CHEAT SHEET ON CATEGORIES, FUNCTORS AND NATURAL TRANSFORMATIONS

RICHARD GARNER

This is a ready reference for some of the basic definitions we will talk about in the Sydney Type Theory meetup on 17th July 2017.

1. Categories

A *category* $\mathcal{C}$ comprises the following data:

- A collection of *objects* $X, Y, Z, \ldots$;
- For each pair of objects, a collection $\mathcal{C}(X, Y)$ of *morphisms* (or *arrows*) from X to Y;
- For each object X, an *identity morphism* $1_X \in \mathcal{C}(X, X)$; and
- For each triple of objects X, Y, Z, a *composition* operation

$$\begin{aligned} \mathcal{C}(Y, Z) \times \mathcal{C}(X, Y) &\to \mathcal{C}(X, Z) \\ (g, f) &\mapsto g \circ f \ ; \end{aligned}$$

subject to the following conditions:

- *Associativity.* For any $f \in \mathcal{C}(X, Y)$, $g \in \mathcal{C}(Y, Z)$ and $h \in \mathcal{C}(Z, W)$, we have

$$(h \circ g) \circ f = h \circ (g \circ f) \qquad \text{in } \mathcal{C}(X, W).$$

- *Unitality.* For any $f \in \mathcal{C}(X, Y)$, we have

$$f \circ 1_X = f = 1_Y \circ f \qquad \text{in } \mathcal{C}(X, Y);$$

Example 1. There is a category $\mathcal{C}$ whose objects are Haskell types `A`, `B`, `C`, ... and whose morphisms from `A` to `B` are Haskell functions of type `A -> B`. The identity on `A` is the program `\x -> x`. The composite of `f :: A -> B` and `g :: B -> C` is `\x -> f (g  x)`.

This example should be approached with caution. For example, there should certainly be a morphism `[Nat] -> [Nat]` in $\mathcal{C}$ which sorts a list of natural numbers. But do we want `quicksort` and `mergesort` to be two different morphisms, or the same? How about two trivially different implementations of `quicksort`—should we distinguish those? In other words, the morphisms of this category $\mathcal{C}$ are more correctly defined to be Haskell functions *identified up to some suitable notion of program equivalence* $\equiv$. It's up to you to decide what $\equiv$ should be:

- A maximal identification (*fully extensional*) would be to identify any two programs that return equal values given equal inputs; so we don't care at all about algorithmics.

- At the other extreme (*fully intensional*) we could declare that morphisms literally are the textual code of Haskell functions of the appropriate type, without any identification.
- The real fun begins if you are interested in constructing an intermediate notion of program equivalence—which amounts to crystallising some vague intuition about what we actually mean by an algorithm.

In the meetup I will not address these subtle points, and henceforth we will take the simple-minded, fully extensional approach.

Example 2. There is a category $\mathcal{E}$ whose objects are Haskell types `A` equipped with an implementation of the typeclass `Eq`. Morphisms from `A` to `B` are Haskell functions `f :: A -> B` identified up to program equivalence $\equiv$ as before, which in addition respect equality, in the sense that

```
\x y -> x == y      ≡      \x y -> (f x) == (f y)
```

as functions `A -> A -> Bool`. You can imagine constructing a similar category out of any other typeclass of your choice.

2. Functors

If $\mathcal{C}$ and $\mathcal{D}$ are categories, then a *functor* $F\colon \mathcal{C} \to \mathcal{D}$ comprises the following data:

- An operation assigning to each object X of $\mathcal{C}$, an object FX of $\mathcal{D}$; and
- For each pair of object X, Y of $\mathcal{C}$, an operation

$$\mathcal{C}(X,Y) \to \mathcal{D}(FX, FY)$$
$$f \mapsto Ff\ ,$$

subject to the following "functoriality" conditions:

- *Preservation of identities.* For any object X of $\mathcal{C}$, we have
$$F(1_X) = 1_{FX} \qquad \text{in } \mathcal{D}(FX, FX);$$
- *Preservation of composition.* For any $f \in \mathcal{C}(X,Y)$ and $g \in \mathcal{C}(Y,Z)$, we have
$$F(g \circ f) = Fg \circ Ff \qquad \text{in } \mathcal{D}(FX, FZ).$$

Example 3. Let $\mathcal{C}$ be the category of Haskell types and programs. There is a functor `[ ]` $: \mathcal{C} \to \mathcal{C}$ which:

- On objects sends a type `A` to the type `[A]` of lists of `A`'s;
- On morphisms sends `f :: A -> B` to `(map f) :: [A] -> [B]`.

Functoriality requires that `(map id)` $\equiv$ `id`, and that `(map g) o (map f)` $\equiv$ `map (g o f)`; of course, if we want this to really be true, we had better choose our notion of program equivalence $\equiv$ appropriately. The fully extensional choice is in particular appropriate.

Example 4. More generally, any implementation of the Haskell `Functor` typeclass *should* yield a functor $\mathcal{C} \to \mathcal{C}$. More precisely, if `F :: * -> *` such that `Functor F`, then we define our functor $\mathcal{C} \to \mathcal{C}$ on objects by sending `A` to `F A` and on morphisms by sending `f :: A -> B` to `fmap f :: F A -> F B`. I can only say that this *should* yield a functor since Haskell has no facility for ensuring the functoriality conditions.

Example 5. Let $\mathcal{C}$ be the category of Haskell types, and $\mathcal{E}$ the category of Haskell types implementing `Eq`. There is a functor $U\colon \mathcal{E} \to \mathcal{C}$ which forgets the implementation.

Example 6. We can take the functor $U\colon \mathcal{E} \to \mathcal{C}$ of the preceding example and follow it by the functor `[ ]` $: \mathcal{C} \to \mathcal{C}$ to get a new functor $F\colon \mathcal{E} \to \mathcal{C}$ which:

- On objects, sends a type `A` equipped with an implementation of `Eq` to the type `[A]` of lists of `A`'s;
- On morphisms, sends an equality-respecting function `f :: A -> B` to the function `map f :: [A] -> [B]`.

3. Natural transformations

If $F, G\colon \mathcal{C} \to \mathcal{D}$ are functors, then a *natural transformation* $\alpha\colon F \Rightarrow G$ is given by the following data:

- For each object X of $\mathcal{C}$, a morphism $\alpha_X \in \mathcal{D}(FX, GX)$

subject to the following *naturality* condition:

- For each morphism $f \in \mathcal{C}(X, Y)$, we have that

$$Gf \circ \alpha_X = \alpha_Y \circ Ff \qquad \text{in } \mathcal{D}(FX, GY).$$

Example 7. Consider again the functor `[ ]` $: \mathcal{C} \to \mathcal{C}$ on the category of Haskell types. There is a natural transformation $\rho\colon$ `[ ]` $\Rightarrow$ `[ ]` for which

$$\rho_{\texttt{A}} = \texttt{reverse :: [A] -> [A]}$$

The naturality condition expresses the fact that, for any `f :: A -> B`,

```
reverse o (map f) ≡ (map f) o reverse
```

as functions `[A] -> [B]`.

Example 8. Consider the functor $F\colon \mathcal{E} \to \mathcal{C}$ of Example 6. There is a natural transformation $\nu\colon F \Rightarrow F$ for which

$$\nu_{\texttt{a}} = \texttt{nub :: [A] -> [A]}\ .$$

Naturality is now the condition that, whenever `f :: A -> B` satisfies

```
\x y -> x == y     ≡     \x y -> (f x) == (f y)
```

we also have that

```
nub o (map f) = (map f) o nub
```

as functions `[A] -> [B]`.

Note that these examples both involve polymorphic functions (`reverse` and `nub`). The basic idea is that any polymorphic function should give you a natural transformation. The fundamental reason for this is explained in a paper by Philip Wadler titled "Theorems for free!". I will not try to explain this reasoning in the meetup, but I am going to try and explain how we can use the idea that "polymorphism = naturality" to solve problems like: *give a classification of all functions of polymorphic type* `[a] -> [a]`.

Department of Mathematics, Macquarie University, NSW 2109, Australia
E-mail address: richard.garner@mq.edu.au