Contents

These notes serve the following purpose. Firstly, they list and summaries important concepts in Category Theory and how they can be expressed in Haskell. Secondly, the book content is compared to other books I am reading in parallel.

In composing this text I will follow these rules: - Notes: are for references to other books and papers - Bold font: is for definitions and questions of challenges - Italic: If things that still need to be defined

appear in text

Table of content

0 Preface

‘Category Theory for Programmers’ was written as a series of blog posts that are at times very vague about the concepts that are introduced

reading .... text was very insightful for me. He actually wrote his second chapter in such a way that Bartosz blog posts make more sense.

> [CH], Chapter 2, page 17: To establish a link between functional programming and category theory, we need to find a category that is applicable. Observe that a type in a programming language, corresponds to a set in mathematics. Indeed, the type `int` in C based languages, corresponds to some finite set of numbers, the type `char` to a set of letters like ‘a’, ‘z’ and ‘$’, and the type `bool` is a set of two elements (true and false). This category, the category of types, turns out to be a very fruitful way to look at programming.

Category	Objects	Arrows	Notes
Hask	types	maps	
Set	sets	maps	
Mon	monoids	functors	Action

Category	Objects	Arrows	Notes
Grph	graphs		Connection
Meas	measure spaces		Amount
Top	topological spaces	continuous functions	Neighborhood
Vect	vector spaces	linear transformations	
Grp	groups	group homomorphisms	Reversable action, symmetry
Para			
Poly			
Cat	categories	functors	Action in context, structure

Table 0-1: Some examples of omnipresent categories

In all categories listed in Talbe 1-1-1, the arrows correspond to *structure preserving maps* (which do not necessarily need to be functions, but can be assignments such as numbers to football players). For the application of category theory to Haskell we work with the category **Hask** which is a suset of **Set** and contains all Haskell types. More on that in Chapter 2.

1 Category: The Essence of Composition

1-1 Arrows as Functions & 1-2 Properties of Composition

In Haskell functions (`f` and `g`) between objects (`A`, `B` and `C`) and their composition can be declared as

	Mathematical Notation	Haskell Type Signature
functions	$f : A \to B$ and $g : B \to C$	`f :: A -> B` and `g :: B -> C`
function composition	$g \circ f : A \to C$	`g . f :: A -> C`
identity function	$\mathrm{id}_A : A \to A$	`id :: a -> a` (part of Prelude)

Table 1-1-1: Mappings between objects in their mathematical and Haskell notation.

In mathematics in general, and in category theory specifically, a commutative diagram is a diagram such that all directed paths in the diagram with the same start and endpoints lead to the same result.

> [SSC], Definition 3.42, page 96: A *diagram* D in $\mathcal{C}$ is a functor $D : \mathcal{J} \to \mathcal{C}$ from any category $\mathcal{J}$, called the *indexing category* of the diagram D. We say that D *commutes* if $D(f) = D(f')$ holds for every parallel pair of morphisms $f, f' : a \to b$ in $\mathcal{J}$.

1-2 Properties of Composition

For any category the following rules two apply, which specify it's composability.

1) Identity Laws In Haskell, the identity function is part of the standard library (Prelude). The identity function is polymorphic because it accepts any type, indicated by the type variable a.

```haskell
id :: a -> a
id x = x

f :: A -> B
f . id == f
id . f == f
```

2) Associative Law As equality is not defined for functions in Haskell, we can it as pseudo code:

```haskell
f :: A -> B
g :: B -> C
h :: C -> D
h . (g . f) == (h . g) . f == h . g . f
```

1-4 Challenges

1. Implement, as best as you can, the identity function in your favorite language. Skip

1. Implement, as best as you can, the identity function in your favorite language. Create a .hs file and write down the identity function:

```haskell
-- chapter_1.hs:
identity :: a -> a
identity x = x
```

Then in terminal enter ghci:

```
ghci: :l chapter_1.hs
ghci: compose sqrt sqrt 1.0
1.0
ghci: compose (+1) (+1) 1.0
```

```
5 3
```

2. Implement the composition function. It takes two functions as arguments and returns a function that is their composition. You can look up what signature our function compose must have, by typing `:t (.)` into ghci. In the same file created above type the new function

```
1 -- chapter_1.hs:
2 compose ::  (b -> c) -> (a -> b) -> a -> c
3 compose g f = (g . f)
```

Testing it in ghci should give the same resutls as:

```
1 ghci: :l chapter_1.hs
2 ghci: compose sqrt sqrt 1.0
3 1.0
4 ghci: compose (+1) (+1) 1.0
5 3
```

3. Write a program that tries to test that your composition function respects identity. To verify that our answer is correctly implemented, we need side effects which we can't use in Prelude. Thus, I use IO:

```
1 main :: IO ()
2 main = do
3   let test_a = compose sqrt identity 4.0 == 2.0
4   let test_b = compose identity sqrt 4.0 == 2.0
5
6   print $ show test_a
7   print $ show test_b
```

and execute the script in the terminal using:

```
1 $ runghc chapter_1.hs
2 $ "True"
3 $ "True"
```

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

5. Is Facebook a category, with people as objects and friendships as morphisms? No, since friendships are not composable.

6. When is a directed graph a category? If the digraph observes the identity and associativity law by containing self-loops and if `a -> b -> c` then it must follow that `a -> c`.

2 Types and Functions

2-5 Pure and Dirty Functions

Putting it simple, all functions are pure in Haskell as long as they do not interact with the `IO Monad`. Situations in which you have to perform I/O actions are, i.e., when implementing random number generators, which have to interact with the outside world (such interaction is also called *side effects*) to generate a reasonably random number. This can be done using, i.e., the `getStdGen` I/O action of the `System.Random` module.

Through Haskell’s pure functions we can make the mapping to the category $\mathbf{Set}$ and reason with Haskell in a categorical manner (while other imparativ programming langues do it through a more artificial way).

2-6 Examples of Types

In programming language, types are sets (often bounded by their bit range)!

`Void` Type for the Empty Set In the Haskell module `Data.Void` it is declared as `absurd :: Void -> a`. The type `Void` represents falsity, and the type of the function `absurd` corresponds to the statement that from falsity follows anything. The `absurd` function can never be executet, as it can no term of type `Void` exists.§

> [SSC], page 7: The empty set $\varnothing := \{\}$ has no elements and is a subset of every other set.

`Unit` Type for the Singleton Set A singleton set contains only on object, such as `44` or `('a', 'b')`. Whatever is being fed into a singleton set returns always this one object,

```haskell
f44 :: a -> Integer
f44 _ = 44
```

> [CM], page 19: A singleton is also called a *point* or *one-element set*, since when it appears in composition, it focuses the preceding mappings to a single point projection.

A special kind of singleton set, that is used for side-effects, is the one that contains the `unit` type,

```haskell
funit :: a -> ()
funit _ = ()
```

`Tuple` Types Tuple types, `(x,y)`, are not discussed in this section, but it will be very important for later in Section 5-5 when introducing products, $x \times y$. For more information on them read [PC], Section 2.3, page 44.

`Bool` and other Type This introduction to types from a categorical perspective is quite different from any book that simply introduces a programming language. The latter often introduces many more types such as `Bool`, `List`, `Int` and so on. Those types however, are a composition of more fundamental concepts and will be introduced later starting with algebraic data types, Section 6.

2-7 Challenges

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.

Searching the web provided my plenty examples on how to write a Memoization function wrapper by hand for specific data types, specifically for Int for Fibonacci sequence, but I haven't figured out how to write a short polymorphic wrapper. Digging into it reveals much I have no idea about yet, such as applying the Yoneda Lemma: http://conal.net/blog/posts/memoizing-polymorphic-functions-via-unmemoization Conal implemented this approach in the MemoTrie module, loaded as `Data.MemoTrie`

More reasons for headaches in Haskell are explained at the beginning of Chapter 4..

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

No, if it would work it would be a random number generator.

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?

Skip

4. Which of these C++ functions are pure? Try to memoize them and observe what happens when you call them multiple times: memoized and not.

Skip

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

There are 2^2 possible *endomaps* `f: Bool -> Bool` (see [CM], page 34).

```haskell
boolMap1 :: Bool -> Bool
boolMap1 True = True
boolMap1 False = False

boolMap2 :: Bool -> Bool
boolMap2 True = False
boolMap2 False = False

boolMap3 :: Bool -> Bool
boolMap3 True = True
boolMap3 False = True

boolMap4 :: Bool -> Bool
boolMap4 True = False
boolMap4 False = True
```

6. Draw a picture of a category whose only objects are the types `Void`, `()` (unit) and `Bool`; with arrows corresponding to all possible functions between these types. Label the arrows with the names of the functions.

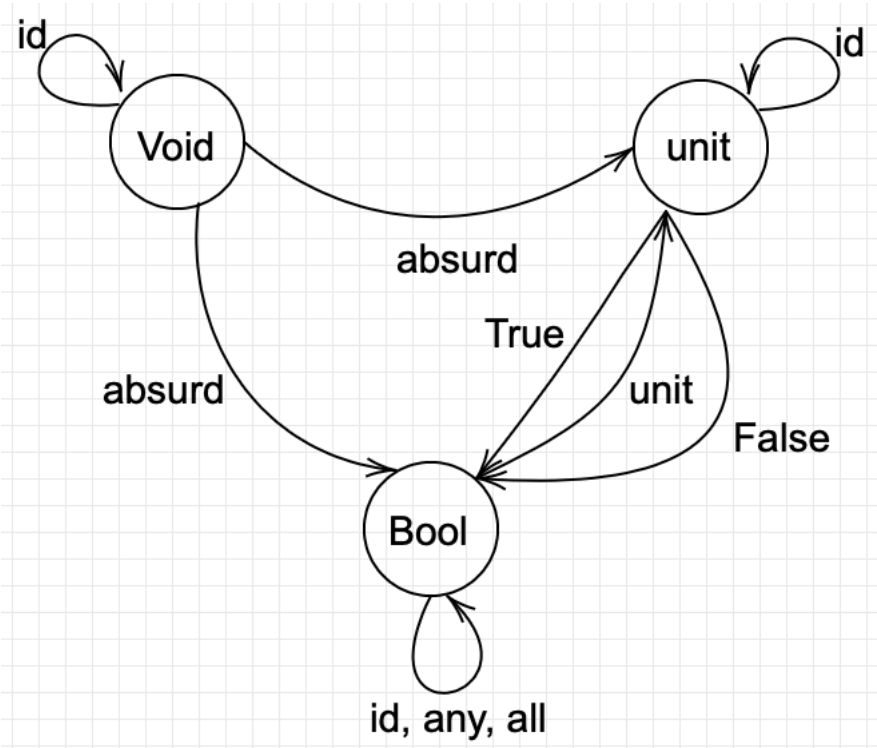

Figure 1: Answer to 2.7.6

3 Categories Great and Small

3-2 Simple Graphs

Free Category: Directed graphs

> Note: This subsection contains the answer for Challenge 1.4.6 on page 10

3-3 Orders

> [SSC], Chapter 1: Goes into far more details on orders than given here.

Preorder A set, whose elements can be ordered by morphisms according to $<$ and $=$. A more precise definition is given in [SSC] (Definition 1.25), which clarifies that preorder relations are binary [SSC] (Definition 1.8): a relation between the sets X and X, i.e. a subset $R \subseteq X \times X$. In other words, for a binary operation, the co- and domains are of the same set.

As booleans values `True` and `False` can be associated to 1 and 0, the boolean set is a preorder.

Partial Order (poset for short) A preorder that satisfies an additional condition: if $a \leq b$ and $b \leq a$ then $a = b$.

> [SSC], Definition 1.25: For a *preorder set*, if $a \leq b$ and $b \leq a$, we write $b \cong a$ and say a and b are equivalent.
>
> Remark 1.30: A *preorder* is a *partial order* if we additionally have that $b \cong a$ implies $b = a$.

Total Order :

Hom-set A set of morphisms from object a to object b in a category $\mathcal{C}$ is called a homset and is written as $\mathcal{C}(a, b)$ (or, sometimes, hom$\mathcal{C}(a, b)$).

3-4 3-5 Monoid as Set and Category

Monoid are ubiquitous in programming, showing up as `Char`, `List`, recursive data structures, futures in concurrent programming, and so on. They are categories, $\mathcal{C}$, with just one object, $* \in \mathcal{C}$, such that there is only one homset, hom$\mathcal{C}(*, *)$, which is the identity morphism, $\text{id}_* : * \to *$. Some refer to a monoid as a singleton category, which acts analogous to a singleton set (which was introduced in

Section 2-6), meaning that a *functor* (functors are introduced in Chapte 7) from a monoid to any other category selects an object in that category.

Monoidal categories have been used to formalise the use of networks in computation and reasoning—amongst others, applications include circuit diagrams, Markov processes, quantum computation, and dynamical systems.

2-6 Examples of Types

> [CH], Definition 1.2, page 10: A *monoid* as set $(M, e, \diamond)$ consists of:

> [PC], Definition 1.46, page 18: A *monoid* as category $(M, e, \diamond)$ consists of: - a single set M, called the *carrier*; - an element $e \in M$, called the *unit*; and - an associative function $\diamond : M \times M \to M$, called the (binary) *operation*.
>
> These are subject to two conditions: - unitality/identity: for any $m \in M$, we have $e \diamond m = m$ and $m \diamond e = m$, thus $e = \mathrm{id}_M$; - associativity: for any $l, m, n \in M$, we have $(l \diamond m) \diamond n = l \diamond (m \diamond n)$.

3-6 Challenges

1. Generate a free category from: - a) A graph with one node and no edges

However I am not quite sure about this answer, because [CM] speaks of interla morphisms on page 30.

- b) A graph with one node and one (directed) edge (Hint: this edge can be composed with itself)
- c) A graph with two nodes and a single arrow between them
- d) A graph with a single node and 26 arrows marked with the letters of the alphabet: a, b, c … z.

2. What kind of order is this? - a) A set of sets with the inclusion relation: A is included in B if every element of A is also an element of B.

Our morphisms are subset relations. Every set includes itself, $A \subseteq A$. Inclusion is also composable. $A \subseteq B$ and $B \subseteq C$ implies $A \subseteq C$. This means that we at least have a preorder. If $A \subseteq B$ and $B \subseteq A$ then $A = B$, which means that we at least have a partial order. Not all objects are a subset of each other though. For example {1} and {2,3} are not subsets of each other. This means we don't have a total order and only a partial order.

- b) C++ types with the following subtyping relation: T1 is a sub-type of T2 if a pointer to T1 can be passed to a function that expects a pointer to T2 without triggering a compilation error.

4 Kleisli Functions

It is quite strange that Kleisli functions are itroduced in the book before, e.g., isomorphisms or other concepts generally introduced beforehand in other books.

In typed functional programming, the Kleisli category is used to model call-by-value functions with side-effects and computation.

> Note: This section contains suggestions for Challenge 2.7.1

> [CH], page 75: Every *monad* defines a new category, called the *Kleisli category*.

4-2 Writer in Haskell

Here is an example on a possible application of a Kleisli category used for logging, memoization, or tracing the execution of functions. A Kleisli category for this purpose is also called *the writer monad*. We start by defining a costume type, which can be wrapped around functions and write something to a log file,

```haskell
type Writer a = (a, String)
```

Then we need some morphism to compose the `Writer` type with the functions we want to keep a log of, `f :: a -> Writer b`. We'll declare the composition as a funny infix operator (infix is, `a + b`, and prefix is `(+)a b`), sometimes called the "fish":

```haskell
(<=<) :: (b -> Writer c) -> (a -> Writer b) -> (a -> Writer c)
```

Note, that I changed the type signature here to that given in the book, as operators are left-associative (meaning they are applied left to right). We can implement the above signature as,

```haskell
m2 <=< m1 = \x ->
    let (y, s1) = m1 x
        (z, s2) = m2 y
    in (z, s2 ++ s1)
```

I don't understand the function above. The Kleisli identity morphism is commonly called *return*,

```haskell
return :: a -> Writer a
return x = (x, "")
```

Having defined the `Writer` we can wrap it around other functions, e.g., `toUpper` and `toWords` provided by Haskell,

```haskell
upCase :: String -> Writer String
upCase s = (map toUpper s, "upCase ")

toWords :: String -> Writer [String]
toWords s = (words s, "toWords ")
```

The composition of the two functions is accomplished with the help of the fish operator,

```haskell
process :: String -> Writer [String]
process = toWords <=< upCase
```

We have accomplished our goal: The aggregation of the log is no longer the concern of the individual functions. They produce their own mes- sages, which are then, externally, concatenated into a larger log.

4-3 Kleisli Categories

For our limited purposes, a Kleisli category has, as objects, the types of the underlying programming language.

"embellishment" corresponds to the notion of an *endofunctor* in a category… something like endomaps but for categories instead of objects in categories (we learn in Sections 7.1 that functors are mappings between categories).

4-4 Challenges

- a) Construct the Kleisli category for partial functions (define composition and identity), meaning a function `Optional` that can be wrapped around another functions that is not defined for all possible values of its argument.

In order to solve this exercise, [PC] Chapter 5 on *monads* (and Section 2.3.4 on *type constructors*) provides some helpful comments.

> [PC], Chapter 5, page 125: "Starting with the category of types and functions, it's possible to construct new categories that share the same objects (types), but redefine the morphisms and their composition."

> "A simple example is [that of partial functions]. These are computations that are not defined for all values of their arguments. We can model such computations using the `Maybe` data type".
>
> ```haskell
> data Maybe a = Just a | Nothing
> ```
>
> "A partial computation from `a` to `b` can be implemented as [...] `f :: a -> Maybe b`. When the partial computation succeeds, this function wraps its output in `Just`, other-wise it returns `Nothing`."

Thus, what we are trying is to create a new category that we will call `Optional(Maybe)`, a *Kleisli category*, which relates to the function it wraps around through an *endofunctor*.

As any introduction to Category Theory tells us, a category consists of ([CM], page 21): objects, morphisms, identities, and composition, which we need to define.

First, let's with the easiest first and declare the identity morphism

```haskell
identity :: a -> Maybe a
identity a = Just a
```

Secondly, we need to define a function that can compose two Kleisli morphisms, for which we use the 'fish' infix operator as as above (but in opposite direction). We define it's type signature to be:

```haskell
(<=<) :: (b -> Maybe c) -> (a -> Maybe b) -> (a -> Maybe c)
```

Here is a way we could implement this: if the first function returns `Nothing`, don't call the second function. Otherwise, call it with the contents of `Just`

```haskell
g <=< f = \a -> case f a of
                  Nothing -> Nothing
                  Just b -> g b
```

Now that we have define the contents of our category, we need to proof that it also satisfies the *identity law* and *associativity law*.

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

```haskell
safeDiv :: Float -> Float -> Maybe Float
safeDiv m n
        | n == 0    = Nothing
        | otherwise = Just (m / n)
```

- c) Compose the functions `safe_root` and `safe_reciprocal` to implement `safe_root_reciprocal` that calculates `sqrt(1/x)` whenever possible.

```haskell
safeSqrt :: Float -> Maybe Float
safeSqrt x
        | x >= 0    = Just (sqrt x)
        | otherwise = Nothing
```

```haskell
safeSqrtDiv :: Float -> Maybe Float
safeSqrtDiv = safeSqrt <=< safeDiv1
```

5 Products and Coproducts

Before jumping into the chapter let's make clear that a *universal property/construction* are unique up to isomorphism …

5-1 Initial Object & 5-2 Terminal Object

> [CH] Definition 1.3, page 13: An object $t \in \mathcal{C}$ is terminal if for all $a \in \mathcal{C}$ there is [a unique morphism $! : a \to t$]. Similarly, $i \in \mathcal{C}$ is the initial if there [a unique morphism $! : i \to a$] to all objects. Figure 5-1-1

Within the Haskell type system, $\mathbf{Hask}$, `Void` takes the place of the inital objects which maps to every other type through `absurd`, `absurd: Void -> a`, and unit `()` takes is the terminal object as everything can uniquly map onto it, `unit: a -> ()`. However, the terminal object does not need to be `()` but could be any singleton set.

Since this *unique* morphism exists *for all* objects in $\mathcal{C}$, we say that terminal objects have a *universal property* (more in depth explenation given in Section 9-1).

Many more notes on initial objects: - [SSC], Section 6.2.1, page 183 - [PC], Section 2.2.2, page 39 - [CH], Definition 1.3, page 13 and terminal objects: - [SSC], Section 3.5.1, page 108 - [PC], Section 2.2.1, page 37 - [CH], Definition 1.3, page 13 - [CM], Article IV.1, page 213 For an explaination of *uniqueness up to isomorphism* see: - https://www.math3ma.com/blog/up-to-isomorphism - https://en.wikipedia.org/wiki/Up_to

5-3 Duality

The duality w.r.t. categories describes the fact, that for any category $\mathcal{C}$ we can define the opposite category $\mathcal{C}^{\text{op}}$ just by reversing all it's morphisms $f : a \to b$ to $f^{\text{op}} : b \to a$ ([CM] denotes it as f^{-1}). It follows then that a terminal object is the initial object in the opposite category.

> This is related to *invertible morphisms*, for which you can find an indepth introduction in [CM] starting from Article II, page 40 (and continuing for a long time), which I find clearer than what is described in the next section on *isomorphisms*.

The table below lists some examples of duality. Most of them weren't introduced yet, so don't worry if you don't know what they mean. However, for *monomorphism* and *epimorphism* are introduced early on in other books so it might be good to mention that you find more information on them in CM Article II page 52f.

Element of $\mathcal{C}$	Element of $\mathcal{C}^{\text{op}}$
identity morphism; id_A	identity morphism; id_A
initial object	terminal object
monomorphism	epimorphism
product; $x \times y$	coproduct; $x + y$
monad	comonad
limits	colimits
cones	cocones

Table 5-3-1: Some examples of duality.

Note that monoidal categories, $\mathcal{C}(*)$, are equal to their opposite category, $\mathcal{C}^{\text{op}}$, as there is only one homset, hom$\mathcal{C}(*,*)$, which is the identity morphism, $\text{id}_* = \text{id}_*^{\text{op}}$.

5-4 Isomorphisms

> [CM], Section 4.2, page 61: If $f : A \to B$, an *inverse* for f is a morphism $g : B \to A$ satisfying both $g \circ f = 1_A$, and $f \circ g = 1_B$ If f has an inverse, we say f is an isomorphism, or invertible morphism.

Spelling this out in words, to say that two functions are equal up to isomorphism, $h(x) \cong k(x)$, means three things (see, e.g., [CM] page 92): - the domain of h equals the domain of k, - the codomain of h equals the codomain of k, and - for each x in the domain of h and k, we must have $h(x) = k(x)$.

5-5 Product

> [PC], Definition 2.19, page 40: Let x and y be objects in a category $\mathcal{C}$. A *product* of x and y consists of three things: - an object, denoted as $x \times y$, - a morphism $\pi_1 : x \times y \to x$, and - another morphism $\pi_2 : x \times y \to y$,
>
> with the following *universal property*: For any other such three things, i.e. for any object a and morphisms $f : a \to x$ and $g : a \to y$, there is a unique morphism $h : a \to x \times y$ such that the following diagram commutes:
>
> Figure 5-5-1
>
> Often we just refer to $x \times y$ as the product of x and y. We call the morphisms π_1 and π_2 *projection maps* and *h* is frequently denoted as $h = \langle f, g \rangle$.
>
> (This definition is basically the same as [SSC], Definition 3.71, page 110)

Note, that in the definition above the universal property is shown as a dashed line, which is quite common in diagrammatic notations (some also use dotted lines). In terms of Haskell, the cartesian product, $x \times y$, is a tuple `(x,y)`, and the two morphisms π_1 and π_2 are the function `fst (x,y)` and `snd (x,y)` which are both part of Prelude.

The important lesson here, which is useful for thinking about programming, is about solving functions as $h : a \to x \times y$: to compute a cartesian product $x \times y$ we can decompose it into a pair of functions that we can easily solve, $f : a \to x$ and $g : a \to y$. In other words, through the universal property of products we have a one-to-one correspondence between

$$\text{hom}\mathcal{C}(a, x \times y) \cong \text{hom}\mathcal{C}(a, x) \times \text{hom}\mathcal{C}(a, y)$$

,

which in Haskell means that `a -> (x,y)` is isomorphic to `(a -> x, a -> y)`.Such decomposition is a common theme in using universal properties for structuring programs. Its helpful to explicitly write down the isomorphism between the two types. In one direction, we have the function `tuple`,

```
tuple :: (c -> a, c -> b) -> (c -> (a, b))
tuple (f, g) = \c -> (f c, g c)
```

The standard library `Control.Arrow` provides this function as an infix operator denoted as `&&&`. In the other direction, we can define the function `untuple`

```
untuple :: (c -> (a, b)) -> (c -> a, c -> b)
untuple h = (\c -> fst (h c), \c -> snd (h c))
```

If a category $\mathcal{C}$ has a terminal object and cartesian products, then it has n-arity products for all n: the terminal object is like the 0-arity product, in other words the terminal object is a unit for the product

(arity is the number of arguments of a function). We would say that $\mathcal{C}$ has *all finite products* and we say that a category is a *cartesian category* if it has all finite products ([PC], page 43). That is, the terminal object 1 (the unit type `()` in Haskell) is a unit for products, such that for any object x in $\mathcal{C}$ we have $1 \times x \cong x \times 1 \cong x$.

Note: the triangles $\pi_1 \circ h = f$ and $\pi_2 \circ h = g$ are both *choice problems* as introduced in [CM] Article 2.2, page 45.

More notes in: - [PC], Section 2.2.3, page 40 - [SSC], Section 3.5.1, page 110

5-6 Coproduct

In the category of sets, the coproduct is the *disjoint union* of two sets. An element of the disjoint union of the two sets `a` and `b` is either an element of `a` or an element of `b`. If the two sets overlap, the disjoint union contains two copies of the common part.

> [PC], Definition 2.31, page 43: Let x and y be objects in a category $\mathcal{C}$. A *coproduct* of x and y is an object, denoted as $x + y$, together with two morphisms $i_1 : x \to x + y$ and $i_2 : y \to x + y$, such that for any object a and morphisms $f : x \to a$ and $g : y \to a$, there is a unique morphism $h : x + y \to a$ such that the following diagram commutes:
>
> Figure 5-6-1
>
> We call the morphisms i_1 and i_2 *inclusion maps*. We will frequently denote h by $h = [f, g]$, and call it the *copairing* of f and g.
>
> (This definition is basically the same as [SSC], Definition 6.11, page 185)

Similar to the lesson we learned about products, the coproduct is important for solving functions like $h : x + y \to a$: to compute it we can decompose it into a pair of functions that we can solve more easily, $f : x \to a$ and $g : y \to a$. In other words, through the universal property of coproducts we have a one-to-one correspondence between

$$\text{hom}\mathcal{C}(x + y, a) \cong \text{hom}\mathcal{C}(x, a) \times \text{hom}\mathcal{C}(y, a)$$

Explain the $\times$ instead of a $+$ in the equation above!!!

When reading any Haskell book, after fundamental data types – such as `Void` and `()` – have been explained, one is introduced to algebraic data type constructors that compose data types to give a new one (such as `Bool = {True, False}`). In this context, without really knowing, one uses coproducts which in computer science are more commonly called *sum types* which are discussed in Section 6-3.

Above learned that the terminal object is a unit for products. The dual for coproducts is, that the inital object 0 (`Void` in Haskell) is a unit for coproducts. That is, for any object x in a category with coproducts $0 + x \cong x + 0 \cong x$.

I also notice that the triangles $h \circ i_1 = f$ and $h \circ i_2 = g$ are Both *determination problems* as introduced in [CM] Article 2.2, page 45.

In Section 12 Bartosz will discus *limits* and *colimits* which are tightly connected to *products* and *coproducts* respectively. In that section, the relation between terminal and initial with product and coproduct will be made more explicit.

Many more notes in: - [PC], Section 2.2.4, page 43 - [SSC], Section 6.6.2, page 185 - [CM], Article IV.5, page 222

5-7 Asymmetry

We'll see later that product behaves like multiplication, with the terminal object playing the role of one; whereas coproduct behaves more like the sum, with the initial object playing the role of zero.

In the category of sets, an isomorphism is the same as a bijection.

5-8 Challenges

1. Show that the terminal object is unique up to unique isomorphism. This challenge asks us to proof the proposition known as the *uniqueness of the terminal object*

> [CM], Article IV.1, page 213 Proposition (Uniqueness of the Terminal Object): If S_1, S_2 are both terminal objects in the category $\mathcal{C}$, then there is exactly one $\mathcal{C}$-map $S_1 \to S_2$, and that map is a $\mathcal{C}$-isomorphism.

Proof: By definition, there is for each object in $\mathcal{C}$ exactly one morphism to the terminal object. Thus if we claim that S_1 and S_2 are both terminal objects of $\mathcal{C}$, then there must be exactly one morphism for $f : S_1 \to S_2$ and $g : S_2 \to S_1$.

This is wrong, Correct: By definition, the only morphism that has the terminal S object as the domain is it's own isomorphism 1_S.

This mean must must mean that $f \circ g = 1_{S_1}$ and $g \circ f = 1_{S_2}$ are the inverse of each other. Thus, the *uniqueness of the terminal object* is also related to the *uniqueness of inverses*.

2. What is a product of two objects in a poset? Hint: Use the universal construction. Thus our two objects x and y are in the *partial order* category, which uses $\leq$ as it's arrows. Drawing the product diagram helps to find the answer,

it is useful to know that divisibility relation defines partial ordering on positive integers

Thus, at the domains are objects smaller or equal to the codomain. Meaning that the product in a poset is a greatest lower bound.

A short answer is given in [PC], Section 2.2, Example 2.20, page 41. [CH] page 28 and [CH] Example 3.4 has the better answer

Look also at [SSC], Example 3.72, page 110 on products in **Set** to help make sense of it.

3. What is a coproduct of two objects in a poset? Reversing our reasoning to the previous question we conclude, that the coproduct is the smallest upper bound, the smallest positive integer that is a multiple of both a and b.

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

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. This problem formulates the dual of a similar problem outlined on page 62 in terms of product. Again, we use the two Haskell types, `Int` and `Bool`, and sampling for the best candidate for their coproduct.

To judge whether one object coproduct (`Either`) is "better" than another (`Int`), we need to check if the former can factorise the latter, i.e. if the latter has two injections `f` and `g` that are factorised by the injections of the former as `f = h . i_1` and `g = h . i_2`.

So let's check if there exists a mapping `h` from the coproduct `Either` to the coproduct `Int` through pattern matching,

```
h :: Either Int Bool -> Int
h (Left x) = x
h (Right x) = fromEnum x
```

6. Continuing the previous problem: How would you argue that `Int` with the two injections i and j cannot be "better" than Either? However, if we would try to find an inverse mapping

```
h' :: Int -> Either Int Bool
h' x = x
h' _ = True
```

we find that it is too small – it only coveres the `Int` dimension of the coproduct.

7. Still continuing: What about these injections?

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`.

6 Simple Algebraic Data Types

This section develops our understanding of *products* and *coproducts* (introduced in Sections 5.5 and 5.6) further in the context of data types.

6-1 Product Types

Product types are of the forme

```
data Pair a b = Pair a b
```

The `Pair` operator is synonymous with `(,)`.

6-2 Records

Programming with tuples and multi-argument constructors can get messy and error prone — keeping track of which component represents what. It’s often preferable to give names to components. A product type with named fields is called a record in Haskell.

Another example then the on given in the book is found in [RWH] Chapter 3, page 55. Here the data type `Customer` is the *coproduct* of objects/types `customerID` and `customerName`.

```
data Customer = Customer {
    customerID :: Int,
    customerName :: String,
}
```

Thus, the injection `h: customerID -> Customer` would be written as `costumer1 = Customer {customerID=1234, customerName=Adam}`. The product, or surjective morphism, through which we can access the separate parts is `customerID costumer1`.

6-3 Sum Types

To model finite coproducts, one uses *sum types* (the concepts of sum types, coproducts, and tagged union have their origin from different communities but their meaning are tightly related).

In Section 5.3 introduced the fact of categorical duality, meaning that given a category $\mathcal{C}$ we can invert all morphisms to obtain its the opposite category $\mathcal{C}^{\text{op}}$. Have learned about products and coproducts, we realy that they are each others dual. Hence, we can also understand sum types as being the dual of product types.

In Haskell, sum types $x + y$ are created using the constructor `x | y` as shown in Section 5.6.

It turns out that $\mathbf{Set}$ is also a (symmetric) monoidal category with respect to coproduct. The role of the binary operation is played by the disjoint sum, and the role of the unit element is played by the initial object. In terms of types, we have Either as the monoidal operator and Void, the uninhabited type, as its neutral element. You can think of Either as plus, and Void as zero.

> [RWH], page 44: A famous two-element set is the familiar `Bool`, is the simplest common example of a category of *algebraic data type*, meaning it can have more than one value constructor.
>
> ```
> 1 data Bool = True | False
> ```

Functions to Bool, `f: a->Bool`, are called *predicates*.

In Haskell we can implement coproducts using the type constructor | as

```
1 data Coproduct x y = Incl1 x | Incl2 y
```

however, in Haskell it is more common to write coproducts as

```
1 data Either x y = Left x | Right y
```

Thus, `Either` is the object $x + y$ in Figure 5-6-1 and the two morphisms $i_{1,2}$ from the input objects x and y are `Left :: a -> Either a b` and `Right :: b -> Either a b`.

Common sum types are `Bool`, `List`, `Maybe`, `Either`. Be aware however, that this is not the end of the story. In the following chapters new concepts are being introduced that will place, e.g., `Maybe` and `List` types into a new light!

6-4 Algebra of Types

We have the sum types with Void as the neutral element, and the product types with the unit type, (), as the neutral element.

Mathematicians have a name for two such intertwined monoids: it's called a semiring. It's not a full ring, because we can't define subtraction of types.

Numbers	Types
0	`Void`
1	`()`
$x + y$	`Either x y = Left x | Right y`
$x \times y$	`(a,b)` or `Pair a b`
$2 = 1 + 1$	`data Bool = True | False`
$1 + a$	`data Maybe = Nothing | Just a`

Table 6-4-1

Logic	Types
false	`Void`
true	`()`
x || y	`Either x y = Left x | Right y`
x && y	`(a,b)` or `Pair a b`

Table 6-4-2

From the universal properties of product and coproduct, follows the distributive property of binary operations, $\delta : (x \times z) + (y \times z) \to (x + y) \times z$ with $x, y, z \in \mathbf{Set}$. In terms of the Haskell category $\mathbf{Hask}$ this would be,

```haskell
distri :: Either (x, z) (y, z) -> (Either x y, z)
distri Left (x, z) = (Left x, z)
distri Right (y, z) = (Right y, z)
```

6-5 Challenges

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

Both `Either` and the Kleisli category `Maybe` are sum types,

```haskell
data Either () a = Left () | Right a
```

and

```haskell
data Maybe a = Just a | Nothing
```

Using the definition of isomorphisms (found in ,e.g., [CM] page 40), we need to show that the two mappings `f: Maybe a -> Either ()a` and `g: Either ()a -> Maybe a` are the inverse of each other, meaning `f . g = maybeID` and `g . f = eitherID`. The to mapping `f` and `g` are provided in Haskell in the `Data.Either.Combinators` module and are called `maybeToLeft`, `leftToMaybe`, `maybeToRight` and `rightToMaybe`.

One can implement `f` as

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

and `g` as

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

2. Skip

3. Skip

4. Skip

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

From Table 6-4-1 we can read that $a + a$ expressed in types is `Either a a`. And the relation $2 = 1 + 1$ is given by `Bool`, of which we need to multiply both sides with `a`. In Section 5.5 we learned that in the categorical context, multiplication is a metaphor for taking the set product. Thus we need to show that there are morphisms π_1 and π_2 from the set product `[(True, a), (False, a)]` to `Either a a` and `Bool`.

We can write π_1 as

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

and it's inverse as

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

7 Functors

While functions act on elements in objects within categories, functors act across categories. In so doing, functors from one category to another are structure-preserving maps of *objects and morphisms*.

Functors are a very powerful concept. For functional programming specifically they are what allow us to define translations between data types, type constructors, and containers. But more general, they help to translate between different branches of mathematics.

The way functors relate to each other is known as *natural transformations*.

> [SSC], Definition 3.28, page 91: Let $\mathcal{C}$ and $\mathcal{D}$ be categories. To specify a *functor* from $\mathcal{C}$ to $\mathcal{D}$, denoted $F : \mathcal{C} \to \mathcal{D}$, (i) for every object $c \in \mathrm{Ob}(\mathcal{C})$, one specifies an object $F(c) \in \mathrm{Ob}(\mathcal{D})$; (ii) for every morphism $f : c_1 \to c_2$ in $\mathcal{C}$, one specifies a morphism $F(f) : F(c_1) \to F(c_2)$ in $\mathcal{D}$.
>
> The above constituents must satisfy two properties, ‘the functor laws’: (a) Preserve identity: for every object $c \in \mathrm{Ob}(\mathcal{C})$, we have $F(\mathrm{id}_c) = \mathrm{id}_{F(c)}$; (b) Preserve composition: for every three objects $c_1, c_2, c_3 \in \mathrm{Ob}(\mathcal{C})$ and two morphisms $f \in \mathrm{hom}\mathcal{C}(c_1, c_2)$, $g \in \mathrm{hom}\mathcal{C}(c_2, c_3)$, in other words $f : c_1 \to c_2$ and $g : c_2 \to c_3$ respectively, the equation $Fg \circ Ff = F(g \circ f)$ holds.

There are a number of commonly used functors, which we are going to list here and explain in more detail later:

> [PC], Definition 3.2, page 67 Two categories $\mathcal{C}$ and $\mathcal{D}$ can be subjected to the following incomplete list of functors - A *identity functor* $\mathrm{id}_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ - A *constant/blackhole functor* $\Delta_M : \mathcal{C} \to \mathcal{M}$ for $m \in \mathcal{M}$, where $\mathcal{M}$ is a monoid containing a singleton set. - A *contravariant functor* $\mathcal{C} \to \mathcal{D}$ is a functor $\mathcal{C}^{\mathrm{op}} \to \mathcal{D}$. - An *endofunctor* on $\mathcal{C}$ is a functor $F : \mathcal{C} \to \mathcal{C}$ - A *bifunctor* on $\mathcal{C}$ is a functor $\mathcal{C} \times \mathcal{C} \to \mathcal{D}$ - A *profunctor* on $\mathcal{C}$ is a functor $\mathcal{C} \times \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$.

Many more notes in: - [PC], Section 3.2, page 66 - [CH], Section 1.2, page 11 - [SSC], Section 3.3.2, page 91

7-1 Functors in Programming

When programming in Haskell, we can’t escape its $\mathbf{Hask} \in \mathbf{Set}$ category and will therefore most often talk about *endofunctore*, functors that $\mathbf{Hask}$ to itself. Such functor maps types to some other types. Without mentioning it, some important Haskell functors were already indirectly dicussed: - `Maybe` functor:

```haskell
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
```

- `List` functor

```haskell
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap _ Nil = Nil
fmap f (Cons x t) = Cons (f x) (fmap f t)
```

- `Reader` functor

```haskell
fmap :: (a -> b) -> (r -> a) -> (r -> b)j
```

- `Writer` functor

Haskell's Prelude provides a `Functor` type class, capturing the general pattern:

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

such that we can rewrite the examples listed above to be an instance of this class. Other means of implementing functors are possible, but this is the most convenient way to do so. An common operation that one uses in many programming languages without referring to it as a functor is `map`:

```haskell
ghci: map (\x -> x > 3) [1..6]
[False,False,False,True,True,True]
```

which works just as our `Functor` typeclass

```haskell
ghci: fmap (\x -> x > 3) [1..6]
[False,False,False,True,True,True]
```

7-1-2 Equational Reasoning

Equational reasoning is at the heart of mathematics, theoretical computer science, and routine for any student of Haskell. It is through equational reasoning that one can check equality of functions in Haskell, while in languages such as Coq one can encode proofs to check equality as they are proof assistants. The process of interpreting code by subsititung equals-for-equals is known as equational reasoning. In the usual notions of equational reasoning one has a trinity of ideas: equations, Lawvere theories, and monads on $\mathbf{Set}$.

7-1-4 Typeclasses

Typeclasses are among the most powerful features in Haskell. They allow one to define generic interfaces that provide a common feature set over a wide variety of types. They are used to help model structures from category theory, including functors, profunctors, monads and many more.

For example, a type class for objects that support equality is defined as:

```haskell
class Eq a where
    (==) :: a -> a -> Bool
```

and a functor as

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

The lowercase `f` is a type variable, similar to type variables `a` and `b`, which Haskell’s compiler recognises as a type constructor from it’s context.

But note, that these concepts from category theory can’t be implement one-to-one in Haskell. Often only the most crucial types are implemented, while the laws data types must obey are not specified in code, but often written in the documentation. This guides usage of the type class. The more closely these structures obey these laws, the more reliable categorical insights are for reasoning about them, and so the more useful category theory is for writing code (for more explaination see [PC], Example 3.47, page 82).

An example of an endofunctor Δ_c, the ‘blackhole’ functor that maps every other category to a singleton is

```haskell
class Functor f where
    fmap :: (a -> b) -> Const c a -> Const c b

instance Functor (Const c) where
    fmap _ (Const v) = Const v
```

More on typeclasses in: - [RWH], Chapter 6, page 135 - [PC], Section 3.5, page 74 - [CH], Section 2.1, page 21

7-2 Functors as Containers

Containers are objects that can contain a type they are parameterised over. Common examples of ‘functional containers’ are `IO`, `Maybe`, `()` and `[]`. The important take-away is, that we can think of such classical container objects (such as `[]`) as functions, which implies that we can also think of functions and functors as containers.

7-3 Functor Composition

The result of `maybeTail` is of a type that's a composition of two functors, `Maybe` and `[]`, acting on a, as

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

7-4 Challenges

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.

So, instead of using `fmap` as defined on page 94, we tweak it a bit. As explained, we can use equational reasoning to proof the functor laws (identity and composition).

Let's start with the shorter one first, proving the identity law for both types in `Maybe`:

```
fmap id Nothing = Nothing      -- using definition of fmap
                = id Nothing   -- using definition of id

fmap id (Just x) = Nothing     -- using definition of fmap
                 = id Nothing  -- using definition of id
```

We see that for the 'Just x' is being mapped onto Nothing instead of to itself.

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

Remember the type signature of the reader functor is

```
class Functor f where
    fmap :: (a -> b) -> (r -> a) -> (r -> b)

instance Functor ((->) r) where
    fmap f g = f . g
```

Let's start again to prove the identity law, `fmap id = id`,

```
fmap id a = id . a
          = a
          = id a
```

and the composition law, `fmap (g . f)= fmap g . fmap f`,

```
fmap (g . f) a = (g . f) . a          -- using definition of fmap
               = g . (f . a)          -- using associative law
               = g . (fmap f a)       -- using definition of fmap
```

```
4                 = fmap g (fmap f a)     -- using definition of fmap
5                 = (fmap g . fmap f) a   -- using definition of
                      composition
```

3. Skip

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).

Remember the `List` category,

```
1 data List a = Nil | Cons a (List a)
```

and the `List Functor` mentioned above

```
1 instance Functor List where
2     fmap _ Nil = Nil
3     fmap f (Cons x t) = Cons (f x) (fmap f t)
```

Again, let’s start with the identities for both parts of of the `List` sum type

```
1 fmap id Nil = Nil
2             = id Nil
3
4 fmap id (Cons a as) = Cons (id a) (fmap id as)
5                     = Cons (id a) as
6                     = Cons a as
7                     = id (Cons a as)
```

Now that we have shown that the identity laws are preserved we move on to the compositin Laws

```
1 fmap (g . f) Nil = Nil
2                  = fmap g Nil
3                  = fmap g (fmap f Nil)
4                  = (fmap g . fmap f) Nil
5
6 fmap (g . f) (Cons a as) = Cons ((g . f) a) (fmap (g . f) as)     --
     def of fmap
7                          = Cons ((g . f) a) (fmap g (fmap f as))  --
                             induction
8                          = Cons (g (f a)) (fmap g (fmap f as))    --
                             def of comp
9                          = fmap g (Cons (f a) (fmap f as))        --
                             def of fmap
10                         = fmap g (fmap f (Cons a as))            --
                             def of fmap
11                         = (fmap g . fmap f) (Cons a as)           --
                             def of comp
```

8 Functoriality

8-1 Bifunctors

Just as one can take the *Cartesian product*, $\times$, for every pair of two objects within one category $\mathcal{A}$, we can also take a product of two categories $F : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$, where F is called a *bifunctor* as it has two arguments (instead of one as a functor has). The product of categories is an important application of bifunctors, but not the only one…

> [CH], Definition 3.7, page 34: Given two categories $\mathcal{A}, \mathcal{B}$ their *product category* $\mathcal{A} \times \mathcal{B}$ is given by: - The objects $a \in \mathcal{A}$ and $b \in \mathcal{B}$ the bifunctor F produces product-pairs (a, b). - The arrows $g : a \to a'$ in $\mathcal{A}$ and the arrows $h : b \to b'$ in $\mathcal{B}$ combine to pairs as $(g, h) : (a, b) \to (a', b')$. - The identity arrwos for (a, b) are the pair $(\mathrm{id}_a, \mathrm{id}_b)$. - Composition of arrows happens per component, i.e. when $g, g' \in \mathcal{A}$ and $h, h' \in \mathcal{B}$:
>
> $$(g, h) \circ (g', h') \equiv (g \circ g', h \circ h')$$
>
> [CH], Definition 3.8, page 34: Let $\mathcal{A}, \mathcal{B}, \mathcal{C}$ be categories. A *bifunctor* is a functor: $F : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$.

In Haskell, similar to functors, we implement bifunctors through a type class (which is provided in the `Data.Bifunctor` library):

```haskell
class Bifunctor f where
    bimap :: (a -> a') -> (b -> b') -> (f a b -> f a' b')
    bimap g h = first g . second h
    first :: (a -> a') -> f a b -> f a' b
    first g = bimap g id
    second :: (b -> b') -> f a b -> f a b'
    second h = bimap id h
```

with the bifunctor `f` being type constructor that takes two arguments. Notice that this is a circular definition, which means that a bifunctor can be implemented in Haskell by either: - only providing the `bimap` definition, - only the `first` and `second` functions (but sometimes this doesn't work because `first g . second h` and `first h . second g` may not commute), or - implement everything: `bimap`, `first`, and `second`.

This Haskell implementation of a bifunctor can be sketched as, Figure 8-1-1

Here, I have given different labels to the three categories involved in the bifunctor mapping. However, as Bartosz states on page 92, when implementing concepts from category theory in Haskell we are

bound by $\mathbf{Hask}$ such that we effectively always use endofunctors, meaning $\mathbf{Hask} \times \mathbf{Hask} \to \mathbf{Hask}$.

More notes in: - [PC], Section 3.4.3, page 83 - [CH], Section 3.3, page 34

8-2 Product and Coproduct Bifunctors

Thus, when we have a morphisms in $\mathcal{A}$ as $f : a \to a'$ and a different morphism in $\mathcal{B}$ as $g : b \to b'$ then the Cartesian product needs to be implements in Haskell as an instance of the `Bifunctor` class as

```
instance Bifunctor (,) where
    bimap f g (a, b) = (f a, g b)
```

This is the instance of pair production, the simplest product type. This instance of the `Bifunctor` makes pairs of types, `(,)a b = (a,b)`.

Above we have introduced the bifunctor through it's application to form products of categories. But we can of course by duality also use bifunctors to create coproducts of categories.

In In Haskell, this is exemplified by the `Either` type constructor being an instance of `Bifunctor`:

```
instance Bifunctor Either where
    -- this instance gives bimap the type signature:
    -- bimap :: (a -> a') -> (b -> b') -> (Either a b -> Either a' b')
    bimap f _ (Left a) = Left (f a)
    bimap _ g (Right b) = Right (g b)
```

Example: Bifunctor as product

```
bimap (+1) (*3) (2, 3) = (3, 9)
```

Example: Bifunctor as coproduct

```
bimap (+1) (*3) (Left 3)  = 4
bimap (+1) (*3) (Right 3) = 9
```

8-3 Functorial Algebraic Data Types

There are two essential building blocks of parameterised algebraic data types: - items independent of the parameter type of a functor, like `Nothing` in `Maybe`, `Nil` in `List`, or the identiy functor. - containers, like `List` or `Tuple`, that encapsulate the independent parameter type named above

Everything else that makes a programming language more expressive is constructed from these two primitives using, - products, $a \times b$, `(a,b)`, and - coproducts, $a + b$, `Left a | Right b`, also called *sum types* (introduced in Section 6-3) or *exponentials* (more on them in Section 9-3).

Types, that are composed out of these primitives are called *algebraic data structures*, and the operation that composes them is the *algebra* of types.

Composition of a bifunctor with two functors in Haskell and treat it as a composition `BiComp` of two functor `fu` and `gu`.

```
newtype BiComp bf fu gu a b = BiComp (bf (fu a) (gu b))
```

```
instance (Bifunctor bf, Functor fu, Functor gu) =>
    Bifunctor (BiComp bf fu gu) where
        bimap :: (fu a -> fu a') -> (gu b -> gu b') -> bf (fu a) (gu b)
            -> bf (fu a') (gu b')
        bimap f1 f2 (BiComp x) = BiComp ((bimap (fmap f1) (fmap f2)) x)
```

Placing

```
{-# LANGUAGE DeriveFunctor #-}
```

at the top of your .hs file will compile functors with the additionally deriving Functor to your data structure

```
data Maybe a = Nothing | Just a deriving Functor
```

8-5 The `Writer` Functor

Advancing the `Write` introduced in Section 4.2

8-6 Covariant and Contravariant Functors

A mapping of categories that inverts the direction of morphisms in this manner is called a *contravariant functor*, $F^{\text{op}} : \mathcal{C}^{\text{op}} \to \mathcal{D}$, otherwise they are called *covariant functors*, $F : \mathcal{C} \to \mathcal{D}$.

```
class Contravariant f where
    contramap :: (b -> a) -> (f a -> f b)
```

8-7 Profunctors

> nLab: If $\mathcal{C}$ and $\mathcal{D}$ are categories, then a profunctor from $\mathcal{C}$ to $\mathcal{D}$ is a functor of the form $H_F : \mathcal{C} \times \mathcal{D}^{\text{op}} \to \mathbf{Set}$, which can also be written as $F : \mathcal{C} \nrightarrow \mathcal{D}$.

A endo-profunctor is $H_F : \mathcal{C} \times \mathcal{C}^{\text{op}} \to \mathbf{Set}$, which generalizes hom-functors hom$\mathcal{C} : \mathcal{C} \times \mathcal{C}^{\text{op}} \to \mathbf{Set}$, which will be discussed more in the next section.

Profunctors can generalize functors.

The Haskell implementation of profunctors can be found in `Data.Profunctor`,

```haskell
class Profunctor p where
    dimap :: (a' -> a) -> (b -> b') -> p a b -> p a' b'
    dimap g h = lmap g . rmap h
    lmap :: (a' -> a) -> p a b -> p a' b
    lmap g = dimap g id
    rmap :: (b -> b') -> p a b -> p a' b'
    rmap h = dimap id h
```

making it a bifunctor that is contravariant in the first argument, and covariant in the second.

More can be read in: - [CH], Section 10.1, page 107 - [PC], Section 3.4.4, page 86

8-8 The Hom-Functor

As you can see, the hom-functor is a special case of a profunctor.

8.9 Challenges

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.

As Bartosz mentioned earlier: “you may even view the built-in `Pair` type as a variation on this kind of declaration, where the name `Pair` is replaced with the binary operator `(,)`.” An example implementation of `(,)` was given on page 117.

To implement this bifunctor we have three options: - i) implementing only `bimap` and accepting the defaults for `first` and `second haskell instance Bifunctor Pair where bimap g h (Pair a b)= Pair (g a)(h b)` * Identity law * Associative law - ii) implementing `first` and `second` and accepting the default of `bimap haskell instance Bifunctor Pair where first g (Pair a b)= Pais (g a)b second h (Pair a b)= Pair a (h b)` * Identity law * Associative law - iii) implementing all three: `bimap`, `first`, and `second haskell instance Bifunctor Pair where bimap g h (Pair a b)= Pair (g a)(h b)first g (Pair a b )= Pais (g a)b second h (Pair a b)= Pair a (h b)` * Identity laws “‘haskell bimap ida idb (Pair a b) = Pair (ida a) (idb b) = Pair a b

```haskell
    first ida (Pair a b) = Pair (ida a) b
                         = Pair a b

    second idb (Pair a b) = Pair a (idb b)
```

```
                              = Pair a b
    ```
* Composition laws
    ```haskell
    bimap (g . h) (g' . h') (Pair a b) = Pair ((g . g') a) ((h . h') b)
                                       = Pair (g (g' a)) (h (h' b))
                                       = bimap g h (Pair (g' a) (h' b))
                                       = bimap g h (bmap g' h' (Pair a
                                           b)
                                       = (bimap g h) . (bimap g' h') (
                                           Pair a b)

    first (g . g') (Pair a b) = Pais ((g . g') a) b
                              = Pair (g (g' a)) b
                              = first g (Pair (g' a) b)
                              = first g (first g' (Pair a b))
                              = (first g) . (first g') (Pair a b)

    second (h . h') (Pair a b) = Pais a ((h . h') b)
                               = Pair a (h (h' b))
                               = second h (Pair a (h' b))
                               = second h (second h' (Pair a b))
                               = (second h) . (second h') (Pair a b)
    ```
```

We could have also, as Bartosz mentioned, considered the `first` and `second` case seperatly, but he also warned that "in general, separate functoriality is not enough to prove joint functoriality."

**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.**

This exercise is an extension of Exercise 2 in Section 6-5.

Even though `Const` is a bifunctor, we implement it as a functor since it's always partially applied

```haskell
data Const c x = Const c

fmap :: (a -> b) -> Const c a -> Const c b
fmap _ (Const v) = Const v
```

also remember the other types needed for this exercise,

```haskell
data Identity x = Identity x
data Either x y = Left x | Right y
data Maybe x = Just x | Nothing
```

Thus we can implement `maybeToEither: Maybe -> Maybe'` and `eitherToMaybe: Maybe'-> Maybe` as
```

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

eitherToMaybe :: Either () a -> Maybe a
eitherToMaybe (Left (Const ())) = Nothing
eitherToMaybe (Right (Identity x)) = Just x
```

To prove the isomorphism we have to show that `maybeToEither . eitherToMaybe = maybeID` and `eitherToMaybe . maybeToEither = eitherID`, meaning we should obtain what we put in.

```haskell
(maybeToEither . eitherToMaybe) (Left (Const ())) = maybeToEither
   Nothing
                                                   = Left (Const ())

(maybeToEither . eitherToMaybe) (Right y) = (maybeToEither .
   eitherToMaybe) (Right (Identity y))
                                          = maybeToEither (Just y)
                                          = Right (Identity y)
                                          = Right y

(eitherToMaybe . maybeToEither) (Just x) = eitherToMaybe (Right (
   Identity x))
                                         = Just x

(eitherToMaybe . maybeToEither) Nothing = eitherToMaybe (Left (Const ()
   ))
                                        = Nothing
```

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 a bifunctor.

First we must implement `PreList` as an instance of `Bifunctor`, which I do through `bimap` and accept the defaults of `first` and `second`,

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

To show that this `Bifunctor` instance of `PreList` behaves like a bifunctor, we need to show it fulfils the bifunctor identity and composition laws.

- Identity laws: "'haskell bimap ida idb Nil = Nil

  bimap ida idb (Cons a b) = Cons (ida a) (idb b) = Cons a b "'

- Composition laws: "'haskell bimap (g . h) (g' . h') Nil = Nil = bimap g h (bimap g' h' Nil) = (bimap g h) . (bimap g' h') Nil

  bimap (g . h) (g' . h') (Cons a b) = Cons ((g . g') a) ((h . h') b) = Cons (g (g' a)) (h (h' b)) = bimap g h (Cons (g' a) (h' b)) = bimap g h (bimap g' h' (Cons a b)) = (bimap g h) . (bimap g' h') (Cons a b) "'

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'.**

5. Skip

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?**

9 Function Types

Function types can also be seen as existing in sets, such that hom$\mathcal{C}(a, b) \in \mathbf{Set}$. Since we can talk of functions as being sets themselves, we can reinterpret $z(a)$ as a product of two sets $z \times a$, and instead of writing $z : a \to b$ we have another function g that acts on the product as $g : z \times a \to b$.

Haskell is build around this conception and you can read more about the practical side in [RWH] Section "Functions Are Data, Too" page 303.

9-1 Universal Construction

We have already come across the most important universal mapping properties in Section 5 and it is very useful to remember them, - Isomorphism, Section 5-4 - Initial object, Section 5-1 - Terminal object, Section 5-2 - Products, Section 5-5 - Coproducts, Section 5-6 - Exponential object, Section 9-3 - Cones and Cocones, Section 12

Since these universal properties are so central to type and category theory, corresponding constructions are built into Haskell's Prelude.

In this section, the important lesson is that functions between two sets, $z : a \to b$, can be understood as another set living in the category of Haskell types, $\mathbf{Hask}$. This perspective uses three objects: - function type, $z =$ hom$\mathcal{C}(a, b)$, which is an object containing a function f - argument type, a, which is an object containing an element x, and - result type, b, which is an object containing an element, $f(x)$

which are composed through *function application*/*evaluation*. As mentioned in the introduction to this chapter, since $z : a \to b$ can be thought of a set of morphisms z projecting a set a to set b, we can rewrite it using the *functoriality of the product*, $z \times a \to b$. However, similar to the product of sets, there can be multiple morphisms, e.g., $z \times a \to b$ and $z' \times a \to b$ related by $\lambda g \times \text{id}_a$. However, only one is the 'ideal' morphism, the from an initial object denoted as $(a \implies b) \times a$ to b, Figure 9-1-1

where $\text{eval}_b^a : ((a \implies b) \times a) \to b$ and $\lambda g : z \to (a \implies b)$ factorise $g = \text{eval}_b^a \circ (\lambda g \times \text{id}_a)$. Regarding λg, notice that it returns a function, thus making it a higher-order function.

Universal properties are typically indicated through dashed lines in diagrams.

$$\texttt{eval}_B^A : B^A \times A \to B$$

More notes on initial objects: - [PC], Chapter 2, page 35 - Section 2.4.2, page 57: on function evaluation

9-2 Currying

Continuing from where we left off in the last section, the relation between g and λg shows, that the universal property establishes a one-to-one correspondence between functions of two variables and functions of one variable returning functions. This correspondence is called currying, and λg is called the curried version of g. This lets us interpret functions as data, which can be passed from and into other functions. This concept is essential in functional programming and very useful in general.

> [CH], Definition 5.1, page 51: Let Z, A, B be sets. We define $(A \to B)$ to be the set of functions between A and B. Given a function of two variables: $g : Z \times A \to B$, we have a function: $\lambda g : Z \to (A \implies B)$, defined by $\lambda g(z)(a) = g(z, a)$. We say that λg is the curried version of g, and going from g to λg is called *currying*. Going the other way around is called *uncurrying*.

More notes on exponentials: - [PC], Section 2.4, page 56

9-3 Exponentials

The function type, $z = \text{hom}\mathcal{C}(a, b)$, which can be mapped to $(a \implies b)$ is often called the *exponential*, denoted as b^a. Just as products satisfy the universal property (see Section 5-5)

$$\text{hom}\mathcal{C}(a, x \times y) \cong \text{hom}\mathcal{C}(a, x) \times \text{hom}\mathcal{C}(a, y)$$

,

and coproducts satisfy the universal property (see Section 5-6)

$$\hom\mathcal{C}(x + y, a) \cong \hom\mathcal{C}(x, a) \times \hom\mathcal{C}(y, a)$$

.

exponential satisfy the universal property (using Definition 5.1 from the previous Section)

$$\hom\mathcal{C}(x \times y, a) \cong \hom\mathcal{C}(x, a^y)$$

.

> [PC], Definition 2.57, page 58: Let $A, B \in \mathcal{C}$ be objects in a cartesian category. An object B^A, equipped with a morphism $\text{eval}_{A,B} : B^A \times A \to B$, is called an *exponential* or *function object/type* for morphisms A to B if it has the following universal property: - for any object X and morphism $g : Z \times A \to B$, there exists a unique map $g' : Z \to B^A$ such that the diagram in Figure 9.1.1 commutes.

Note that what is commonly referred to as *function types* in computer science, is commonly called *exponentials* in category theory; see [PC] Exercise 1.14 and [CM] page ??? for why.

9-4 Cartesian Closed Categories

> [CH], Definition 5.2, page 52: *Cartesian closed category* (CCC): A category $\mathcal{C}$ is called *cartesian closed category*, if the following conditions are satisfied: - it has a *terminal object*, - for each pair $a, b \in \mathcal{C}$ there exists a *product* $a \times b$, and - for each pair $a, b \in \mathcal{C}$ there exists an *exponential*, b^a.

Having explained all previous concepts using the $\mathbf{Set}$ category, we know that it is a prominent example of a cartesian closed category.

A cartesian closed category, that additionally contains initial objects and coproducts is called bicartesian closed category.

Read more in: - [PC], Section 2.4, page 56 - [CH], Chapter 5, page 51 - [SSC], Section 7.2.1, page 225

9-5 Exponentials and Algebraic Data Types

- Zeroth Power:
$$a^0 = 1$$
- Powers of One:
$$1^a = 1$$

- First Power:
$$a^1 = a$$
- Exponentials of Sums:
$$a^{b+c} = a^b \times a^c$$
- Exponentials of Exponentials:
$$(a^b)^c = a^{b+c}$$
- Exponentials over Products:
$$(a \times b)^c = a^c \times b^c$$

9-6 Curry-Howard Isomorphism

10 Natural Transformations

We talked about functors as a means of lifting functions over structure so that we may transform only the contents, leaving the structure alone (`fmap :: (a -> b)-> f a -> f b`). What if we wanted to transform only the structure and leave the type argument to that structure or type constructor alone? With this, we've arrived at natural transformations!

> [CH], Definition 1.10, page 15: Let $\mathcal{C}$ and $\mathcal{D}$ be categories, and let $F, G : \mathcal{C} \to \mathcal{D}$ be two functors. A *natural transformation* α from $\mathcal{C}$ to $\mathcal{D}$, is a set of morphisms: $\alpha = \{\alpha_a : Fa \to Ga \,|\, a \in \mathrm{Ob}(\mathcal{C})\}$, indexed by objects in $\mathcal{C}$, so that for all morphisms $f : a \to b$ the diagram: Figure 10-0-1
>
> commutes. The red square diagram inside $\mathcal{D}$ is called the *naturality square* or *naturality condition*, $Gf \circ \alpha_a = \alpha_b \circ Ff$. We write $\alpha : F \implies G$, and call α_a *the component of* α *at* a.

As mentioned earlier, we can't implement concepts from category theory one-to-one in Haskell. Thus, also for the natural condition we only define the category of functors, one for every type, since Haskell can't proof the naturality condition.

In Haskell, one kind of natural transformation is implement in the `natural-transformation` package, which transforms a container type `f a` into another container `g a`,

```haskell
{-# LANGUAGE RankNTypes #-}

type natTrans f g = forall a . f a -> g a
```

Read more in: - [PC], Section 3.5, page 88 - [CH], Section 1.4, page 15 - [SSC], Section 3.3.4, page 95

10-1 Polymorphic Functions

Natural transformation are secretly at the heart of polymorphic functions. Infact, a natural transformation is a polymorphic function.

A first example of a natural transformation is the Exercises 4.1.1 in [RWH] on list operations, such as `safeHead` which is a function polymorphic in `a`,

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

which needs to satisfy the naturality condition

```
fmap f . safeHead = safeHead . fmap f
```

10-2 Beyond Naturalitys

Since all standard algebraic data types are functors, any polymorphic function between such types is a natural transformation.

10-3 Functor Category

As mentioned in the introduction to this chapter, just as functions form a set, functors form a category.

The cool thing is, just as the exponential $z = \text{hom}\mathcal{C}(a, b)$ is a set of morphisms between two sets in the same category, do the functors from $\mathcal{C} \implies \mathcal{D}$ form a category. And we can compose those categories of functor! Let $\alpha : \mathcal{C} \implies \mathcal{D}$ and $\beta : \mathcal{D} \implies \mathcal{E}$, then we have $\alpha \circ \beta : \mathcal{C} \implies \mathcal{E}$, defined by $(\alpha_a \circ \beta_a) = \alpha_a \circ \beta_a$.

So let's recap what a category is made up of: - objects - maps - for each map f, one object as domain of f and one object as codomain of f - for each object A an identity map, which has domain A and codomain A - composition of maps

with the following rules: - identity laws - associativity laws

10-4 2-Categories

This richer structure is an example of a 2-category, a generalization of a category where, besides objects and morphisms (which might be called 1-morphisms in this context), there are also 2-morphisms, which are morphisms between morphisms.

> nLab: The notion of a 2-category generalizes that of category: a 2-category is a higher category, where on top of the objects and morphisms, there are also 2-morphisms.
>
> A 2-category consists of - (small) categories as objects; - functors as 1-morphisms between objects; - natural transformations as 2-morphisms between morphisms.

No reading among my references on this topic.

10-5 Conclusion

10-6 Challenges

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

So this is the opposite way around than was given as an example.

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

To prove the naturality condition through equational reasoning we have two cases to consider. Firstly, `Nothing`:

```haskell
fmap f . alpha Nothing = fmap f Nil
                       = Nil
                       = alpha Nothing
                       = alpha . fmap f Nothing
```

Secondly, `Just x`:

```haskell
fmap f . alpha (Just x) = fmap f (Cons a (List a))
                        = f (Cons a (List a))
                        = alpha (Just (f x))
                        = alpha . fmap f (Just x)
```

Thus is both cases we have shown that `fmap f . alpha = alpha . fmap f`.

2. Define at least two different natural transformations between `Reader ()` and the list functor. How many different lists of `()` are there? Remember the `Reader` functor from Sections 7.1, which is implemented in Haskell as:

```haskell
instance Functor ((->) r) where
    fmap f g = f . g
```

However, in this chapter the `Reader` functor was implemented differently and we use the following definition

```
newtype Reader e a = Reader (e -> a)

instance Functor (Reader e) where
    fmap f (Reader g) = Reader (\x -> f (g x))
```

, which actually is pretty much the same as the original definition…

Thus, we are aked to find two different netural transformations between `Reader ()-> List a`, where the `List` container was defined as

```
data List a = Nil | Cons a (List a)
```

Thus, the natural transformation α is of the form

```
alpha :: Reader () a -> List a
```

As Bartosz mentioned on page 165 "There are only two of these, `dumb` and `obvious`",

```
dumb :: Reader () a -> List a
dumb (Reader _) = Nil
```

Verifying naturality condition through equational reasoning again

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

And for the other one

```
obvious :: Reader () a -> List a
obvious (Reader g) = Cons g a (List g a)
```

Verifying naturality condition through equational reasoning again

```
                              -- definition of obvious
fmap f . obvious (Reader g) = fmap f (Cons g a (List g a))
                              -- definition of
                              = Cons (f . g) a (List (f . g) a)
                              -- definition of
                              = obvious (Reader (f . g))
                              -- definition of
                              = obvious . fmap f (Reader g)
```

But besides these, we can also have other transformations such as

```
double :: Reader () a -> [a]
```

```haskell
double (Reader g) = [g (), g ()]
```

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

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

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

6. Create a few test cases for the opposite naturality condition of transformations between different `Op` functors. Here’s one choice:

```haskell
op :: Op Bool Int
op = Op (\x -> x > 0)
```

and

```haskell
f :: String -> Int
f x = read x
```

References

[CM] ‘Conceptual Mathematics’ by F. William Lawvere and Stephen H. Schanuel

[SSC] ‘An Invitation to Applied Category Theory: Seven Sketches in Compositionality’ by Brendan Fong and David I. Spivak

[PC] ‘Programming with Categories’ by Brendan Fong, Bartosz Milewski, David I. Spivak

[CH] https://github.com/jwbuurlage/category-theory-programmers by Jan-Willem Buurlage

[RWH] ‘Real World Haskell’ by Bryan O’Sullivan, Don Stewart, John Goerzen; 2008

[HPFP] ‘Haskell Programming from First Principles’ by Christopher Allen and Julie Moronuki; 2016

[WIWIK] ‘What I Wish I Knew When Learning Haskell’ by Stephen Diehl; 2020