

Category Theory in Type Systems

AKA: BUCKLE YOUR SEATBELTS BECAUSE IN THREE SHORT MINUTES I AM GOING TO LEARN YOU A THING I ONLY LEARNED MYSELF HALF AN HOUR AGO.

Jonathan Hayase

April 24, 2018

Math 171 – Abstract Algebra – Spring 2018

Review: Categories

Recall: Informally, a category \mathcal{C} consists of

1. class $\text{ob}(\mathcal{C})$ of **objects**;
2. class $\text{hom}(\mathcal{C})$ of **arrows** $\phi : A \rightarrow B$ for objects A and B ;
3. a “well behaved” composition operation \circ on arrows.

Review: Functors

Recall: Informally, a functor* F from \mathcal{A} to \mathcal{B}

1. assigns every object in \mathcal{A} to an object in \mathcal{B} ,
2. assigns every arrow in \mathcal{A} to an arrow in \mathcal{B}

such that domains, codomains, compositions, and identities are preserved.

*For simplicity, I am only covering covariant functors in this presentation.

Category Theory of Type Systems

In programming, type systems have a natural interpretation as a category!

Let the \mathcal{T} be the category of types in a programming language L .

1. The objects of \mathcal{T} are **types** (i.e. integers, floats, bools).
2. The arrows of \mathcal{T} are **functions** which map values of one type to those of another.
3. The operation \circ is just regular function composition.

Functions as composable arrows

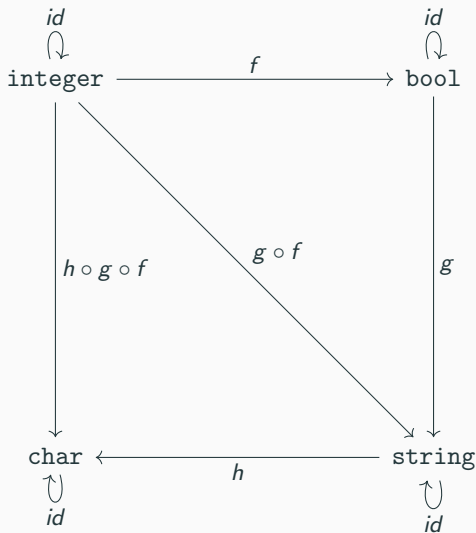


Figure 1: A commutative diagram of three functions: f , g , and h .

So now what?

- So far, we've only discussed “primitive” types.
- What about lists, sets, matrices, etc.?
- For now, consider `list<E>`, a type representing ordered collections of objects of type `E`.

`list` is a functor.

1. `list` is an endofunctor (i.e. it maps \mathcal{T} to itself).
2. It's pretty clear to see that `list` maps a type $E \in \mathcal{T}$ to the list type containing elements of type E , also in \mathcal{T} .
3. But what do we assign to the arrows of \mathcal{T} ?

“Mapping”

Q: Given an arrow (i.e. function) from type A to type B, how can we define a function from `list<A>` to `list`?

[†]Not to be confused with mapping in mathematics.

“Mapping”

Q: Given an arrow (i.e. function) from type A to type B, how can we define a function from `list<A>` to `list`?

A: By “mapping”[†]

```
map(int, ["1", "2", "3"]) == [1, 2, 3]
```

In this example, given `int : string → integer` we can define a new arrow `mapint : list<string> → list<integer>`.

[†]Not to be confused with mapping in mathematics.

Functors are everywhere!

- Collection types such as lists, vector, matrices, etc.
- Nullable types like `std::optional` in C++, `Maybe` in Haskell, and `T?` in C#.
- And more!

Just the beginning...
