Category Theory for Programmers

Jan-Willem Buurlage

November 21, 2016

This document contains notes for a small-scale seminar on category theory in the context of (functional) programming, organized at CWI. The goal of the seminar is to gain familiarity with concepts of category theory that apply (in a broad sense) to the field of functional programming. It could be an idea to have an associated (toy) project that examplifies the concepts that are discussed.

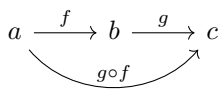
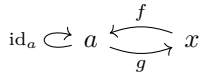
Although the main focus will be on the mathematics, examples should be made in Haskell to illustrate how to apply the concepts, and possibly examples in other languages as well (such as Python and C++).

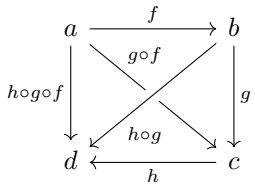
Chapter 1

Categories

1.1 Core definitions

We start with giving the definition of a category:

> **Definition**: A *category* $\mathcal{C} = (O, A)$ is a set O of *objects* and A of *arrows* between these objects, along with a notion of *composition* $\circ$ *of arrows* and a notion of an identity arrow id_a for each object $a \in O$.
>
> The composition operation and identity arrow should satisfy the following laws:
>
> - *Composition*: If $f : a \to b$ and $g : b \to c$ then $g \circ f : a \to c$.
>
> 
>
>
> - *Composition with identity arrows*: If $f : x \to a$ and $g : a \to x$ where x is arbitrary, then:
>
> $$\mathrm{id}_a \circ f = f, \ g \circ \mathrm{id}_a = g.$$
>
> 
>
>
> - *Associativity*: If $f : a \to b$, $g : b \to c$ and $h : c \to d$ then:
>
> $$(h \circ g) \circ f = h \circ (g \circ f).$$
>
> This is the same as saying that the following diagram commutes:
>
> Saying a diagram commutes means that for all pairs of vertices a' and b' all paths from between them are equivalent (i.e. correspond to the same arrow of the category).



If $f : a \to b$, then we say that a is the *domain* and b is the *codomain* of b. It is also written as:

$$\mathrm{dom}(f) = a,\ \mathrm{cod}(f) = b.$$

The composition $g \circ f$ is only defined on arrows f and g if the domain of g is equal to the codomain of f.

We will write for objects and arrows respectively simply $a \in \mathcal{C}$ and $f \in \mathcal{C}$, instead of $a \in O$ and $f \in A$.

Examples:

Some examples of familiar categories:

Name	Objects	Arrows
Set	sets	maps
Top	topological spaces	continuous functions
Vect	vector spaces	linear transformations
Grp	groups	group homomorphisms

In all these cases, arrows correspond to functions, although this is by no means required. All these categories correspond to objects from mathematics, along with *structure preserving maps*. **Set** will also play a role when we discuss the category **Hask** when we start talking about concrete applications to Haskell.

There are also a number of very simple examples of categories:

- **0**, the empty category $O \equiv A \equiv \emptyset$.
- **1**, the category with a single element and (identity) arrow:

$$\mathrm{id}_a \circlearrowright a$$

- **2**, the category with a two elements and a single arrow between these elements

$$\mathrm{id}_a \circlearrowright a \xrightarrow{f} b \circlearrowright \mathrm{id}_b$$

Another example of a category is a *monoid*, which is a specific kind of category with a single object. A monoid is a set M with a associative binary operation $\cdot : S \times S \to S$ and a unit element (indeed, a group without necessarily having inverse elements, or a *semi-group with unit*).

This corresponds to a category $\mathcal{C}(M)$ where:

- There is a single object (for which we simply write M)
- There are arrows $s : M \to M$ for each element $s \in M$.
- Composition is given by the binary operation of the monoid: $s_1 \circ s_2 \equiv s_1 \cdot s_2$.

1.2 Functors

A functor is a map between categories. This means it sends objects to objects, and arrows to arrows.

> **Definition**: A *functor* T between categories $\mathcal{C}$ and $\mathcal{D}$ consists of two functions (both denoted simply by T):
>
> - An *object function* that maps objects $a \in \mathcal{C}$: $a \mapsto Ta \in \mathcal{D}$
> - An *arrow function* that assigns to each arrow $f : a \to b$ in $\mathcal{C}$ an arrow $Tf : Ta \to Tb$ in $\mathcal{D}$, such that:
>
> $$T(\mathrm{id}_a) = \mathrm{id}_{Ta},\ T(g \circ f) = Tg \circ Tf.$$

A functor is a very powerful concept, since intuitively it allows you to translate between different branches of mathematics! They also play an important role in functional programming. Where among many other things, they are useful for defining the type of *containers*.

Functors can be composed, and this allows one to define a category of categories[1], where the arrows are functors.

Examples

- The identity functor: $\mathrm{id}_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ is defined as:

$$\begin{aligned} \mathrm{id}_{\mathcal{C}} :\ & a \mapsto a \\ & f \mapsto f \end{aligned}$$

- The constant functor $\Delta_d : \mathcal{C} \to \mathcal{D}$ for fixed $d \in \mathcal{D}$:

$$\begin{aligned} \Delta_d :\ & a \mapsto d \\ & f \mapsto \mathrm{id}_d \end{aligned}$$

- The 'power-set functor': $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ sends subsets to their image under maps. Let $A, B \in \mathbf{Set}$, $f : A \to B$ and $S \subset A$:

$$\begin{aligned} &\mathcal{P}A = \mathcal{P}(A), \\ &\mathcal{P}f : \mathcal{P}(A) \to \mathcal{P}(B),\ S \mapsto f(S) \end{aligned}$$

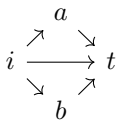
1.3 Special objects, arrows and functors

Special objects

For objects, we distuinguish two special kinds:

[1] Actually, there is some technicalities to be worked out and the resulting category consists of 'small categories' only.

Definition 1. An object $x \in \mathcal{C}$ is **terminal** if for all $a \in \mathcal{C}$ there is exactly one arrow $a \to x$. Similarly, it is **initial** if there is exactly one arrow $x \to a$ to all objects.



Here, i is initial, and t is terminal.

Special arrows

There are a number of special kind of arrows:

Definition 2. An arrow $f : a \to b \in \mathcal{C}$ is a **monomorphism** (or simply mono), if for all objects x and all arrows $g, h : x \to a$ and $g \neq h$ we have:

$$g \circ f \neq h \circ f.$$

To put this into perspective, we show that in the category **Set** monomorphisms correspond to injective functions;

Theorem 1. In **Set** a map f is mono if and only if it is an injection.

Proof. Let $f : A \to B$. Suppose f is injective, and let $g, h : X \to A$. If $g \neq h$, then $g(x) \neq h(x)$ for some x. But since f is injective, we have $f(g(x)) \neq f(h(x))$, and hence $h \circ f \neq h \circ f$, thus f is mono.

For the contrary, suppose f is mono. Let $\{*\}$ be the set with a single element. Then for $x \in A$ we have an arrow $\{*\} \to A$ corresponding to the constant function $\tilde{x}(*) = x$, then $f \circ \tilde{x}(*) = f(x)$. Let $x \neq y$. Since f is mono, $(f \circ \tilde{x})(*) \neq (f \circ \tilde{y})(*)$, and hence $f(x) \neq f(y)$, thus f is an injection. □

There is also an generalization of the notion of *surjections*.

Definition 3. An arrow $f : a \to b \in \mathcal{C}$ is a **epimorphism** (or simply epi), if for all objects x and all arrows $g, h : b \to x$ we have:

$$g \circ f = h \circ f \implies g = h.$$

Finally, we introduce the notion of an 'invertible arrow'.

Definition 4. An arrow $f : a \to b \in \mathcal{C}$ is an **isomorphism** if there exists an arrow $g : b \to a$ so that:

$$g \circ f = \mathrm{id}_a \quad \text{and} \quad f \circ g = \mathrm{id}_b.$$

(Introduce split, sections, retractions here? It is already a lot). In set, epi and mono imply iso. This however does not hold for general categories!

Special functors

Finally, we turn our attention to special kinds of functors. For this we first introduce the notion of a *hom-set* of a and b, the set[2] of all arrows from a to b:

$$\mathrm{Hom}_{\mathcal{C}}(a,b) = \{f \in \mathcal{C} \mid f : a \to b\}.$$

Definition 5. A functor $F : \mathcal{C} \to \mathcal{D}$ is **full** if for all pairs $a, b \in \mathcal{C}$ the induced function:

$$\begin{aligned} F :\ \mathrm{Hom}_{\mathcal{C}}(a,b) &\to \mathrm{Hom}_{\mathcal{C}}(Fa, Fb), \\ f &\mapsto Ff \end{aligned}$$

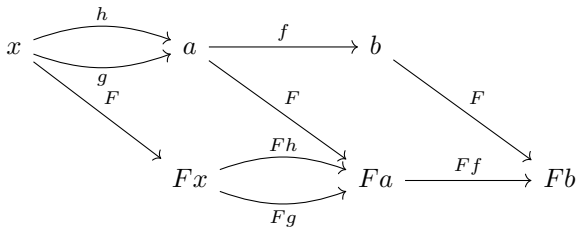
is a surjection. It is called **faithful** if it is an injection.

When after applying F an arrow Ff or an object Fa has a certain property(i.e. being initial, terminal or epi, mono), it is implied that f (or a) had this property, then we say the F ***reflects*** *the property.*

This allows for statements such as this:

Theorem 2. A faithful functor reflects epis and monos.

Proof. As an example we will prove it for a Ff that is mono. Let $f : a \to b$ such that Ff is mono, and let $h, g : x \to a$ such that $h \neq g$.



Since $g \neq h$ and F is faithful, we have $Fg \neq Fh$. This implies, because Ff is mono, that $Ff \circ Fg \neq Ff \circ Fh$, and since F is a functor we have $F(f \circ g) \neq F(f \circ h)$, implying $f \circ g \neq f \circ h$, and hence f is mono.

□

[2]Here we assume that this collection is a set, or that the category is so-called *locally small*

1.4 Natural transformations

Definition 6. A **natural transformation** μ between two functors $F, G : \mathcal{C} \to \mathcal{D}$ is a family of morphisms:

$$\mu = \{\mu_a : Fa \to Fb \mid a \in \mathcal{C}\},$$

indexed by objects in $\mathcal{C}$, so that for all morphisms $f : a \to b$ the diagram

$$\begin{array}{ccc} Fa & \xrightarrow{\mu_a} & Ga \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle Gf} \\ Fb & \xrightarrow{\mu_b} & Gb \end{array}$$

commutes. This diagram is called the *naturality square*. We write $\mu : F \Rightarrow G$, and call μ_a *the component of* μ *at* a.

We can *compose* natural transformations, turning the set of functors from $\mathcal{C} \to \mathcal{D}$ into a category. Let $\mu : F \Rightarrow G$ and $\nu : G \Rightarrow H$, then we have $\nu \circ \mu : F \Rightarrow H$ defined by (in components):

$$(\nu \circ \mu)_a = \nu_a \circ \mu_a.$$

Where the composition of the rhs is simply composition in $\mathcal{D}$.

Chapter 2

Types and functions: a category for programmers

> ""A monad is a monoid in the category of endofunctors, what's the problem?"
>
> *James Iry jokes about Haskell in his blog post A Brief, Incomplete, and Mostly Wrong History of Programming Languages*

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.

Why do we want to look at types? Programming safety and correctness. In this part we will hopefully give an idea of how category theory applies to programming, but we will not go into to much detail yet, this is saved for later parts.

We will take as our model for the category of types the category **Set**. Recall that the elements of **Set** are sets, and the arrows correspond to maps. There is a major issue to address here: Mathematical maps and functions in a computer program are not identical (bottom value $\perp$). We may come back to this, but for now we consider **Set**, although we will refer to the category **Hask** in the following (it is enough to think of it as **Set**).

In Haskell, we can express that an object has a certain type:

```
f :: Integer
```

In C++ we would write someting like this:

To define a function $f : A \to B$ from type A to type B in Haskell:

```
f :: A -> B
```

To compose:

```
g :: B -> C
h = f . g
```

This means that `h` is a function `h :: A -> C`! Note how easy it is to compose functions in Haskell. Compare how this would be in C++, if we were to take two polymorphic functions in C++ and compose them:

```
template <typename F, typename G>
auto operator*(F f, G g) {
    return [&](auto x) { return g(f(x)); };
}

int main() {
    auto f = [](int x) -> float { return ...; };
    auto g = [](float y) -> int { return ...; };

    std::cout << (f * g)(5) << "\n";
}
```

We need some additional operations to truly turn it into a category. It is easy to define the identity arrow in Haskell (at once for all types):

```
id :: A -> A
id a = a
```

in fact, this is part of the core standard library of Haskell (the Prelude) that gets loaded by default. Ignoring reference types and e.g. `const` specifiers, we can write in C++:

```
template <typename T>
T id(T x) {
    return x;
}
```

There is one issue we have glared over; in mathematics all functions are *pure*: they will always give the same output for the same input. This is not always the case for computer programs, using IO functions, returning the current date, using a global variable are all examples of impure operations that are common in programming. In Haskell, *all functions are pure*, and this is a requirement that allows us to make the mapping to the category **Set**. The mechanism that allows Haskell programs to still do useful things is powered by *Monads*, which we will discuss later.

Although many of the things we will consider can apply to other languages (such as Python and C++), there is a strong reason why people consider often consider Haskell as an example in the context of category theory and programming; it originates in academia and therefore takes care to model the language more accurately. For example, since we take as our model the category **Set**, there should be a type that corresponds to the empty set $\emptyset$. In C / C++, the obvious candidate would be `void` for this set, but consider a function definition:

```
void f() { ... };
```

This can be seen as a function from `void -> void`. We can call this function using `f()`, but what does it mean to call a function? We always invoke a function for an argument, so `void` actually corresponds to the set with a single element! Note that C functions that return void either do nothing useful (i.e. discard their arguments), or are impure. Indeed, even using a pointer argument to return a value indirectly modifies a 'global state'! In Haskell, the type corresponding to the *singleton set* (and its single value) is denoted with `()`. Meaning that if we have a function:

```
f :: () -> Int
```

we can invoke it as `f()`! Instead, the type `Void` corresponds to the empty set, and there can never be a value of this type. There is even a (unique) polymorphic (in the return type!) function that takes `Void` added to the prelude:

```
absurd :: Void -> a
```

You may be tempted to discard the type `Void` as something that is only used by academics to make the type system 'complete', but there are a number of legitimate uses for `Void`. An example is *Continuation passing style*, or CPS, where functions do not return a value, but pass control over to another function:

```
type Continuation a = a -> Void
```

In other words, a continuation is a function that *never returns*, which can be used to manipulate control flows (in a type-safe manner).

To summarize this introduction, in the category of 'computer programs', types are objects, and *pure* functions between these types are arrows. Next, we consider how we can apply some of the concepts we have seen, such as functors and natural transformations, to this category.

2.1 Containers as functors

When we consider functors in the category of types, the first question is 'to what category?'. Here, we will almost exclusively talk about functors from **Hask** to itself, i.e. *endofunctors*.

Endofunctors in **Hask** map types to types, and functions to functions. There are many examples of functors in programming. Let us first consider the concept of *lists of objects*, i.e. arrays or vectors. In C++ a list would be written as:

```
std::vector<T> xs;
```

or in Python we would have;

```
>>> import numpy as np
>>> a = np.array([1,2,3], dtype='int')
>>> type(a)
<class 'numpy.ndarray'>
>>> a.dtype
dtype('int64')
```

Note here that the true type of the numpy array is hidden inside the object, meaning its the responsiblity of the program to make sure that the types of operations match! The reason that we consider `numpy` arrays is that normal 'lists' in Python are actually *tuples*, which we will discuss when we talk about products and coproducts.

Let us consider the mathematical way of expressing this:

Example 1. Lists of some type are more generally called **words over some alphabet** (i.e. a set) X, and we denote the set of all finite words of elements in X as X^*. Elements in X^* look like:

$$(x_1, x_2, x_3)$$
$$(x_1)$$
$$()$$

These are all examples of *words* in X (where the last example corresponds to the empty word). If we want to construct a *word functor* T, then T would then have the signature:

$$\begin{aligned} T &: X \to X^* \\ &: (f : X \to Y) \mapsto (Tf : X^* \to Y^*) \end{aligned}$$

For this second option, we have an obvious candidate for the precise function, let $f : X \to Y$ be some map, then Tf maps a word in X gets to a word in Y in the following way:

$$Tf(x_1, x_2, x_3, ...x_n) = (f(x_1), f(x_2), f(x_3), \ldots, f(x_n)).$$

Type classes en type constructors

We will express this idea in Haskell, but before we can do this we first have to consider type classes and -constructors. A *type constructor* is a 'function' (on types, not an arrow) that creates a type out of a type. A *type constructor* can have multiple *value constructors*, and these constructors can be differentiated between using something called *pattern matching* which we will see later. As an example, consider `Bool`.

```
data Bool = True | False
```

Here, we define the type constructor `Bool` as the resulting type corresponding to the *value* given by the value constructors `True` and `False`, which both are nullary constructors (that take no argument as types!). Normally however, type constructors take one or multiple types for their value constructors:

```
data Either = Left a | Right b
```

Here, the type constructor either hold either a value of type `a` or of type `b`, corresponding to the value constructors `Left` and `Right`. We will revisit this idea (and `Either`) when talk about products and coproducts.

A type class is a *common interface for types*. It defines a family of types that support the same operations. For example, a type class for objects that support equality is defined as:

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

If we want to express the concept[1] *functor* using a typeclass, we have to state that it can send types to types, and that it sends functions between two types to functions with the appropriate signature, i.e.:

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

This says that `F` is a functor, if there is a function `fmap` that takes a function `f :: a -> b` and maps it to a function `fmap f :: F a -> F b`. Note that we dont explicitely have to state that `F` sends types to types, because this can be induced from the fact that we use `F a` where the compiler expects a type.

The List functor

The *list functor* in Haskell is denoted with `[]`, and a list of type `a` is denoted `[a]` (which is syntactic sugar, normally the type would be `[] a`).

Let us try to define this functor from the ground up. If we would write `List` instead of `[]`, then first we have to define what a list is. We can define this as follows:

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

Here the type constructor has two possible ways of constructing (partitioning the possible values of the type): a list of `a`s is either empty (corresponding to the *constructor* `Nil`), or that it is the concatenation (corresponding to the *constructor* `Cons`) of an object of type `a` with another list of `a`s. Note that this is a recursive definition!

Next we define the `fmap` corresponding to our `List` functor (i.e. how it maps functions). The corresponding definition to the map described for the *word functor* is:

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

If a list is empty, then we get te empty set, otherwise we map the indivual values in the list recursively using the given `f`. In `C++` this `fmap` functor roughly corresponds to `std::transform`, while for Python the closest thing would be the `map` function. With these two definitions, `List` is a functor! We could check the that it satisfies the requirements.

As mentioned, `List` is implemented in the standard library as `[]`, and `Cons` is written as `:`, while the empty set is written also as `[]`. This allows you to write:

```
x = 1 : 2 : [] -- this results in `[1, 2] :: [Int]`!
```

The Maybe functor

As a simpler example, consider a type that either has no value or it has a value corresponding to some type `a`. In Haskell, this is called `Maybe`, while in C++ this is called `std::optional`,

[1] In C++, type constructors are referred to as *concepts*, and they have been a long time coming (but are not yet in the standard)

in Python the same idea could be achieved using:

```
def fn(a):
    if (a >= 0)
        return sqrt(a)
    return None
```

This function returns `None` (corresponding to 'no value') if we provide 'invalid input'. This functor can be defined as:

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

And to turn it into a functor, we define `fmap`:

```
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just a) = Just (f a)
```

2.2 Polymorphic functions as natural transformations

Now that we view type constructors as functors, we can consider natural transformations between type constructors. If we let `a` be a type, then a natural transformation `alpha` would be something that maps between `F a` and `G a`, where `F` and `G` are type constructors:

```
alpha :: F a -> G a
```

Note that implicitely we talk about the component of `alpha` at `a`, since this function is *polymorphic* the right component gets picked by the compiler. For example, say we have a list `[a]`, and we want to obtain the first element of this list. If the list is empty, then there is no such element, otherwise we obtain an `a`; i.e. the result is a `Maybe a`:

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

Here, we have a natural transformation between the `List` and the `Maybe` functor!

Parametric polymorphism and ad-hoc polymorphism

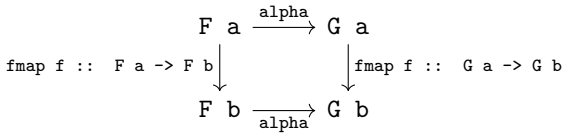
In C++, a template does not have to be defined for all types, i.e. we can write:

```
template <typename T>
T f(T a);

template <>
int f(int a) { return 2 * a; }

template <>
float f(float a) { return 2.0f * a; }
```

Here, e.g. `f<int>(1)` would yield `2`, while `f<char>('a')` would result in a compilation error.

In Haskell, this is not allowed, polymorphic functions must work for *all types*, this is called parametric polymorphism. Specializing function definitions is done using type classes[2]. This has an important consequence (or perhaps, it is the underlying reason): a parametric polymorphic function satisfies automatically the naturality conditions.

The corresponding diagram is:



Here the left `fmap` works for `F`, while the right `fmap` corresponds to `G`, and the top `alpha` is implicitely the component at `a`, while the bottom one is the component at `b`. What would have to show, is that automatically

```
fmap f .  alpha = alpha .  fmap f
```

This can be shown in a very general context, and it has to do with the fact that the 'bodies'for `f`, `fmap` and `alpha` are the same for all types. We will show this in an upcoming part when we discuss *free theorems*.

[2] in C++ this would be done using overloading and (partial) template specialization

Chapter 3

Products, Co-products and Algebraic Datatypes

Bifunctorality and `Either` o

Chapter 4

Monads

Chapter 5

Yonedda's Lemma, reader and writer monads, Kleisli categories

Chapter 6

Limits and co-limits

Chapter 7

‘Theorems for free!’

Chapter 8

Ideas

Section on 'modularity':

> *Bartosz Milewski*: "... Elegant code creates chunks that are just the right size and come in just the right number for our mental digestive system to assimilate them. So what are the right chunks for the composition of programs? Their surface area has to increase slower than their volume ... The surface area is the information we need in order to compose chunks. The volume is the information we need in order to implement them ... Category theory is extreme in the sense that it actively discourages us from looking inside the objects ... The moment you have to dig into the implementation of the object in order to understand how to compose it with other objects, you've lost the advantages of your programming paradigm."

Chapter 9

Literature

9.1 Blogs

1. *Bartosz Milewski*: "Category Theory for Programmers", a blog post series that gives an excellent overview of interesting topics. https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/

9.2 Papers

1. About **Hask**: http://www.cs.ox.ac.uk/jeremy.gibbons/publications/fast+loose.pdf
2. Free theorems: http://ttic.uchicago.edu/~dreyer/course/papers/wadler.pdf (also Reynold: http://www.cse.chalmers.se/edu/year/2010/course/DAT140_Types/Reynolds_typesabpara.pdf).

9.3 Books

1. Conceptual Mathematics: A first introduction to categories.
2. Maclane, Category Theory for the working mathematician
3. Category Theory for Computer Scientists