrow 2), (0 \rightarrow 1, 1 \rightarrow 0), (0 \rightarrow 1, 1 \rightarrow 1), (0 \rightarrow 1, 1 \rightarrow 2)$

p30.2 Show that the category of models for the Lawvere theory of monoids is equivalent to the category of monad algebras for the list monad.

Solution First, let's note that the category of models of the Lawvere theory for monoids is equivalent to the category of all monoids, Mon . Now we will prove

that Mon is equivalent to the category of monad algebras for the list monad.

First, given a monoid over the set a , we can produce an algebra (a, f) where f maps the list L to the monoidal product of the elements in L . Next, given an algebra (a, f) , we can produce a monoid over a by defining the monoidal product of a_1, a_2 to be $f([a_1] \text{ cat } [a_2])$. The unit of this monoid is $[]$, and because of the monad condition $f \circ \mu_a = f \circ T f$ we see that:

$$\begin{aligned} f[a_1, f[a_2, a_3]] &= \\ f[a_1, a_2, a_3] &= \\ f[f[a_1, a_2], a_3] \end{aligned}$$

So the monoid associativity law is automatically satisfied.

p30.3 The Lawvere theory of monoids generates the list monad. Show that its binary operations can be generated using the corresponding Kleisli arrows.

Solution The binary operations in the Lawvere theory of monoids are elements of the homset $\text{LMon}(2, 1)$, which are functions of two arguments that we can implement with only the monoidal operator. Each of these functions can be defined by a list composed of only those 2 unique elements. Since each Kleisli arrow in the hom-set $\text{KlT}(1, 2)$ corresponds to a list composed of elements from the 2 element set, we can represent each binary operation in the Lawvere theory of monoids with a Kleisli arrow in $\text{KlT}(1, 2)$.

p30.4 FinSet is a subcategory of Set and there is a functor that embeds it in Set . Any functor on Set can be restricted to FinSet . Show that a finitary functor is the left Kan extension of its own restriction

Solution Say $K: \text{Set} \rightarrow \text{FinSet}$ is a functor that embeds a set into FinSet , such that for any finite set n in Set , $K n = n$. Then $\text{FinSet}(K n, a)$ is a hom-set between elements in FinSet , and so $\text{FinSet}(K n, a) = a^{(K n)} = a^n$.

Now consider some finitary functor F . The left Kan extension of F 's restriction to FinSet along K is

```

LanK F a =
  ∫n FinSet(K n, a) × F n =
  ∫n an × F n = \\ definition of finitary functor
F

```

So a finitary functor is the left Kan extension of its own restriction.

Section 31: Monads, Monoids, and Categories

p31.1 Derive unit and associativity laws for the tensor product defined as composition of endo-1-cells in a bicategory.

Solution

Unit Law The left and right compositions of any endo-1-cell T and the identity 1-cell id are $T \cdot id$ and $id \cdot T$. By the definition of a bicategory there exist invertible 2-cells mapping each of these endo-1-cells to T .

Associativity Law Given three endo-1-cells T_1, T_2, T_3 , by the definition of a bicategory there exists an invertible 2-cell that maps between $((T_1 \cdot T_2) \cdot T_3$ and $T_1 \cdot (T_2 \cdot T_3)$.

p31.2 Check that monad laws for a monad in Span correspond to identity and associativity laws in the resulting category.

Solution A monad in Span consists of an endo-1-cell that has the sets Ar, Ob with the functions

```

dom :: Ar -> Ob
cod :: Ar -> Ob

```

and the associated 2-cells:

```

μ: Ar x Ar -> Ar
η: Ob -> Ar

```

This monad defines a category consisting of the objects in Ob and the arrows in

Ar , where each arrow in Ar connects $\text{dom } Ar$ to $\text{cod } Ar$.

Identity η assigns an identity arrow to each object such that

$$\begin{aligned}\text{dom } \eta &= \text{id} \\ \text{cod } \eta &= \text{id}\end{aligned}$$

Therefore for any object $o1$ in Ob and arrow $a1$ in Ar where $\text{cod } a1 = o1$, we see that:

$$\begin{aligned}\text{dom } (\mu (a1, \eta o1)) &= \text{dom } a1 \\ \text{cod } (\mu (a1, \eta o1)) &= \text{cod } (\eta o1) = o1 = \text{cod } a1\end{aligned}$$

So the composition of an arrow with the identity arrow does not change that arrow's domain or codomain.

Associativity By the monoid law for μ

$$\mu (Ar \times \mu (Ar \times Ar)) = \mu (\mu (Ar \times Ar) \times Ar)$$

Therefore, for any arrows $a1, a2, a3$, we see that

$$a1 \cdot (a2 \cdot a3) = (a1 \cdot a2) \cdot a3$$

p31.3 Show that a monad in Prof is an identity-on-objects functor.

Solution In Prof , we define a monad with an endo-profunctor T such that $T: C^{\text{op}} \times C \rightarrow \text{Set}$. The composition of profunctors is

$$(q \cdot p) \ a \ b = \int^c p \ c \ a \times q \ b \ c$$

So the composition of T with itself is:

$$\begin{aligned}(T \cdot T) \ C \ C &= \\ \int^c T \ c \ C \times T \ C \ c &= // \text{ existential quantifier} \\ T \ C \ C\end{aligned}$$

Which implies that T must map each object in C to itself.

p31.4 What's a monad algebra for a monad in `Span` ?

Solution Given a monad `m` over some object `a`, we form an algebra over this monad with a map `alg :: m a -> a` that satisfies commutativity conditions. For a monad in `Span`, we can use `dom` or `cod` for `alg`.

Identity `alg . ηa = ida` This holds by the definition of `η` for `Span`

Associativity `alg . μa = alg . m alg` Without loss of generality, we can see the following:




```
dom . μa (a1, a2) =  
  dom a1 =  
  dom . m dom (a1, a2)
```

Tags: Category Theory, Functional Programming, Mathematics, Solutions



← **PREVIOUS POST (/2018-10-17-REPCOMP/)**

NEXT POST → (/2019-06-01-ICLR/)

 (<https://github.com/dshieble>)  (<mailto:danshiebler@gmail.com>)
 ([https://linkedin.com/in/dan-shiebler-10219b42#Dan Shiebler](https://linkedin.com/in/dan-shiebler-10219b42#Dan%20Shiebler))

Dan Shiebler • 2021

Theme by beautiful-jekyll (<http://deanattali.com/beautiful-jekyll/>)