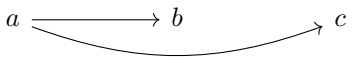
Category Theory for Programmers

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++).



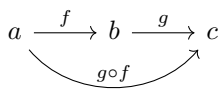
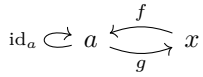
$c \qquad\qquad d$

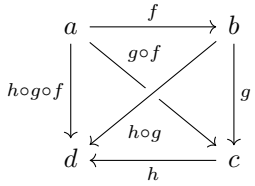
Chapter 1

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.

Examples

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 $S \subset A$:

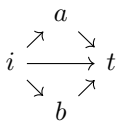
$$\mathcal{P}A = \mathcal{P}(A),$$
$$\mathcal{P}f : \mathcal{P}(A) \to \mathcal{P}(B),\ S \mapsto f(S)$$

1.3 Special objects, arrows and functors

Special objects

For objects, we distuinguish two special kinds:

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 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[1] 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:

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

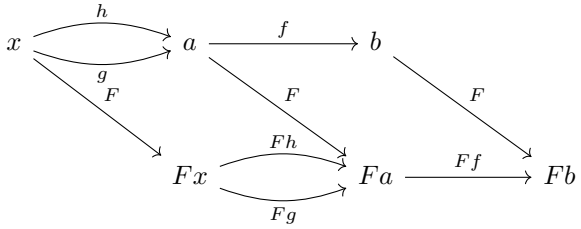
is a surjection. It is called **faithful** if it is an injection.

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.

□

[1] 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: categories in functional programming

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. Maybe something like declarative programming.

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 will com back to this, but for now we consider **Set**.

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

```
a :: Integer
```

In C++ we would write:

```
int a;
```

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*(G g, F f) {
    return [&](auto x) { return g(f(x)); };
}

int main() {
    auto f = [](int x) -> float { return (x * x) * 0.5f; };
    auto g = [](float y) -> int { return (int)y; };

    std::cout << (g * f)(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;
}
```

So, types are objects, and (computer program) functions between these types are arrows. Can we apply some of the concepts we have seen, such as functors and natural transformations?

Functors in Haskell. Type constructors. Note that here we only look at *endofunctors*.

Polymorphic functions as natural transformations

Kleisli category?

Chapter 3

Products, Co-products and Algebraic Datatypes

Chapter 4

Monads

Chapter 5

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 6

Literature

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

Chapter 7

Pandoc examples: