swissQuant
Mastering Complexity

Mathematical Challenge March 2019

Category theory in typed functional programming

Reference

[1] Eilenberg, S.; MacLane S. (1945), 'General Theory of Natural Equivalences' in 'Transactions of the American Mathematical Society, Vol 58. pp 231 294

[2] Wadler, P. (1995), 'Monad for Functional Programming' in 'Advanced Functional Programming', pp 24-55

Description

Introduction

Category Theory (CT) introduced in 1945 [1] is often referred as a general abstraction of function theory. It has applications in theoretical sciences, where is used to map or find equivalences between problems in different fields, which can then be jointly tackled in terms of categorical elements. Programming languages researchers recently realized that most of their problems can profit from being formulated in the higher-level abstractions introduced by CT. This is related with the increased interest for the functional programming paradigm, which emerged in computer science. Indeed, the fundamental property of programming with pure functions is composability, which is also the main focus of CT ("composition of morphisms"). For example, the early version Haskell was a pure functional language without any functionality of reading data from files or reading data from network. [2] Philip Wadler made Haskell more accessible by extending the language with IO type which he modelled with Monads, a widely used element of CT.

Application

The main and high-level goal of FP is to come up with solutions of problems which yield safer, more performant and more scalable results. Some of the solutions developed in a strict FP framework, once matured, were subsequently adopted even by general-purpose languages with the hope of similar results. Typed Functional Programing and its concepts started to be more attractive to programmers with the raise of multicore CPUs because they were providing composable concurrency primitives in a natural way. This is especially useful, in the "Big Data" context, where computations need to be distributed in order to deal with large amount of data.

Using FP concepts can be very powerful in a programming context, especially when the laws governing FP are strictly followed. Below we will introduce some basic concepts from CT and show how they relate to types that are actually used in languages like Haskell. For example, one of the

most popular big data analytics framework (Spark) uses Monoidal (tensor) categories for aggregation in a concurrent manner.

Category theory concepts

A **Category** consists in objects and maps between objects, represented by arrows. **Objects** often correspond to sets equipped with a given algebraic structure. In programming, they usually represent types. **Morphisms (arrows),** on the other side, are usually maps induced by algebraic operations. In programming, they represent functions which are total, deterministic and pure.

A category A:

- A collection $ob(A)$ of objects;
- for each $a, b \in ob(A)$, a collection $\hom_A(a,b)$ of maps or arrows or morphisms from a to b;
- An associative binary operation between morphisms, called composition;
  for each $a, b, c \in \mathrm{ob(A)}$

$$\hom_A(b,c) \times \hom_A(a,b) \rightarrow \hom_A(a,c)$$
$$(g,f) \mapsto g \circ f$$

  and for each $f \in \hom_A(a,b)$, $g \in \hom_A(b,c)$ and $h \in \hom_A(c,d)$, we have
$$(h \circ g) \circ f = h \circ (g \circ f);$$
- for each $a \in ob(\mathcal{A})$, an element 1_a of $\hom_A(a,a)$, called the identity on a, such that $f \circ 1_a = f = 1_b \circ f$, for each $f \in \hom_A(a,b)$.

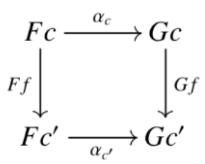
A morphism between categories, called a **Functor**, is a map which preserves categorical structures More precisely, given two categories A and A', a Functor $F: A \rightarrow A'$ is an assignments of objects $F: ob(A) \rightarrow ob(A')$ and morphisms $F: \hom_A(a,b) \rightarrow \hom_{A'}(F(a), F(b))$ such that for each A object *a:* $F(id_a) = id_{F(a)}$ and for each composable pair in A $a \xrightarrow{f} b \xrightarrow{g} c$, $F(g \circ f) = F(g) \circ F(f)$.

There are many known functors that we use in every day programming, like: List, Either, IO, Product and Sum. These functors are usually not implemented in the way we are used to in subtype polymorphism, instead they originated from ad-hoc polymorphism and implemented by type classes. Example with Haskell code:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
instance Functor [] where
  fmap _ [] = []
  fmap f (x:xs) = f x : map f xs
```

Natural transformations are maps associated with pair of functors F,G: C ⇒ D. A natural transformation α: F ⇒ G consists of:

- an arrow α_c : Fc → Gc in D for each object c ∈ C, such that, for any morphism f: c → c' in C, the following square of morphisms in D commutes.



With the **Yoneda Lemma [1]** one can prove that different representations of objects define the same data. Consider the set of morphisms $\hom_C(a,b)$ between *a* and *b* in category *C* and the functor $hom_C(a, \cdot)$. If we collect all natural transformations to the functor F, than we can say that we can get in such way all the structure about *A* in the context of F. In mathematical terms

$$F(A) \cong C(A,\cdot) \xrightarrow{\eta} F$$

In Haskell notation:

$$f\ a \ \cong forall\ x.(a \to x) \to f\ x$$

In programing languages many elements can form categories but we mainly focus on types (and functions). Developers aim to pick the *"right"* abstraction when they model their problems. This is, however, usually encoded in the object level, which makes their combination in an OOP general purpose language difficult. On the other hand, abstractions rooted in CT provide more composable interfaces, which, in typed functional programming language, give more powerful building blocks to reason about in equational and type directed way.

Questions:

- **Q1**: Provide an example of type which is isomorphic to a cartesian product.
- **Q2**: Show that Monads' 2 representations *join*: $forall\ x.M(M\ x) \to M\ x$ and *bind*: $forall\ x\ y.M\ x \to (\mathrm{x} \to \mathrm{M\ y}) \to M\ y$ represent the same type. You might use the Yoneda lemma.
- **Q3**: Identify the natural transformation between List and the Maybe functors, which would retrieve the head of list. You can use the Haskell notation or draw the natural transformation square (commuting) diagram.
- **Q4**: Provide a type from the Functor family which can map predicate functions (a -> Bool)? The functor needs to be able to map functions where the type parameter is in the negative position of argument.

We look forward to your opinions and insights.

Best Regards,

swissQuant Group Leadership Team