What did category theory ever do for us (functional programmers)

An extreme pragmatic un-academic approach

Sergei Winitzki

Scale by the Bay

2019-11-15

Functional programming, pragmatically

What do functional programmers do most often (and most benefit from)?

- Work with collections using methods `map`, `filter`, `fold`, `flatMap`
- Use **disjunctive types** (“case classes”, “algebraic data types”) for domain modeling
- Use type constructors with arbitrary type parameters
  `Flow[KafkaMessage[MyData], KafkaMessage[MyResult], NotUsed]`
- Use **functor blocks** (`for` / `yield` syntax, “for comprehensions”)

```
for {
  x <- Future(...)
  y <- Future(...)
} yield f(x, y)
```

- Use curried functions (e.g. Akka Streams)

```
GraphDSL.create { implicit builder => (source, sink) => ... }
```

No category theory needed so far!

- Programmers can learn these features of Scala and become productive

Uses of category theory. I. Typeclasses with laws

Typeclasses: type constructors with methods `map`, `filter`, `fold`, `flatMap`
To be useful for programming, the methods must satisfy certain laws

- `map`: identity, composition
- `filter`: identity, composition, partial function, naturality
- `fold` (traverse): identity, composition, naturality
- `flatMap`: identity, composition, naturality

Example of incorrect implementation of typeclasses:

```
case class Bad[A](x: Int, do: A => Int) {
  def map[B](f: A => B): Bad[B] = Bad(x, _ => x)
  def flatMap[B](f: A => Bad[B]): Bad[B] = Bad(x, _ => x)
  def filter(p: A => Boolean): Bad[A] = this
}
```

We can now use `for` / `yield` syntax with sources of type `Bad`...
...but the code will have strange, hard-to-find bugs:

```
val b: Bad[String] = Bad(1, _.length)
val c: Bad[String] = for {
                       x <- b  // x : String ??
                       y = x
                     } yield y


scala> b.do("abc")
res0: Int = 3

scala> c.do("abc")
res1: Int = 1
```

The methods of `Bad` are not lawful (and *cannot* be!)

One can transform and simplify typeclass laws into a form of "lifting"

- Need to derive equivalent formulations of typeclass laws...
  - ...two months later:

Typeclass	Method	Type signature
functor	`fmap`	`(A => B) => F[A] => F[B]`
contrafunctor	`cmap`	`(B => A) => F[A] => F[B]`
filterable functor	`liftOpt`	`(A => Option[B]) => F[A] => F[B]`
monad	`flatMap`	`(A => F[B]) => F[A] => F[B]`
applicative functor	`ap`	`F[A => B] => F[A] => F[B]`
comonad	`coflatMap`	`(F[A] => B) => F[A] => F[B]`

Finding patterns in typeclass laws. II. Laws of liftings

The laws of typeclass methods follow the same pattern ("lifting laws")

$$\text{identity law}: \quad \text{lift}\,(\text{id}_1) = \text{id}_2$$

$$\text{composition law}: \quad \text{lift}\,(f) \circ \text{lift}\,(g) = \text{lift}\,(f \diamond g) \quad \text{with some definition of } \diamond$$

- Heuristic picture of "lifting":
  - types `A`, `B`, ..., are lifted into types `F[A]`, `F[B]`, ...
  - one "twisted" function type is lifted into another
- Category theory provides a universal language ("category", "functor") that describes all these examples
  - A "category" $\mathcal{C}$ is a description of what types A,B, ... ("objects") are used and how to twist the function types ("morphisms $A \rightsquigarrow B$") e.g. `F[A] => B` or `B => F[A]` etc.
    - ⋆ Must have an identity morphism $\text{id}_{\mathcal{C}} : A \rightsquigarrow A$, a composition operation $\diamond_{\mathcal{C}}$, and the identity and associativity laws
  - A "functor" is a description of how to map types and functions from one category to another
    - ⋆ Functors need to satisfy the laws of identity and composition

Examples of categories

- "Plain": Morphisms $A \rightsquigarrow B$ are plain functions `A => B`
- `F`-lifted: Objects are `F[A]`, morphisms `F[A] => F[B]`
- `F`-Kleisli: Morphisms `A => F[B]`, identity morphism `pure: A => F[A]`, `F`-Kleisli composition: $f \diamond_F g =$ `f andThen _.flatMap(g)`
  - Kleisli composition `(A => F[B])` $\diamond_F$ `(B => F[C]) : (A => F[C])`
  - The laws of `F`-Kleisli category are equivalent to the laws of monads
- `F`-Applicative: Morphisms `F[A => B]`, identity morphism `pure(identity): F[A => A]`, applicative composition: `map2(f, g, p => q => p andThen q)`
  - The laws of `F`-Applicative category are equivalent to the laws of applicative functors
- Opposite category: like "plain" except functions are `B => A` and composed in the opposite order

Example of using category theory: Filterable functors

Filterable functors `F[_]` have `filter[A](p: A => Boolean): F[A] => F[A]`
An *equivalent* method is `liftOpt[A, B](f: A => Option[B]): F[A] => F[B]`

- The laws of `liftOpt` are the laws of (categorical) functor from the `Option`-Kleisli category to the `F`-lifted category

$$\text{liftOpt}\,(f^{:A\rightarrow\text{Opt}^B})\,{}^\circ_9\,\text{liftOpt}\,(g^{:B\rightarrow\text{Opt}^C}) = \text{liftOpt}\,(f\diamond_{\text{Opt}} g)$$

```
liftOpt(f) andThen liftOpt(g) == liftOpt(f andThen _.flatMap(g))
```

- Category theory assures that the chosen laws are useful and consistent
  - It is not obvious what the laws of `filter` must be
- Category theory suggests two generalizations:
  - Use a different monad M instead of `Option`, obtain M-filterable functors
  - Use the opposite category and obtain filterable contrafunctors
- Category theory *does not* show how to derive or prove laws

Uses of category theory. II. Type constructor libraries

Examples of functionality in a "type constructor library":

- Free functor, free filterable, free applicative, free monad, free etc., generated by a given type constructor `F[_]`
- Church encoding of a free monad ("tagless final")
  - Parameterized by a type constructor `F[_]`
- Church encoding of a recursive type constructor `F[_]` makes it non-recursive
  - Parameterized by `F[_]` and by recursion scheme

Category theory suggests generalizations: e.g. applicative contrafunctor
Category theory motivates consistent rigorous laws for these constructions

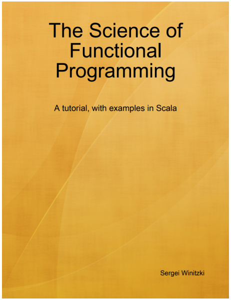
- To derive these laws, need special techniques
- Category theory does not help deriving or proving these laws

A new book: *The Science of Functional programming*

I am working on a new book,
The Science of Functional Programming: A tutorial, with examples in Scala
https://github.com/winitzki/sofp (free as in GNU FSF)

The book will explain (with examples and exercises):

- techniques of reasoning about types and type constructors
- techniques for symbolic calculations with code
- deriving and verifying laws symbolically (as equations for functions)
- real-life motivations for (and applications of) these techniques



Summary

- Practical programmers need to learn functional programming, not category theory
- Category theory helps understand where the typeclass laws came from
  - Category theory *does not* help writing custom typeclass instances
  - Category theory *does not* show how to prove laws
- Category theory motivates general type constructor libraries
  - To prove that the code is correct, one needs to perform symbolic reasoning
  - Mastering the type/code reasoning takes about 6 months of practice
    - and you will be able to use FP much more effectively in actual coding
- Lots of explanations, examples, and exercises in the upcoming book
- Current progress: chapters 1–8 ready
- Chapters 9 and 13 are in progress
- Source (LaTeX) for the book: `https://github.com/winitzki/sofp`