CSE 382: Week 03

The Functor Pattern Part 1: map and filter

::

...every sufficiently good analogy is yearning to become a functor.

- John Baez

The Functor Pattern

Functor is probably a word you haven't run into before. In the functional programming world it is used a lot so it would be a good idea for you to start understanding and using it. Now, you could go out and do a lot of googling to try and come up with some definitions of what a functor is with relation to computer science in general and functional programming specifically. Instead of doing so, how about we provided a few definitions used in this course along with the bodies of knowledge the definitions come from. When you feel you're ready to tackle these, go ahead and read them. 🙂

- *Homomorphism*: a transformation of one set, group, category, etc. into another that preserves in the second the relations between elements of the first.
- *Functor*: the transformation from set to set, group to group, category to category, etc. that preserves the relations between the elements of the first set $F :: A \to B$

Alright. So that was tough and probably left you in a little bit of a brain fog. How about splitting them up and taking them one at a time to see if more light can be shed on each of them and the relationships between them? In doing so, let's try to provide you with a high-level understanding of what each of these things are while staying true to their meaning.

Homomorphism (Mathematics)

Homomorphism comes from the Greek *homoios morphe* meaning 'similar form.' So if two sets, groups, trees, or some other type of category of things are homomorphic, then when trying to solve a vexing problem in a specific set, you can convert the problem and set into another set where the problem is easier to solve, solve the problem there, and then convert the solution back to the original set. Then the problem is solved. In problem solving space, this approach is called *transform and conquer*.

Here is an example from the area of the testing of medications. Imagine you have a medication that might cure a specific, common form of cancer. It also might be deadly. How can the medication be tested? One traditional approach is to view mice, rats, and other mammals as homomorphs of humans. Give the drug to these in randomized trials and you can tell if the drug is poisonous to mammals. Unlike Mathematics and Computer Science, this approach doesn't guarantee the drug isn't poisonous to humans, these animals not being exact homomorphs for humans, but it does greatly reduce the probability that the drug would kill people.

Functor (Mathematics)

The word functor comes from the word function. It was adopted in the 1930's by mathematicians who study Category Theory to express the idea of a function that is a mapping of one category to another.

Here is an example of how functors work in Category Theory. Consider this situation. The Computer Science Department has reworked the CS degree. In doing so, a bunch of new classes have been created, most old ones have been killed, and a few carry over from the old 2020-2021 catalog to the new 2021-2022 one.

When making this kind of a change, it is very important for continued student and program success that the prerequisite structure of the courses of the old catalog be reflected in the courses in the new catalog. If not, student progress will be stymied and graduations delayed. Let's call the process of mapping the old courses to the new and the relationships between the old courses to the relationships between the new courses OldCatalogToNewCatalog.

From a mathematician's perspective OldCatalogToNewCatalog is a functor.

Here is another more 'mathy' example. Consider the sets in Figure 1.

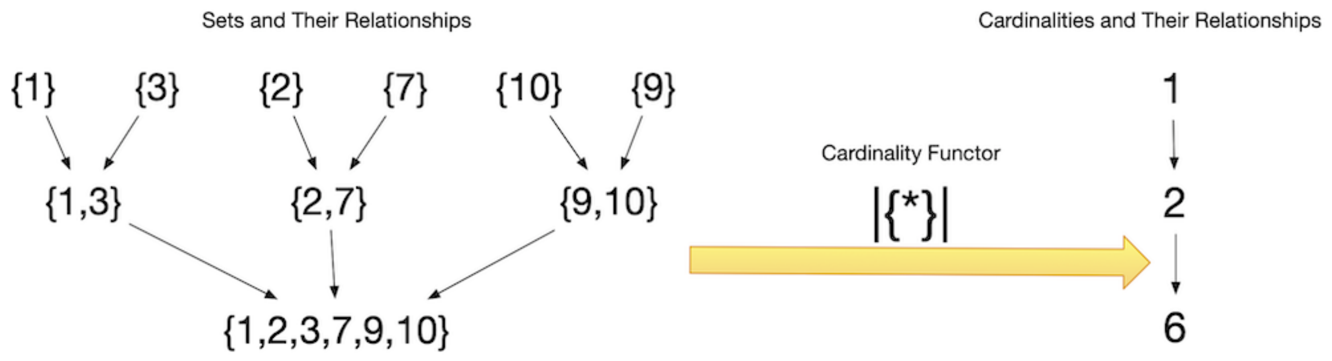

Fig.1 - Trulli, Puglia, Italy.

Notice the relationships between the sets and the relationships between the cardinalities. The cardinality functor has preserved the relationships in both cases and mapped the sets to the appropriate numeric value.

Functor (Functional Programming)

The definition of Functor in Functional Programming and other types of computer languages is inspired by, but not the same as, that used in Category Theory. There are subtle differences between the two, but that is beyond the scope of this class. If you are interested in a mathematical perspective on functional programming functors, there is a short succinct statement on stackExchange and another math heavy, detailed explanation.

Here is the takeaway from this controversy. Just because you understand what a functor is in functional programming, don't go thinking you know what a functor in Category Theory is. You probably don't. If you want to know that, go study Category Theory. It will open your mind to a very different way of thinking.

In this course we will use the re-imagined functional programming definition of Functor. For this definition, there are two laws that must be true in order for something to be a functor. The first is called the identity law. Essentially it means that the functor does not use or produce any side effects. The second is the composition law, $f(g \circ h) = f(g) \circ f(h)$. To be a functor, when the functor applies to the composition of one or more functions the result must be the same as the functor being applied to each function separately. At this point in time, just be aware of these laws. You will see them described more formally at the end of the reading for week 04.

Here is a non-computing example of how functional programming functors work. Consider this situation. Alice is a Chemistry major. Or another way to say this is she is in the Chemistry major category. Alice goes to a party designed to help people from different colleges interact and get to know each other. If she meets another Chemistry major, her interactions with that person will probably be the same as they have always been with other Chemistry majors.

However, at the party Alice meets an Art major named Stan. Even though Alice has not changed, she is after all still herself, she will have to interact with Stan in a different way. She will attempt to modify the ways she interacts with other Chemistry majors in order to interact with Stan. When she does this, she is applying an AtParty functor.

Map

An application of our functor definition you are familiar with is the `map` BIF (Built In Function). In `map`, a list and a lambda function used to modify each element of the list are passed as `map's` two parameters. The value of the map functor is a new list containing modified versions of each element of the original list as you see in the first line of Figure 1.

Fig.2 - The declaration and definition of the map functor.

$$\begin{aligned} &map :: [a]\ (\lambda :: a \to b) \to [b] \\ &map :: [a]\ (\lambda :: a \to b) \to [\,],\ \text{when } [a] \text{ is empty ;} \\ &map :: [h \mid t]\ (\lambda :: a \to b) \to \\ &\qquad [\lambda\ h] : [map\ t\ (\lambda :: a \to b)]\ \text{otherwise.} \end{aligned}$$

When you used the map BIF before you probably used it to add, subtract, multiply, or divide some list of numbers or something similar to that. You saw above that functors are applied to functions. There seems to be a conflict between the way you've used map and the definition but there isn't. All you need to remember is variables can be viewed as being functions that always return the same result. After all, for $f(x) = 1, 1 = f(n)$.

Being applied in a Functional Programming space, the map functor is defined recursively. If the list-type parameter of map is empty, the value of the map functor is an empty list (the first function clause in Figure 1). If the list-type parameter is non-empty, the head of the list is passed to the lambda-type parameter. The value of the lambda with the head as its parameter is then prepended to a recursive call to map itself with the tail of the list-type parameter passed instead of the complete list (the second function clause of Figure 1).

Thus, each element of the original list is converted to a new element, possibly even a new type, and then added to a distinct list that is created as the value of the call to the map functor.

Filter

The `filter` BIF is significantly different. Instead of a 1-to-1 correspondence between the list passed as a parameter and the list that is the value of `filter`, the value list can, but does not have to, have fewer elements. The value list contains only those elements of the original list that cause the lambda parameter to return $true$.

Fig.3 - The declaration and definition of the filter functor.

$$\begin{array}{l} filter :: [a]\ (\lambda :: a \to Boolean) \to [b] \\ filter :: [a]\ (\lambda :: a \to Boolean) \to [\,] \quad when\ [a] \text{ is empty ;} \\ filter :: [h \mid t]\ (\lambda :: a \to Boolean) \to \\ \qquad [h] +\!\!+\ filter\ t\ \lambda \quad when\ \lambda\ h = true \\ \qquad filter\ t\ \lambda \quad when\ \lambda\ h = false. \end{array}$$

The filter functor is defined recursively. If the list-type parameter of map is empty, the value of the map functor is an empty list (the first clause of the function in Figure 2). If the list-type parameter is non-empty (the second clause of the function in Figure 2), the head of the list is passed to the lambda-type parameter, and if the value of the lambda is then true, the head is prepended to a recursive call to filter itself with the tail of the list-type parameter passed instead of the complete list .

Thus, each element of the original list that matches the logic specified in the lambda is added to a distinct list that is created as the value of the call to the filter functor.

Application

Consider this situation. You have been asked to write a small section of a larger app for the company you work for. The data you are given will be a list of tuples. Each tuple describes a person, stored in our database, who has previously applied to work for the company. The list of candiates available to your computation, is sorted, newest to oldest, based on when they submitted their application.

Each person is a 3-tuple of shape, $\{name, years\_experience, [skill]\}$, with the third element is a list of skills. The end product of your computations needs to be a list of potential employees. Each of these is a 4-tuple of shape $\{name, front\_or\_backend, years\_experience, [skill]]\}$. Those with Erlang experience have $backend$ as the second element of the 4-tuple. Those with JavaScript experience but no Erlang experience have $frontend$ as the second element of the 4-tuple. Also, if the applicant doesn't have JavaScript or Erlang experience, they are not to be included in the end product of your calculations.

In this example, there is are two category changes. The initial category is $applicants$. The second category, after the filtering, is $beginners$. The final category is $potential\_employees$. The relations between the elements, the years of experience of each applicant, is also preserved.

Thnk about the $f(g \circ h) = f(g) \circ f(h)$ rule for functors. Let either map or $filter$ be f in the rule. You could use $filter$ multiple times filtering out elements by one property of each list element every time you filtered. You could also filter once based on multiple properties of each list element. Considering map, you could use map multiple times to make multiple changes, or you could use map once to make multiple changes.