Contents

These notes serve the following purpose. Firstly, they list and summaries important concepts in Category Theory and how they can be expressed in Haskell. Secondly, the book content is compared to other books I am reading in parallel.

In composing this text I will follow these rules: - Notes: are for references to other books and papers - Bold font: is for definitions and questions of challenges - Italic: If things that still need to be defined appear in text

Table of content

0 Preface

11 Declarative Programming

---

12 Limits and Colimits

As Bartosz writes "in category theory everything is related to everything and everything can be viewed from many angles", and thus whenever a new concept is introduced it can be linked in different ways to various previously introduced concepts. In reading Bartosz's blog post I found it very helpful to read Section 8.5 of [CH] along side it.

This chapter advances the subjects of *products*, *coproducts*, *terminal* and *initial object* that were introduced in Section 5. Here, we will simplify and generalise the universal property of the product and coproduct through functors and natural transformations, and see reveal the deeper relation with terminal and initial objects.

Remember the definition of a product, which says that, given any pair of maps $(f : C \to X, g : C \to Y)$, there exists a unique map $C \to X \times Y$, such that certain diagrams commute. Such pair of maps is at the heart of unifying terminal objects, products of sets, preorders, categories and more. It therefore deservers another name, $\mathbf{Cone}(X, Y)$, inspired by the diagram such as Figure 5-5-1. Also, remember the definition of a diagram as given in Section 1-1.

> [SSC], Definition 3.77, page 112: Cone and Limits
>
> Let $D : \mathcal{J} \to \mathcal{C}$ be a diagram. A *cone* is $(\mathcal{C}, c_*)$ over D consists of i) an object $C \in \mathcal{C}$, ii) for each object $j \in \mathcal{J}$, a morphism $c_j : C \to D(j)$. To be a cone, these must satisfy the following property: for each $f : j \to k$ in $\mathcal{J}$, we have $c_k = D(f) \circ c_j$.
>
> A *morphism of cones* (C, c_*) is a morphism $a : C \to C'$ in $\mathcal{C}$ such that for all $j \in \mathcal{J}$ we have $c_j = c'_j \circ a$. Cones over D, and their morphisms, form a category $\mathbf{Cone}(D)$.
>
> The *limit* of D, denoted as $\lim D$, is the terminal object in the category $\mathbf{Cone}(D)$. Say it is the cone $\lim D = (C, C_*)$; we refer to C as the *limit object* and the map c_j for any $j \in \mathcal{J}$ as the jth *projection map*.

To read more on this in an applied context see [SSC] Chapter 3 on databases.

Why does [PC] not have anything on limits?

More notes in: - [SSC], Section 3.5, page 108 - [CH], Section 8.5, page 89

12-1 Limit as a Natural Isomorphism

12-2 Examples of Limits

Three important examples of limits are: - product - equalizer - pullback

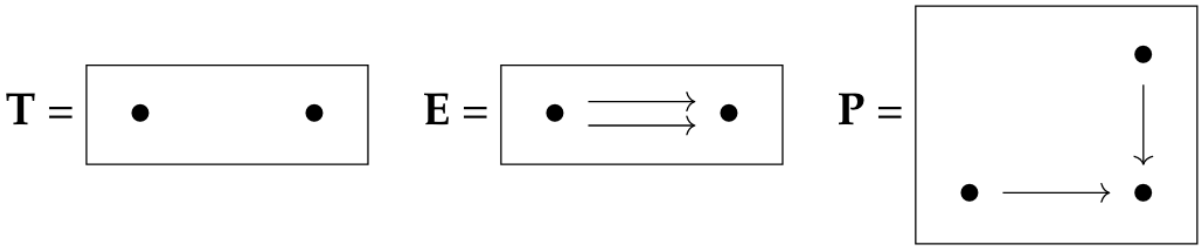

Figure 1: Figure 12-1-1: Diagrams of product, equalizer, and pullback”

More notes in:

- [SSC], Section 3.5, page 108
- [CH], Section 8.5, page 89

12-3 Colimits

When you invert the direction of all arrows in a cone, you get a co-cone, and the universal one of those is called a colimit.

The dual of the pullback is called the pushout. It's based on a diagram called a span, generated by the category 1 <- 2 -> 3.

12-4 Continuity

The actual definition of a *continuous functor* $\mathcal{F}$ from a category $\mathcal{C}$ to $\mathcal{C}'$ includes the requirement that the functor preserve limits. Continuous functors are of most relevant for topological spaces

The continuous functor is defined in Definition 7.17 in [SSC], page 232, but through a discussion of *topological spaces*

12-5 Challenges

1. How would you describe a pushout in the category of C++ classes?

Skip

2. Show that the limit of the identity functor is the initial object.

> Answers: - Theorem 3.3 in https://ncatlab.org/nlab/show/initial+object - https://math.stackexchange.com/questions object-is-limit-of-identity-functor-converse

3. Subsets of a given set form a category. A morphism in that category is defined to be an arrow connecting two sets if the first is the subset of the second. What is a pullback of two sets in such a category? What's a pushout? What are the initial and terminal objects?

4. Can you guess what a coequalizer is?

> Answers: - [SSC], Example 6.37, page 193 - https://ncatlab.org/nlab/show/coequalizer - https://en.wikipedia.org/wiki/Coequalizer

Remark 2.3. By formal duality, a coequalizer in $\mathcal{C}$ is equivalently an equalizer in the opposite category $\mathcal{C}^{\mathrm{op}}$.

5. Show that, in a category with a terminal object, a pullback towards the terminal object is a product.

6. Similarly, show that a pushout from an initial object (if one exists) is the coproduct.

> Answers: - [SSC], Proposition 6.28, page 191

13 Free Monoids

This chapter continuous our discussion of monoids which we started in Sec. 3-4 & 3-5.

13-1 Free Monoid in Haskell

```haskell
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
```

As mentioned already twice in Part 1, unit and associativity laws cannot be expressed in Haskell and must be verified by the programmer every time a monoid is instantiated.

The fact that a list of any type forms a monoid is described by this instance definition!

```haskell
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

13-2 Free Monoid Universal Construction

From many categories representing 'sets with added structure' (groups, monoids, vector spaces, rings, topological spaces, ...) there is a *forgetful functor* going to $\mathbf{Set}$, where objects are sent to their underlying sets.

As an additional example, there is also a forgetful functor $F : \mathbf{Cat} \to \mathbf{Grph}$, sending each category to the graph defined by its objects and arrows.

A functor between monoids is called a monoid homomorphism.

> [PC], Example 3.4, page 68: Monoid homomorphis
>
> Consider the monoids $\mathcal{Z}_\times(\mathcal{Z}, 1, \times)$ and $\mathcal{B}_{\text{AND}}(\mathcal{B}, \text{true}, \text{AND})$. Let `is_odd`: $\mathcal{Z} \to \mathcal{B}$ be the function that sends odd numbers to **`true`** and even numbers to **`false`**. This is a monoid homomorphism. It preserves identities because 1 is odd, and it preserves composition because the product of any two odd numbers is odd, but the product of anything with an even number is even.

More on *monoid homomorphisms*:

- [PC], Sec. 3.2.2, page 68

13-3 Challenges

1. You might think (as I did, originally) that the requirement that a homomorphism of monoids preserve the unit is redundant. After all, we know that for all `a` we have `h a * h e = h (a * e) = h a`. So `h e` acts like a right unit (and, by analogy, as a left unit). The problem is that `h a`, for all `a` might only cover a sub-monoid of the target monoid. There may be a "true" unit outside of the image of `h`. Show that an isomorphism between monoids that preserves multiplication must automatically preserve unit.

2. Consider a monoid homomorphism from lists of integers with concatenation to integers with multiplication. What is the image of the empty list `[]`?

We understand "image" as being a, e.g., objects, set, etc. in another category into which it is brought through a functor.

This question connects to the example mentioned on page 215:

```
[2] ++ [3] = [2, 3]
2*3=6
```

Answer: the image of `[]` is 0.

Assume that all singleton lists are mapped to the integers they contain, that is `[3]` is mapped to `3`, etc. What's the image of `[1, 2, 3, 4]`?

$$1 \times 2 \times 3 \times 4 = 24$$

**How many different lists map to the integer 12?

$$1 \times 12 = 12$$

$$1 \times 2 \times 6 = 12$$

$$1 \times 3 \times 4 = 12$$

Leading to the following lists on the codomain of the monoidal homomorphism

```
[12]
[2,6]
[3,4]
```

```
[1,12]
[1,2,6]
[1,3,4]
```

Is there any other homomorphism between the two monoids?

The two monoids are: a) lists of integers with concatenation `M(Int, , ++)` b) integers with multiplication `M(Int, , *)`

3. What is the free monoid generated by a one-element set? Can you see what it's isomorphic to?

14 Representable Functors

14-1 The Hom Functor

14-2 Representable Functors

More reading:

- [PC], Section 3.6, page 92

14-3 Challenges

1. Show that the hom-functors map identity morphisms in C to corresponding identity functions in $\mathbf{Set}$.

2. Show that `Maybe` is not representable.

3. Is the `Reader` functor representable?

4. Using `Stream` representation, memoize a function that squares its argument.

5. Show that `tabulate` and `index` for `Stream` are indeed the inverse of each other. (Hint: use induction.)

15 The Yoneda Lemma

15-3 Challenges

16 Yoneda Embedding

16-5 Challenges

References

[CM] ‘Conceptual Mathematics’ by F. William Lawvere and Stephen H. Schanuel

[SSC] ‘An Invitation to Applied Category Theory: Seven Sketches in Compositionality’ by Brendan Fong and David I. Spivak

[PC] ‘Programming with Categories’ by Brendan Fong, Bartosz Milewski, David I. Spivak

[CH] https://github.com/jwbuurlage/category-theory-programmers by Jan-Willem Buurlage

[RWH] ‘Real World Haskell’ by Bryan O’Sullivan, Don Stewart, John Goerzen; 2008

[HPFP] ‘Haskell Programming from First Principles’ by Christopher Allen and Julie Moronuki; 2016

[WIWIK] ‘What I Wish I Knew When Learning Haskell’ by Stephen Diehl; 2020