Categories for programmers: foreword

Brendan Fong Bartosz Milewski David I. Spivak

January 7, 2020

This course is about how to think using category theory. What does this mean? Category theory is a favorite tool of ours for structuring our thoughts, whether they be about pure math, science, programming, engineering, or society. It's a formal, rigorous toolkit designed to emphasise notions of relationship, composition, and connection. It's a mathematical tool, and so is about abstraction, stripping away details and noticing patterns; one power it gains from this is that it is very concise. It's great for compressing thoughts, and communicating them in short phrases that can be reliably decoded.

The flip side of this is that it can feel very unfamiliar at first – a strange, new mode of thought – and also that it often needs to be complemented by more concrete tools to fully realise its practicality. One also needs to practice a certain art, or taste, in knowing when an abstraction is useful for structuring thought, and when it should be broken, or approximated. Programmers know this well, of course: a programming language never fully meets its specification; there are always implementation details that the best must be familiar with.

In this book, we've chosen programming, in particular *functional programming*, as the vehicle through which we'll learn how to think using category theory. This choice is not arbitrary: category theory is older than digital computing itself, and has heavily influenced the design of modern functional programming languages such as Haskell. As we'll see, it's also influenced best practice for how to structure code using these languages.

Thinking is not just an abstract matter; the best thought has practical consequences. In order to teach you how to think using category theory, we believe that it's important to give a mechanism for implementation and feedback on how category theory is affecting your thought. So in this book, we'll complement the abstraction of category theory – lessons on precise definitions of important mathematical structures – with an introduction to programming in Haskell, and lessons on how category theoretic abstractions are approximated to perform practical programming tasks.

Category theory is a vast toolbox, that is still under active construction. A vibrant community of mathematicians and computer scientists is working hard to find new perspectives, structures, and definitions that lead to compact, insightful ways to reason about the world. We won't teach all of it. In fact, we'll teach a very small core, some central ideas all over fifty years old.

The first we'll teach is the namesake: categories. Programming is about composition: it's about taking programs, some perhaps just single functions, and making them work in sync to construct a larger, usually more expressive, program. We call this way

of making them work in sync *composition*. A category is a world in which things may be composed. In programming these things are called, well, programs, but in category theory we use the more abstract word *morphism*. Morphisms are often thought of as processes or relationships. Processes have an input and an output; similarly, relationships form between objects. Morphisms are similar: categories also have *objects*, and morphisms have an input object, called a *domain*, and an output object, called a *codomain*.

Since programming is about composition, and categories model the essence of composition, one interesting game to play is to think of – we'll say *model* – a programming language as a category. In this model, morphisms correspond to programs. What do the domain and codomain of a morphism correspond to? The objects of the category correspond to *types*, like `Int` or `String`. In a category, we can only compose a morphism f with a morphism g if the codomain of f is the same as the domain of g. If we're modelling a programming language as a category, this suggests we want to only allow composition of programs f and g if the output type of f is the same as the input type of g. A program that consumes an `Int` shouldn't be able to accept a `String`! Haskell is designed with this principle in mind, and checks for this sort of error at compile time, only allowing construction of programs that are well-typed.

Category theory is about relationships, and as part of this viewpoint, relationships between categories are very important. The basic sort of relationship is known as a *functor*. A functor between categories $\mathcal{C}$ and $\mathcal{D}$ must give an object of $\mathcal{D}$ for every object of $\mathcal{C}$. These correspond to type constructors (which a bit of additional structure). Going deeper still, the basic sort of relationship between functors is known as a *natural transformation*. These too are useful for thinking about programming: they are used to model polymorphic types.

As may be now clear, types play an a fundamental role in thinking about programming from a categorical viewpoint. We'll next talk about how category theory lends insight into methods for constructing new types: first algebraic datatypes, and then recursive datatypes.

In category theory, this becomes a question of how to construct new objects from given objects. In category theory we privilege the notion of relationship, and a deep, beautiful part of theory concerns how to construct new objects just by characterizing them as having a special place in the web of relationships that is a category. For example, in Haskell there is the unit type `()`, and this has the special property that every other type `a` has a unique function `a → ()`. We say that the unit type thus has a special *universal property*, and in fact if we didn't know the unit type existed, we could recover or construct it by giving this property. More complicated universal properties exist, and we can construct new types in this way. In fact, we'll see that it's nice if our programming language can be modelled using a special sort of category known as a *cartesian closed category*; if so, then our programming language has product types and function types.

Another way of constructing types is using the (perhaps confusingly named) notion of algebras and coalgebras for a functor. In particular, specifying a functor lets us talk about universal constructions known as initial algebras and final coalgebras. These allow us to construct recursive datatypes, and methods for accessing them. Examples include lists and trees.

Functional programming is very neat and easy to reason about. It's lovely to be able to talk simply about the program `square: Int -> Int`, for example, that takes

an integer, and returns its square. But what happens if we also want this program to wait for an input integer, or print the result, or keep a log, or modify some state variable? We call these *side effects* of the program. Functional programming teaches that we should be very careful about side effects, as they can happen away from the type system, and this can make programs less explicit and modular, and more difficult to reason about. Nonetheless, we can't be too dogmatic. We do want to print results! Monads are special functors that each describe a certain sort of side effect, and are a useful way of being careful about side effects.