Categories from scratch

Raphael Poss

April 2014

modified: 2018-09-03
category: Computer Science
tags: analysis, functional programming

Contents

Prologue 3

Overview 3

Topics of study 3

Recollections 4
- Background terminology . . . 4
- Observability . . . 4
- Grouping things together . . . 4
- Composition semantics . . . 5
- Neutral behaviors . . . 5
- Associativity . . . 7

Identification of the analogies 7
- Objects and morphisms . . . 7
- Modeling: be careful about equivalences . . . 7
- Arrow notation . . . 8
- Composition . . . 9
- Identity . . . 9
- Definition of a category . . . 9

Goodies from category theory 10
- Unicity of the identity . . . 10
- Invertibility and isomorphisms . . . 10
- Duality . . . 11
- Functors . . . 11
- Some follow-up concepts to read about . . . 12

Further reading 12

Acknowledgements 12

Copyright and licensing 12

Note

The latest version of this document can be found online at https://dr-knz.net/categories-from-scratch.html. Alternate formats: Source, PDF.

Prologue

The concept of *category* from mathematics happens to be useful to computer programmers in many ways. Unfortunately, all "good" explanations of categories so far have been designed by mathematicians, or at least theoreticians with a strong background in mathematics, and this makes categories especially inscrutable to external audiences.

More specifically, the common explanatory route to approach categories is usually: "here is a formal specification of what a category is; then look at these known things from maths and theoretical computer science, and admire how they can be described using the notions of category theory." This approach is only successful if the audience can fully understand a conceptual object using only its formal specification.

In practice, *quite a few people only adopt conceptual objects by abstracting from two or more contexts where the concepts are applicable,* instead. This is the road taken below: reconstruct the abstractions from category theory using scratches of understanding from various fields of computer engineering.

Overview

The rest of this document is structured as follows:

1. introduction of example Topics of study: unix process pipelines, program statement sequences and signal processing circuits;
2. Recollections of some previous knowledge about each example; highlight of interesting analogies between the examples;
3. Identification of the analogies with existing concepts from category theory;
4. a quick preview of Goodies from category theory;
5. references to Further reading.

Topics of study

"**Pipes**" Unix process pipelines: chains of Unix processes linked by FIFOs.

"**Compilers**" Programs that transform programs.

"**Circuits**" Signal processing circuits: circuits in the real world with physical connectors for input and output of electrical signals.

Many introductory texts about category theory use another group of examples: type systems for programming languages, matrices from linear algebra and sometimes directed graphs. I did not choose them here as I believe they are already too abstract for many computing engineers. Instead, I believe these examples should only be mentioned as additional instances of categories after the concept has been properly recognized.

Recollections

Background terminology

Pipes:
In a Unix system, processing activities are organized in *processes*. Nearly all *communication* between processes and with I/O devices is organized via file descriptors: to real files on disk, to terminals with the user, but also FIFO buffers between processes. Communication can be monitored directly on terminals, or by looking at the contents of files on disk, or interleaving the program `tee` between programs chained by FIFOs.

Compilers:
A compiler is a program that takes another program as input, and transforms it to produce another program as output. The set of valid inputs for a compiler is its *input language*, and the set of possible outputs is its *output language*. For example, a "C compiler" accepts C code as input and produces assembly code as output (for some specific ISA).

Circuits:
An electronic circuit is usually recognized when one sees a plastic enclosure with some metal bits sticking out. *Signal processors* are a specific type of circuit, with a notion of "input" and "output" connectors: For example, a hi-fi amplifier has distinct pins for the "audio source" and one or more "speaker" output connectors. When *plugged in* to an active input signal, the circuit is itself activated and starts driving its output pins. The output signal can be exploited by further plugging to other circuits, or output devices. It can also be measured, for example using an oscilloscope.

Observability

Each topic has two "groups" of *things*: things that can be *observed* from the "outside", and things that appear (mostly) as *black boxes* from the outside but are "connected" between the things in the first group:

Example:	**Pipes**	**Compilers**	**Circuits**
Observables:	The data streams going through file descriptors.	The set of all possible programs written in the input languages or generated in the output language.	The electric signals over time.
Black boxes:	The running processes with their internal state (program counter, stack, heap, etc.).	The program transformation algorithms.	The circuits themselves.

Grouping things together

Each topic provides a mechanism to plug the "black box" things together:

Pipes:	**Compilers**:	**Circuits**:
Given the two commands "`grep -v foo`" and "`grep -v bar`", one can write the command "`grep -v foo | grep -v bar`", which combines the two behaviors.	Given a compiler from C to assembly text, and a compiler from assembly text to machine code, one can combine them (eg. by means of a script) to create a compiler from C to machine code.	Given the a DVI-VGA adapter and a VGA-SVideo adapter, it is possible to plug them together to form a DVI-SVideo adapter.

Composition semantics

Each topic has a notion of "good" compositions that "make sense", and "bad" compositions that nonsensical and not expected to "work properly":

Pipes:	**Compilers**:	**Circuits**:
Piping `ls` with `grep foo` is sensical.	Connecting the output of a C-to-assembly compiler to the input of a assembly-to-code compiler is sensical.	Plugging a USB-serial adapter to a DB9-DB25 serial adapter (with a 9-pin interface between them) is sensical.
Piping `ls` with `gv` (PostScript viewer) is nonsensical.	Connecting the output of a C-to-C compiler to the input of an assembly-to-code compiler is nonsensical.	Plugging a USB-serial adapter to an EGA display (physically possible as they share the same DB9 connector) is nonsensical.

To identify "good" from "bad" compositions, each topic places a large emphasis on the notion of *interface*:

Pipes:	**Compilers**:	**Circuits**:
Both `ls` and `grep` operate on plain text streams, which is why they compose well with pipes. In contrast `gv` expects PostScript as input, which `ls` cannot produce.	Each compiler has a notion of *input language* for the set of accepted input programs and *output language* for the set of possible outputs. The languages must match when composing the compilers together.	The USB-serial adapter and serial-serial adapter plug well together because they both use the same standard (RS232) signalling protocol at the interface.

Remarkably, users *understand* or conceptualize interfaces, despite the fact they are not always defined explicitly beforehand.

Neutral behaviors

Each topic has some instances of black box things that "do nothing", ie have a *neutral* behavior:

Pipes:	**Compilers**:	**Circuits**:
when the input and output data streams are the same (byte-wise).	when the input and output languages are the same.	when the output signals are measured the same as the input.

The *knowledge* of whether a black box is neutral can be gained in either or three ways.

Either it is known to be neutral *by construction*, because the specification is available for scrutinity and can be proven to define a neutral behavior:

Pipes:	**Compilers**:	**Circuits**:
The program source code contains a loop that iteratively inputs a byte and outputs the same byte, stopping at end-of-stream.	The transformation algorithm does not change the semantics of the program.	The blue print defines direct links between the input and output pins.

Or, the knowledge is provided *externally*, eg. by fiat:

Pipes:	**Compilers**:	**Circuits**:
The manual page for a command specifies that the process will replicate its input to its output unchanged.	The documentation says it is a "source-to-source" compiler, or explicitly indicates that its input and output language are the same.	The manufacturer guarantees that the circuit is fully pass-through.

Or, it is *discovered*: to find out whether a black box thing A is neutral, assuming an observer has access to a pre-existing, valid "observable thing" that can be fed to A, then the observer can deduce the behavior of A is "neutral" if the observable as a result of A's activity is the same as the original observable:

Pipes:	**Compilers**:	**Circuits**:
An unknown program `xxx` is known *a priori* to only read from its standard input and write to its standard output. So one can run the command "`xxx <iN >oN`" for various input files `iN` and compare whether the contents of each file `oN` are equal to the corresponding `iN`. If so, `xxx` appears to be "neutral".	The input language is known, but not the output language. So one can generate some random but valid input programs and feed them to the compiler. If the output programs are also valid input programs, then original program piece appears to be "neutral".	A signal processor appears to have the same number of input and output pins, and its input signal specification is known *a priori*. So one can use various valid input signals, feed them to the circuit, and measure the output. If the output signals measures the same as the input every time, the circuit appears to be "neutral".

Once a black box thing N is known to be neutral, then its composition "left" and "right" with another black box thing A can be assumed to have the same behavior as A on its own:

Pipes:	**Compilers**:	**Circuits**:
Both "`xxx | grep foo`" and "`grep foo | xxx`" can be assumed to behave like `grep foo` once `xxx` is known to be neutral.	A compiler built by composing a neutral compiler N either before or after another compiler C will have the same input and output languages as C.	Plugging a neutral circuit on either the input or output side of another circuit A will process signals as A alone would.

Associativity

When one composes three black box things A, B and C together (assuming the compositions are point-wise sensical), the *order* in which the composition is realized does not change the behavior:

Pipes:
The commands "`(ls | grep -v foo) | grep -v bar`" and "`ls | (grep -v foo | grep -v bar)`" have equivalent behavior on their data streams.

Compilers:
If a script A invokes a Scheme-to-C compiler C1 followed by a C-to-assembly compiler C2, and another script B invokes A and then an assembly-to-code compiler C3, then B has the same input and output language as a script C that calls C1 then D, where D calls C2 then C3.

Circuits:
Whether a DVI-VGA adapter is plugged into a VGA-SVideo adapter, and then the result is plugged into a TV screen with SVideo input, or if a VGA-SVideo adapter is first plugged into the TV, and then plugged to a DVI-VGA adapter, both resulting circuits are working TVs from a DVI input signal.

Identification of the analogies

The previous section has introduced the following key concepts:

- things that can be "observed" from the outside, and "black box" things connected between observables;
- neutral black box things that preserve behavior;
- associative composition of black box things.

These are the concepts manipulated in category theory.

Objects and morphisms

First, the observables are named **objects**. The black boxes things are named **morphisms** or **arrows**.

Example:	**Pipes**	**Compilers**	**Circuits**
Objects:	Data streams.	Languages.	Signals.
Morphisms:	Processes.	Compilers.	Circuits.

Modeling: be careful about equivalences

Categories are defined over *mathematical sets* of objects and morphisms. Sets are different from simple “collections” (or “bag”) of things from the real world: all their elements are distinct, according to some *equivalence* relation.

So in order to talk about categories over things from the real world, we must first choose how to define the mathematical sets. This choice is called a *model* and multiple models are possible for the same collection of things.

The most focus should be given to the set of morphisms. The set of objects is simply derived from it once the morphisms are properly identified. For example, in our pipes example, considering what happens to data streams. What does it mean for two processes to be equal or different?

We can choose for example "data stream equality". By this standard, two processes that filter out lines containing the text "foo" over any data stream are the "same thing." So "`sed '/foo/d'`" and "`grep -v foo`" are the same morphism.

If we choose this definition for morphisms, then the objects are not individual files (or time-particular datastreams over FIFOs), but rather entire classes of all possible data streams that compare equal to each other byte by byte. For example, a stream that delivers "helloworld" in one go is the same stream as another that delivers "hello" and then "world" 5 seconds later.

Another possible choice for a definition is "physical equality". By this standard, two processes that run at different times or in different physical regions of the system are distinct, even if they perform the same task. So two processes run from the same command (eg `cat`) at different times end up as different morphisms in the set.

If we choose this definition, then the mathematical objects are not only data streams, but also where and when the bytes are physically encoded. So two streams that deliver "helloworld" in different places/times are distinct objects.

The rule of thumb while choosing a definition is the following: if one wants to talk about categories over sets of objects and morphisms that are *already* mathematically defined, then all is well. If one wants to use category theory over things that are not yet mathematical, be careful to explain *clearly and explicitly* how they are modeled using mathematical sets, and which equivalence relation is used.

For the next sections, we use the following definitions:

Pipes:
We use "data stream equality" as defined above. With this, the commands "`sed '/foo/d'`" and "`grep -v foo`" define the same morphism; so do "`tr x y`" and "`sed s/x/y/g`".
With this definition, each morphism may have multiple names (different commands to define it). This is ok.

Compilers:
We use "program equality": two compilers are the same morphism if they produce the same output program from the same input program. By this definition, two different C-to-assembly compilers (eg. `gcc` and `clang`) are different morphisms, but a compiler defined by a script combining `cpp` with `gcc` is the same morphism as `gcc` on its own.
Again, with this definition, each morphism may have multiple names.

Circuits:
We use "interface and protocol equality". With this, both the USBGEAR/USBG-232FT-1 and MCT/U232-P9 are the same morphism, as they both interface USB to RS232. However, a USB-to-EGA adapter would not be the same morphism, because it uses a different signal protocol even though the interface (DB9) is the same.

Arrow notation

The **arrow notation** *describes* a morphism: "$A \to B$" is a description for a morphism from object A to object B. The two sides of a morphism can be named "input" and "output", but are usually named "source" and "target".

There may be multiple morphisms between any two objects, so the arrow notation does not identify a particular morphism; instead, it can be seen as an "interface" or "type" for the set of all morphisms between

the designated objects:

Pipes:	**Compilers:**	**Circuits:**
Both the processes resulting from running "`tr x y <f1 >f2`" and "`tr yz <f1 >f2`" are different morphisms, and both can be described by the arrow "`f1` $\rightarrow$ `f2`".	Both the `gcc` and `clang` programs are morphisms, and both can be described by the arrow "C $\rightarrow$ assembly".	Both the USBGEAR/USBG-232FT-1 and MCT/U232-P9 are morphisms, and both can be described by the arrow "USB $\rightarrow$ RS232".

Composition

If two morphisms have the same intermediary object, it is possible to compose them together (cf. Composition semantics above). This is abstracted by an **composition operator** noted "$\cdot$": given two compatible morphisms f and g, "$g \cdot f$" designates their composition, ie. $(g \cdot f)(x) = g(f(x))$.

By construction, if f can be described by $A \rightarrow B$ and g by $B \rightarrow C$, then $g \cdot f$ can be described by $A \rightarrow C$.

Composition is associative: for any f, g, h, $(f \cdot g) \cdot h = f \cdot (g \cdot h)$.

Identity

For any object, there must exist at least one morphism that keeps the object unchanged. Each such **identity** morphism for an object x is called "id_x" (sometimes also "$\mathbf{1}_x$") and can be described as $x \rightarrow x$; it must satisfy the following property: for every morphism $f : A \rightarrow B$, $\mathrm{id}_B \cdot f = f = f \cdot \mathrm{id}_A$.

So there must be at least as many identity morphisms as there are objects.

For categories defined by modeling over concrete things, it may be necessary to extend the mathematical set of morphisms with "theoretical" identity morphisms, when there are no concrete identities.

For example, it is not possible to build a concrete identity circuit: any one-to-one pairing of physical input and output connectors with direct wires between them is bound to introduce noise in the signal due to the physical distance. However, the mathematical set modeling circuits can be naturally extended to include "virtual" identity circuits that preserve signals unchanged.

Hopefully, with many categories the identity morphisms can be concretely constructed in the application domain:

Pipes:	**Compilers:**	**Circuits:**
For any data stream, the morphism defined from the commands "`cat`" or "`grep '.*'`" is an identity.	For the C language, the preprocessor (`cpp`) is an identity.	see above.

Definition of a category

A **category** is an *algebraic structure* formed over:

- a (mathematical) set of objects,
- a (mathematical) set of morphisms over these objects containing at least one identity morphism for each object,

- an associative composition operator.

Category:	**Pipes**	**Compilers**	**Circuits**
Objects:	Data streams.	Languages.	Signals.
Morphisms:	Stream transformers.	Compilers.	Models of circuits.
Identities:	Running `cat` or similar pass-through commands.	`cpp` for C, in general `cat` for any langage.	Virtual pass-through circuits.
Composition:	Chaining processes via FIFO buffers, eg by running them with the pipe operator in commands.	Creating a script that invokes two existing compilers, applying the 2nd on the output of the 1st.	Plugging the circuits together.

Goodies from category theory

Unicity of the identity

Although the *existence* of identity morphisms is a prerequisite to form a category (axiomatic), it is possible to prove within category theory that each identity is *unique*.

You can do this as follows.

Suppose you have two morphisms $\mathrm{id1}_x$ and $\mathrm{id2}_x$ that preserve object x and satisfy the axiomatic identity properties:

1. for every $f : y \to x$, $\mathrm{id1}_x \cdot f = \mathrm{id2}_x \cdot f = f$; and
2. for every $g : x \to y$, $g \cdot \mathrm{id1}_x = g \cdot \mathrm{id2}_x = g$.

In equation #1, replace f by $\mathrm{id2}_x$, and you find that $\mathrm{id1}_x \cdot \mathrm{id2}_x = \mathrm{id2}_x$.
In equation #2, replace g by $\mathrm{id1}_x$, and you find that $\mathrm{id1}_x \cdot \mathrm{id2}_x = \mathrm{id1}_x$.
Since both left-hand sides are equal, you have proven that $\mathrm{id2}_x = \mathrm{id1}_x$. ■

What this means in practice: if you can construct/define two morphisms in a category and prove they satisfy the identity laws, then you have proven they are the same morphism. In our examples, that means identity commands (pipes), compilers or circuits become interchangeable with regards to their properties in category theory. This can be used to simplify formulas that use complex morphisms into simpler ones.

Invertibility and isomorphisms

The general notion of invertibility for a morphism can be expressed purely in the vocabulary of category theory:

$f : A \to B$ is invertible if there exists $g : B \to A$ such that $g \cdot f : \mathrm{id}_A$ and $f \cdot g : \mathrm{id}_B$.

Invertible morphisms are also called **isomorphisms**.

By extension, two *objects* A and B are **isomorphic** if there exists at least one isomorphism described by $A \to B$.

Duality

For any category C, it is possible to define mathematically another category C^{op} where the source and target of every morphism are interchanged. This is called the "opposite category" or **dual category** of C.

Duality is "invertible": $(C^{op})^{op}$ is the same category as C.

The dual category of a category C of concrete objects may be purely abstract, ie. without concrete representations for morphisms in the application domain of C, for example if the morphisms in C are not invertible.

Nevertheless, duality serves an important purpose: say, you have demonstrated a property that holds within a category C, which you can express within the language of category theory using a formula σ (some text string).

If you then replace all occurences of "source" by "target" and vice-versa in σ, and all occurences of $f \cdot g$ by $g \cdot f$, you obtain a new formula σ^{op}. By construction, this formula is true in C^{op}. It is said to be the **dual property** of σ. Conversely, if you know a property σ to be true in C^{op}, then σ^{op} will be true in C as well.

Why this is useful: many important/useful results and properties of mathematics come in pairs that are expressed using "symmetric" formulas, which are dual in category theory. For example, **monomorphisms** and **epimorphisms** are morphisms for which different properties hold, but their definitions are dual. From this, if one can derive another unrelated property ϕ that relies on the fact a morphism is monomorphic, then thanks to duality, automatically the dual property ϕ^{op} is also proven over epimorphisms. Thanks to category theory, many pairs of results/theorems in algebra can be obtained with half the effort.

Functors

The observation that there are some common features between different categories intuitively brings the idea to transform one category into another, while preserving its structure.

For example, our category of "pipes" over Unix data streams can be transformed into a category of "networked services" over network data streams trivially, by attaching the program `nc` around each Unix processes.

Such a transformation of a category into another is called a **functor**. Generally, a functor *F* from a category *C* to a category *D* has the following properties:

- for each object *x* in *C*, *F* associates an object in *D* noted *F(x)*;
- for each morphism $f : x \to y$ in C, *F* associates a morphism in *D* that can be described by $F(x) \to F(y)$, noted *F(f)*;
- for every identity morphism $h : x \to x$ in *C*, *F(h)* is an identity morphism for *F(x)* in *D*;
- for every pair of morphisms *f* and *g* in *C*, $F(g \cdot f) = F(g) \cdot F(f)$ in *D*.

(This specific flavor of functors is called "covariant." If *F* maps each arrow in *C* to an arrow with opposite direction in *D*, it is called "contravariant" instead. Covariant and contravariant functors are dual.)

Side property: Because of the properties of functors, the algebraic structure formed by 1) a set of categories 2) a set of functors over these categories 3) the identity functors that leave each category in set #1 unchanged, and 4) the natural generalization of composition, together, is itself a category.

Some follow-up concepts to read about

- **Natural transformations**: a constuction that transforms a functor into another functor, that respects the category structure of the functor transformations.
- **Functor category**: category where the objects are functors, and the morphisms are natural transformations between them.
- **Categorical logic**, especially used in computer science, focuses on semantic systems with a difference between syntax and semantics. To use categorical logic, one defines one category for syntax, one category for semantics, and phrases interpretation as a functor between them. An example application is proofs of behavior correctness, or "correct by construction" languages: by choosing an appropriate interpretation functor, proofs over the syntax category can be carried over transparently to the category of semantics.

Further reading

- Benjamin L. Russell. Motivating Category Theory for Haskell for Non-mathematicians. Benjamin's Adventures in Programming Language Theory Wonderland, January 2009.
- James Cheney. Category theory for dummies (I). Programming Languages Discussion Group, March 2004.
- José Antonio Ortega Ruiz. Programmers go banana. Programming musings, March 2006. Contains an extremely synthetic yet approachable definition of categories with diagrams.
- Gabriel Gonzalez. Model-view-controller in Haskell. Haskell for All, April 2014. Explains how to use category theory to abstract the model-view-controller pattern of software engineering in a type-safe manner.

Acknowledgements

I am grateful to Karst Koymans for helping develop my understanding of categories and denouncing major problems with an earlier version of this document.

Copyright and licensing

Copyright © 2014-2026, Raphael Poss. Permission is granted to distribute, reuse and modify this document according to the terms of the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.