Category theory for programmers

An introduction to the mathematics of functional programming

Jan-Willem Buurlage

Contents

Introduction

This document contains notes for a small-scale seminar on category theory in the context of (functional) programming, organized at Centrum Wiskunde & Informatica, the national Dutch research centre for mathematics and computer science. The goal of the seminar is to gain familiarity with concepts of category theory (and other branches of mathematics) that apply (in a broad sense) to the field of functional programming.

Although the main focus is on the mathematics, examples are given in Haskell to illustrate how to apply the concepts. In some places, examples are given in other languages as well (such as Python and C++).

I would like to thank:

- Tom Bannink for supplying the proof for the bifunctor example.
- Peter Kristel for valuable comments on the Yoneda embedding
- Willem Jan Palenstijn for corrections and comments regarding cartesian closed categories.
- Tom de Jong for examples and suggestions for the section on adjunctions
- Edward Kmett for examples of Monads arising from adjunctions in Haskell.

– Jan-Willem Buurlage (janwillembuurlage@gmail.com)

Preliminaries

Today, the most common programming style is *imperative*. Imperative programming lets the user describes *how* a program should operate, mostly by directly changing the memory of a computer. Most computer hardware is imperative; a processor executes a machine code sequence, and this sequence is certainly imperative. This is originally described by mathematicians such as Turing and von Neuman in the 30s.

A different way of programming is *declarative programming*, which is a way of expressing *what* you want the program to compute (without explicitely saying how it should do this). A good way of expressing what you want to have computed, is by describing your program mathematically, i.e. *using functions*, which is what we explore here. This functional style of looking at computations is based on work in the 20s/30s by Curry and Church among others.

The difficulty in using a *(typed, pure) functional* programming language, is that the **functions that you write** between types **should behave like mathematical functions** on the corresponding sets. This means, for example, that if you call a function multiple times with the same arguments, it should produce the same result every time. This is often summarized as a *side-effect free function*. Other difficulties are that values are in principle immutable.

Something else that would allow us to more accurately describe our programs in a mathematical way is if execution is *lazy*, and Haskell indeed is lazy. This means we can work with **infinite lists and sequences**, and only peeking inside such as a list causes the necessary computations to be done (or 'collapses the wave function' if you want a quantum analogy).

Part I

Basic theory

Chapter 1

Categories, functors and natural transformations

1.1 Core definitions

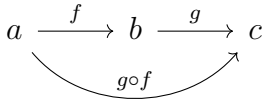
We start with giving the definition of a category:

Definition 1.1. A **category** $\mathcal{C} = (O, A, \circ)$ consists of:

- a collection O of *objects*, written $a, b, \ldots \in O$.
- a collection A of *arrows* written $f, g, \ldots \in A$ between these objects, e.g. $f : a \to b$.
- a notion of *composition* $f \circ g$ of arrows.
- an identity arrow id_a for each object $a \in O$.

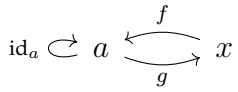
The composition operation and identity arrow should satisfy the following laws:

- *Composition*: If $f : a \to b$ and $g : b \to c$ then $g \circ f : a \to c$.



- *Composition with identity arrows*: If $f : x \to a$ and $g : a \to x$ where x is arbitrary, then:

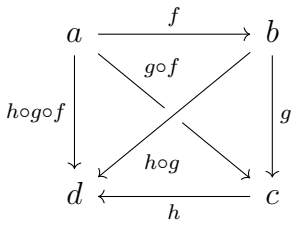
$$\mathrm{id}_a \circ f = f, \ g \circ \mathrm{id}_a = g.$$



- *Associativity*: If $f : a \to b$, $g : b \to c$ and $h : c \to d$ then:

$$(h \circ g) \circ f = h \circ (g \circ f).$$

This is the same as saying that the following diagram commutes:



Saying a diagram commutes means that for all pairs of vertices a' and b' all paths from between them are equivalent (i.e. correspond to the same arrow of the category).

If $f : a \to b$, then we say that a is the *domain* and b is the *codomain* of b. It is also written as:

$$\mathrm{dom}(f) = a, \ \mathrm{cod}(f) = b.$$

The composition $g \circ f$ is only defined on arrows f and g if the domain of g is equal to the codomain of f.

We will write for objects and arrows respectively simply $a \in \mathcal{C}$ and $f \in \mathcal{C}$, instead of $a \in O$ and $f \in A$.

Examples of categories

Some examples of familiar categories:

Name	Objects	Arrows
Set	sets	maps
Top	topological spaces	continuous functions
Vect	vector spaces	linear transformations
Grp	groups	group homomorphisms

In all these cases, arrows correspond to functions, although this is by no means required. All these categories correspond to objects from mathematics, along with *structure preserving maps*. **Set** will also play a role when we discuss the category **Hask** when we start talking about concrete applications to Haskell.

There are also a number of simple examples of categories:

- **0**, the empty category $O = A \equiv \emptyset$.
- **1**, the category with a single element and (identity) arrow:

$$\mathrm{id}_a \circlearrowright a$$

- **2**, the category with a two elements and a single arrow between these elements

$$\mathrm{id}_a \circlearrowright a \xrightarrow{f} b \circlearrowright \mathrm{id}_b$$

- $\rightrightarrows$: the category with two elements and two parallel arrows between these elements:

$$\mathrm{id}_a \circlearrowright a \rightrightarrows b \circlearrowright \mathrm{id}_b$$

From now on we will sometimes omit the identity arrows when drawing categories.

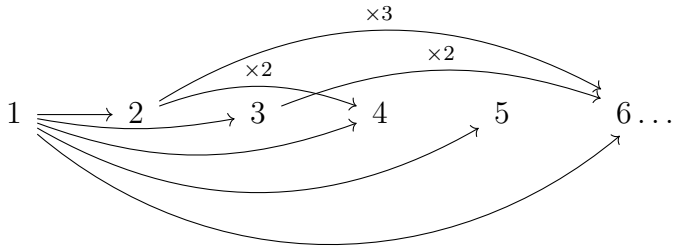
- Another example of a category is a *monoid category*, which is a specific kind of category with a single object.

  Definition 1.2. A *monoid* $(M, \cdot, e)$ consists of:
  - a set M
  - an associative binary operation $(\cdot) : M \times M \to M$
  - a unit element w.r.t $(\cdot)$, i.e. $\forall_m \; e \cdot m = m$

  Indeed, it is a group structure without requirement of inverse elements. It is also called a *semi-group with unit*)

  This corresponds to a category $\mathcal{C}(M)$ where:
  - There is a single object (for which we simply write M)

  - There are arrows $m : M \to M$ for each element $m \in M$.
  - Composition is given by the binary operation of the monoid: $m_1 \circ m_2 \equiv m_1 \cdot m_2$.
  - The identity arrow id_M is equal to e, the unit of the monoid.
- We can also consider natural numbers $\mathbb{N}_{>0}$, with arrows going from each number to its multiples.



- A partially ordered set (poset): a binary relation $\leq$ over a set S s.t. for $a, b, c \in S$:

  - $a \leq a$
  - $a \leq b,\ b \leq a \implies a = b$
  - $a \leq b,\ b \leq c \implies a \leq c$

  also corresponds to a category.

1.2 Functors

A functor is a map between categories. This means it sends objects to objects, and arrows to arrows.

Definition: A *functor* T between categories $\mathcal{C}$ and $\mathcal{D}$ consists of two functions (both denoted simply by T):

- An *object function* that maps objects $a \in \mathcal{C}$: $a \mapsto Ta \in \mathcal{D}$
- An *arrow function* that assigns to each arrow $f : a \to b$ in $\mathcal{C}$ an arrow $Tf : Ta \to Tb$ in $\mathcal{D}$, such that:

$$T(\mathrm{id}_a) = \mathrm{id}_{Ta},\ T(g \circ f) = Tg \circ Tf.$$

A functor is a very powerful concept, since it allows you to translate between different branches of mathematics! They also play an important role in functional

programming. Where among many other things, they are useful for defining the *container types* or more generally *type constructors*.

Functors can be composed, and this allows one to define a category of categories[1] **Cat**, where the arrows are functors.

Examples of functors

- The identity functor: $\mathrm{id}_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$ is defined as:

$$\begin{aligned} \mathrm{id}_{\mathcal{C}} : \ & a \mapsto a \\ & f \mapsto f \end{aligned}$$

- The constant functor $\Delta_d : \mathcal{C} \to \mathcal{D}$ for fixed $d \in \mathcal{D}$:

$$\begin{aligned} \Delta_d : \ & a \mapsto d \\ & f \mapsto \mathrm{id}_d \end{aligned}$$

- The *power-set functor*: $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ sends subsets to their image under maps. Let $A, B \in \mathbf{Set}$, $f : A \to B$ and $S \subset A$:

$$\begin{aligned} & \mathcal{P}A = \mathcal{P}(A), \\ & \mathcal{P}f : \mathcal{P}(A) \to \mathcal{P}(B), \ S \mapsto f(S) \end{aligned}$$

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

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

- Dual-set functor

$$\begin{aligned} * : & \ \mathbf{Vect} \to \mathbf{Vect} \\ & : W \mapsto W^* \\ & : (f : V \to W) \mapsto (f^* : W^* \to V^*) \end{aligned}$$

  This is an example of a *contravariant functor* (a functor from **Vect** to $\mathbf{Vect}^{\mathrm{op}}$, the category with reversed arrows and composition rules.

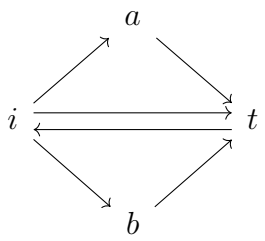
[1] Actually, there are some technicalities to be worked out and the resulting category consists of 'small categories' only.

1.3 Special objects, arrows and functors

Special objects

For objects, we distuinguish two special kinds:

Definition 1.3. An object $x \in \mathcal{C}$ is **terminal** if for all $a \in \mathcal{C}$ there is exactly one arrow $a \to x$. Similarly, it is **initial** if there is exactly one arrow $x \to a$ to all objects.



Here, i is initial, and t is terminal.

Special arrows

There are a number of special kind of arrows:

Definition 1.4. An arrow $f : a \to b \in \mathcal{C}$ is a **monomorphism** (or simply mono), if for all objects x and all arrows $g, h : x \to a$ and $g \neq h$ we have:

$$f \circ g \neq f \circ h.$$

To put this into perspective, we show that in the category **Set** monomorphisms correspond to injective functions;

Theorem 1.5. In **Set** a map f is mono if and only if it is an injection.

Proof. Let $f : A \to B$. Suppose f is injective, and let $g, h : X \to A$. If $g \neq h$, then $g(x) \neq h(x)$ for some x. But since f is injective, we have $f(g(x)) \neq f(h(x))$, and hence $h \circ f \neq h \circ f$, thus f is mono.

For the contrary, suppose f is mono. Let $\{*\}$ be the set with a single element. Then for $x \in A$ we have an arrow $\{*\} \to A$ corresponding to the constant function $\tilde{x}(*) = x$, then $f \circ \tilde{x}(*) = f(x)$. Let $x \neq y$. Since f is mono, $(f \circ \tilde{x})(*) \neq (f \circ \tilde{y})(*)$, and hence $f(x) \neq f(y)$, thus f is an injection. □

There is also an generalization of the notion of *surjections*.

Definition 1.6. An arrow $f : a \to b \in \mathcal{C}$ is a **epimorphism** (or simply epi), if for all objects x and all arrows $g, h : b \to x$ we have:

$$g \circ f = h \circ f \implies g = h.$$

Finally, we introduce the notion of an 'invertible arrow'.

Definition 1.7. An arrow $f : a \to b \in \mathcal{C}$ is an **isomorphism** if there exists an arrow $g : b \to a$ so that:

$$g \circ f = \mathrm{id}_a \text{ and } f \circ g = \mathrm{id}_b.$$

In set, **epi** and **mono** imply **iso**. This however does not hold for general categories!

Special functors

Lastly, we turn our attention to special kinds of functors. For this we first introduce the notion of a *hom-set* of a and b, the set[2] of all arrows from a to b:

$$\mathrm{Hom}_{\mathcal{C}}(a, b) = \{f \in \mathcal{C} \mid f : a \to b\}.$$

Definition 1.8. A functor $F : \mathcal{C} \to \mathcal{D}$ is **full** if for all pairs $a, b \in \mathcal{C}$ the induced function:

$$\begin{aligned} F : \mathrm{Hom}_{\mathcal{C}}(a, b) &\to \mathrm{Hom}_{\mathcal{D}}(Fa, Fb), \\ f &\mapsto Ff \end{aligned}$$

is a surjection. It is called **faithful** if it is an injection.

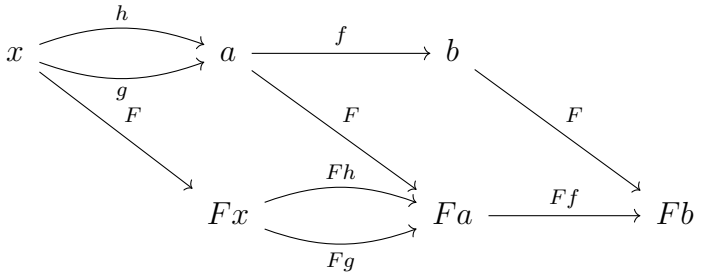
When after applying F an arrow Ff or an object Fa has a certain property(i.e. being initial, terminal or epi, mono), it is implied that f (or a) had this property, then we say the F ***reflects*** *the property*.

This allows for statements such as this:

Theorem 1.9. A faithful functor reflects epis and monos.

Proof. As an example we will prove it for a Ff that is mono. Let $f : a \to b$ such that Ff is mono, and let $h, g : x \to a$ such that $h \neq g$.

[2]Here we assume that this collection is a set, or that the category is so-called *locally small*



Since $g \neq h$ and F is faithful, we have $Fg \neq Fh$. This implies, because Ff is mono, that $Ff \circ Fg \neq Ff \circ Fh$, and since F is a functor we have $F(f \circ g) \neq F(f \circ h)$, implying $f \circ g \neq f \circ h$, and hence f is mono.

□

1.4 Natural transformations

Definition 1.10. A **natural transformation** μ between two functors $F, G : \mathcal{C} \to \mathcal{D}$ is a family of morphisms:

$$\mu = \{\mu_a : Fa \to Ga \mid a \in \mathcal{C}\},$$

indexed by objects in $\mathcal{C}$, so that for all morphisms $f : a \to b$ the diagram

$$\begin{array}{ccc} Fa & \xrightarrow{\mu_a} & Ga \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle Gf} \\ Fb & \xrightarrow{\mu_b} & Gb \end{array}$$

commutes. This diagram is called the *naturality square*. We write $\mu : F \Rightarrow G$, and call μ_a *the component of* μ *at* a.

We can *compose* natural transformations, turning the set of functors from $\mathcal{C} \to \mathcal{D}$ into a category. Let $\mu : F \Rightarrow G$ and $\nu : G \Rightarrow H$, then we have $\nu \circ \mu : F \Rightarrow H$ defined by (in components):

$$(\nu \circ \mu)_a = \nu_a \circ \mu_a.$$

Where the composition of the rhs is simply composition in $\mathcal{D}$.

1.5 Exercises

Exercise 1.1. Let $\mathcal{C}$ be a category, and let $f : a \to b$ in $\mathcal{C}$ be iso with inverse $g : b \to a$. Show that g is unique, i.e. for any g' that is an inverse of f we have $g' = g$.

Exercise 1.2. Let $F : \mathcal{C} \to \mathcal{D}$, and let $f : a \to b$ be an isomorphism in $\mathcal{C}$. Show that $Ff : Fa \to Fb$ is an isomorphism in $\mathcal{D}$.

Exercise 1.3. Is there a functor $Z : \mathbf{Grp} \to \mathbf{Grp}$ so that $Z(G)$ is the center of G?

Exercise 1.4. Let $F : \mathcal{C} \to \mathcal{D}, G : \mathcal{D} \to \mathcal{E}$ be functors, define $G \circ F : \mathcal{C} \to \mathcal{E}$ and show that it is a functor.

Exercise 1.5. Let $F, G : \mathcal{C} \to \mathcal{D}$ be functors, and let $\mu : F \Rightarrow G$. Show that μ is an isomorphism (in the category of functors between $\mathcal{C}$ and $\mathcal{D}$) if and only if its components are isomorphisms (in $\mathcal{D}$) for all $a \in \mathcal{C}$.

1.6 References

- 1.1 – 1.4 and 1.8 of Mac Lane
- 1.1, 1.2, 2.1, 3.1 and 3.2 of Asperti and Longo
- 2.1, 2.7, 2.8, 3.1, 4.2, 4.3 of Barr and Wells
- 1.1, 1.7, 1.10 of the 'Category Theory for Programmers' blog by Bartosz Milewski (best to study after reading Chapter 2)

Chapter 2

Types and functions: a category for programmers

> ""A monad is a monoid in the category of endofunctors, what's the problem?"
>
> *James Iry jokes about Haskell in his blog post A Brief, Incomplete, and Mostly Wrong History of Programming Languages*

To establish a link between functional programming and category theory, we need to find a category that is applicable. Observe that a *type* in a programming language, corresponds to a *set* in mathematics. Indeed, the type `int` in C based languages, corresponds to some finite set of numbers, the type `char` to a set of letters like `'a'`, `'z'` and `'$'`, and the type `bool` is a set of two elements (`true` and `false`). This category, the category of types, turns out to be a very fruitful way to look at programming.

Why do we want to look at types? Programming safety and correctness. In this part we will hopefully give an idea of how category theory applies to programming, but we will not go into to much detail yet, this is saved for later parts.

We will take as our model for the category of Haskell types (**Hask**) the category **Set**. Recall that the elements of **Set** are sets, and the arrows correspond to maps. There is a major issue to address here: Mathematical maps and functions in a computer program are not identical (bottom value $\perp$). We may come back to this, but for now we consider **Set** and **Hask** as the same category.

In Haskell, we can express that an object has a certain type:

```
a :: Integer
```

In C++ we would write:

```
int a;
```

To define a function $f : A \to B$ from type A to type B in Haskell:

```
f :: A -> B
```

To compose:

```
g :: B -> C
h = f . g
```

This means that `h` is a function `h :: A -> C`! Note how easy it is to compose functions in Haskell. Compare how this would be in C++, if we were to take two polymorphic functions in C++ and compose them:

```
template <typename F, typename G>
auto operator*(G g, F f) {
    return [&](auto x) { return g(f(x)); };
}


int main() {
    auto f = [](int x) -> float { return ...; };
    auto g = [](float y) -> int { return ...; };

    std::cout << (g * f)(5) << "\n";
}
```

We need some additional operations to truly turn it into a category. It is easy to define the identity arrow in Haskell (at once for all types):

```
id :: A -> A
id a = a
```

in fact, this is part of the core standard library of Haskell (the Prelude) that gets loaded by default. Ignoring reference types and e.g. `const` specifiers, we can write in C++:

```
template <typename T>
T id(T x) {
    return x;
}
```

There is one issue we have glared over; in mathematics all functions are *pure*: they will always give the same output for the same input. This is not always the case for computer programs, using IO functions, returning the current date, using a global variable are all examples of impure operations that are common in programming. In Haskell, *all functions are pure*, and this is a requirement that allows us to make the mapping to the category **Set**. The mechanism that allows Haskell programs to still do useful things is powered by *monads*, which we will discuss later.

Although many of the things we will consider can apply to other languages (such as Python and C++), there is a strong reason why people consider often consider Haskell as an example in the context of category theory and programming; it originates in academia and therefore takes care to model the language more accurately. For example, since we take as our model the category **Set**, there should be a type that corresponds to the empty set $\emptyset$. In C / C++, the obvious candidate would be `void` for this set, but consider a function definition:

```
void f() { ... };
```

This can be seen as a function from `void -> void`. We can call this function using `f()`, but what does it mean to call a function? We always invoke a function for an argument, so `void` actually corresponds to the set with a single element! Note that C functions that return void either do nothing useful (i.e. discard their arguments), or are impure. Indeed, even using a pointer argument to return a value indirectly modifies a 'global state'! In Haskell, the type corresponding to the *singleton set* (and its single value) is denoted with `()`. Meaning that if we have a function:

```
f :: () -> Int
```

we can invoke it as `f()`! Instead, the type `Void` corresponds to the empty set, and there can never be a value of this type. There is even a (unique) polymorphic (in the return type!) function that takes `Void` added to the prelude:

```
absurd :: Void -> a
```

You may be tempted to discard the type `Void` as something that is only used by academics to make the type system 'complete', but there are a number of legitimate uses for `Void`. An example is *Continuation passing style*, or CPS, where functions do not return a value, but pass control over to another function:

```
type Continuation a = a -> Void
```

In other words, a continuation is a function that *never returns*, which can be used to manipulate control flows (in a type-safe manner).

Recall that an initial object has exactly one arrow to each other object, and a terminal object has exactly one arrow coming from each other object. These objects are unique up to isomorphism. In the category of types, they correspond to `Void` and `()` respectively.

To summarize this introduction, in the category of 'computer programs', types are objects, and *pure* functions between these types are arrows. Next, we consider how we can apply some of the concepts we have seen, such as functors and natural transformations, to this category.

2.1 Containers as functors

When we consider functors in the category of types, the first question is 'to what category?'. Here, we will almost exclusively talk about functors from **Hask** to itself, i.e. *endofunctors*.

Endofunctors in **Hask** map types to types, and functions to functions. There are many examples of functors in programming. Let us first consider the concept of *lists of objects*, i.e. arrays or vectors. In C++ a list would be written as:

```
std::vector<T> xs;
```

or in Python we would have;

```
>>> import numpy as np
>>> a = np.array([1,2,3], dtype='int')
>>> type(a)
<class 'numpy.ndarray'>
>>> a.dtype
dtype('int64')
```

Note here that the true type of the numpy array is hidden inside the object, meaning its the responsiblity of the program to make sure that the types of operations match! The reason that we consider `numpy` arrays is that normal 'lists' in Python are actually *tuples*, which we will discuss when we talk about products and coproducts.

Let us consider the mathematical way of expressing this:

Example 2.1. Lists of some type are more generally called **words over some alphabet** (i.e. a set) X, and we denote the set of all finite words of elements[1] in X as X^*. Elements in X^* look like:

$$(x_1, x_2, x_3)$$
$$(x_1)$$
$$()$$

These are all examples of *words* in X (where the last example corresponds to the empty word). If we want to construct a *word functor* T, then T would then have the signature:

$$\begin{aligned} T &: X \to X^* \\ &: (f : X \to Y) \mapsto (Tf : X^* \to Y^*) \end{aligned}$$

For this second option, we have an obvious candidate for the precise function, let $f : X \to Y$ be some map, then Tf maps a word in X gets to a word in Y in the following way:

$$Tf(x_1, x_2, x_3, ...x_n) = (f(x_1), f(x_2), f(x_3), \ldots, f(x_n)).$$

Type classes and type constructors

We will express this idea in Haskell, but before we can do this we first have to consider type classes and -constructors. A *type constructor* is a 'function' (on types, not an arrow) that creates a type out of a type. A *type constructor* can have multiple *value constructors*, and these constructors can be differentiated between using something called *pattern matching* which we will see later. As an example, consider `Bool`.

```
data Bool = True | False
```

[1] Also called the *Kleene closure* of X

Here, we define the type constructor `Bool` as the resulting type corresponding to the *value* given by the value constructors `True` and `False`, which both are nullary constructors (that take no argument as types!). Normally however, type constructors take one or multiple types for their value constructors:

```
data Either a b = Left a | Right b
```

Here, the type constructor either hold either a value of type `a` or of type `b`, corresponding to the value constructors `Left` and `Right`. We will revisit this idea (and `Either`) when talk about products and coproducts.

A type class is a *common interface for types*. It defines a family of types that support the same operations. For example, a type class for objects that support equality is defined as:

```
class Eq a where
    (==) :: a -> a -> Bool
```

If we want to express the concept[2] *functor* using a typeclass, we have to state that it can send types to types, and that it sends functions between two types to functions with the appropriate signature, i.e.:

```
class Functor F where
    fmap :: (a -> b) -> F a -> F b
```

This says that `F` is a functor, if there is a function `fmap` that takes a function `f :: a -> b` and maps it to a function `fmap f :: F a -> F b`. Note that we do not explicitely have to state that `F` sends types to types, because this can be induced from the fact that we use `F a` where the compiler expects a type.

The List functor

The *list functor* in Haskell is denoted with `[]`, and a list of type `a` is denoted `[a]` (which is syntactic sugar, normally the type would be `[] a`).

Let us try to define this functor from the ground up. If we would write `List` instead of `[]`, then first we have to define what a list is. We can define this as follows:

[2] In C++, type constructors are referred to as *concepts*, and they have been a long time coming (but are not yet in the standard)

```haskell
data List a = Nil | Cons a (List a)
```

Here the type constructor has two possible ways of constructing (partitioning the possible values of the type): a list of `a`s is either empty (corresponding to the *constructor* `Nil`), or that it is the concatenation (corresponding to the *constructor* `Cons`) of an object of type `a` with another list of `a`s. Note that this is a recursive definition!

Next we define the `fmap` corresponding to our `List` functor (i.e. how it maps functions). The corresponding definition to the map described for the *word functor* is:

```haskell
instance Functor List where
    fmap _ Nil = Nil
    fmap f (Cons x t) = Cons (f x) (fmap f t)
```

If a list is empty, then we get the empty set, otherwise we map the indivual values in the list recursively using the given `f`. In `C++` this `fmap` functor roughly corresponds to `std::transform`, while for Python the closest thing would be the `map` function. With these two definitions, `List` is a functor! We could check the that it satisfies the requirements.

As mentioned, `List` is implemented in the standard library as `[]`, and `Cons` is written as `:`, while the empty list is written also as `[]`. This allows you to write:

```haskell
x = 1 : 2 : [] -- this results in `[1, 2] :: [Int]`!
```

The Maybe functor

As a simpler example, consider a type that either has no value or it has a value corresponding to some type `a`. In Haskell, this is called `Maybe`, while in C++ this is called `std::optional`, in Python the same idea could be achieved using:

```python
def fn(a):
    if (a >= 0)
        return sqrt(a)
    return None
```

This function returns `None` (corresponding to 'no value') if we provide 'invalid input'. This functor can be defined as:

```haskell
data Maybe = Nothing | Just a
```

And to turn it into a functor, we define `fmap`:

```haskell
instance Functor Maybe where
    fmap _ Nothing = Nothing
    fmap f (Just a) = Just (f a)
```

2.2 Polymorphic functions as natural transformations

Now that we view type constructors as functors, we can consider natural transformations between type constructors. If we let `a` be a type, then a natural transformation `alpha` would be something that maps between `F a` and `G a`, where `F` and `G` are type constructors:

```haskell
alpha :: F a -> G a
```

Note that implicitely we talk about the component of `alpha` at `a`, since this function is *polymorphic* the right component gets picked by the compiler. For example, say we have a list `[a]`, and we want to obtain the first element of this list. If the list is empty, then there is no such element, otherwise we obtain an `a`; i.e. the result is a `Maybe a`:

```haskell
head :: [a] -> Maybe a
head [] = Nothing
head (x:xs) = x
```

Here, we have a natural transformation between the `List` and the `Maybe` functor!

Parametric polymorphism and ad-hoc polymorphism

In C++, a template does not have to be defined for all types, i.e. we can write:

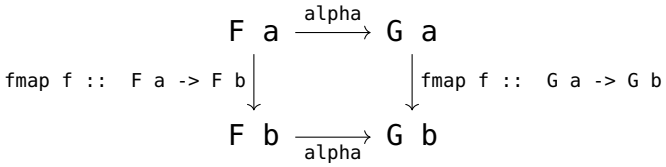
```cpp
template <typename T>
T f(T a);

template <>
int f(int a) { return 2 * a; }
```

```
template <>
double f(double a) { return 2.0 * a; }
```

Here, e.g. `f<int>(1)` would yield `2`, while `f<char>('a')` would result in a compilation error.

In Haskell, this is not allowed, polymorphic functions must work for *all types*, this is called parametric polymorphism. Specializing function definitions is done using type classes[3]. This has an important consequence (or perhaps, it is the underlying reason): a parametric polymorphic function satisfies automatically the naturality conditions.

The corresponding diagram is:



Here the left `fmap` works for `F`, while the right `fmap` corresponds to `G`, and the top `alpha` is implicitely the component at `a`, while the bottom one is the component at `b`. What would have to show, is that automatically

```
fmap f . alpha = alpha . fmap f
```

This can be shown in a very general context, and it has to do with the fact that the 'bodies'for `f`, `fmap` and `alpha` are the same for all types. We will show this in an upcoming part when we discuss *free theorems*.

Let us revisit our `head :: [a] -> Maybe a` example, and consider the naturality condition here. It says that:

```
fmap f . head = head . fmap f
```

Here, the fmap on the lhs corresonds to the `Maybe` functor, while on the rhs it corresponds to the `[]` functor. The lhs can b e read like this; take the first element of the list, then apply f on it. The rhs can be read as "apply the function f to the enitre list, then take the first element". The result is the same; the funtion f applied to the

[3] in C++ this would be done using overloading and (partial) template specialization

head of the list (if any). But for the rhs we apply the function `f` for each element in the list, while on the lhs we only apply it to the head. Because of the constraint on polymorphic function, the compiler knows that the result is equal and can choose which one to use!

2.3 References

- 1.2, 1.7 of the 'Category Theory for Programmers' blog by Bartosz Milewski

Chapter 3

Products, co-products and algebraic data types

3.1 Duality and products of objects

Duality

For any category, we can define the category with all arrows (and composition) reversed.

Definition 3.1. The *opposite category* $\mathcal{C}^{\text{op}}$ of a category $\mathcal{C}$ is the category with:

- The same objects as $\mathcal{C}$.
- For all arrows $f : a \to b$ in $\mathcal{C}$, there is an arrow $f^{\text{op}} : b \to a$
- The composition of $f^{\text{op}} : a \to b$ and $g^{\text{op}} : b \to c$ is given by:

$$g^{\text{op}} \circ f^{\text{op}} = (f \circ g)^{\text{op}}$$

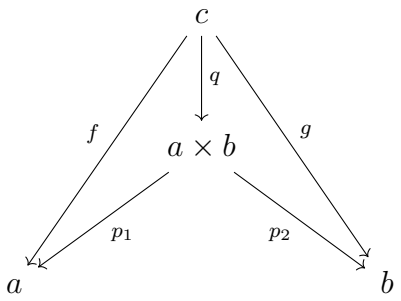
The opposite category is very useful, because many concepts defined in the original category have 'dual notions' in the opposite category. Clearly, for example, an *initial object in $\mathcal{C}$ is a terminal object in $\mathcal{C}^{op}$*. Similarly, an arrow that is *mono in $\mathcal{C}$ is epi in $\mathcal{C}^{op}$*. This is called **duality**, and provides so-called 'co-' notions of constructs, as well as 'co-' versions of theorems.

Whenever defining something it always make sense to see what this means in the opposite category, giving you a lot of free information. For example, we showed that faithful functors reflects mono's. Looking at the dual category, we immediately have that it also reflects epi's!

Products

Initial objects and terminal objects have a *universal property*, they are defined by the property that e.g. all other objects have a *unique morphism to the object*. A more involved example of such a universal property is the *notion of a product of objects*. The categorical product is a unifying definition for many 'products' encountered in mathematics, such as the cartesian product, product group, products of topological spaces, and so on.

Definition 3.2. Let $\mathcal{C}$ be a category, and let $a, b \in \mathcal{C}$ be objects in $\mathcal{C}$. A *product* of a and b is an object $a \times b \in \mathcal{C}$ along with two arrows $p_1 : a \times b \to a$ and $p_2 : a \times b \to b$ (the *projections*) so that for all objects $c \in \mathcal{C}$ and arrows $f : c \to a$ and $g : c \to b$ there exists a unique morphism $q : c \to a \times b$ that makes the following diagram commute:

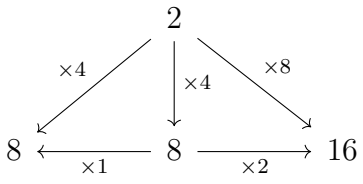


In this case, the (unique) arrows q are what gives the product a *universal mapping property*. If a product exists, it is unique op to unique isomorphism.

We say that the functions f and g *factors* through $a \times b$, or that $a \times b$ *factorizes* f and g. The reason for this name is clear when making the analogy with numbers. Consider:

$$f = p_1 \circ q, \; g = p_2 \circ q.$$

For an example with numbers:

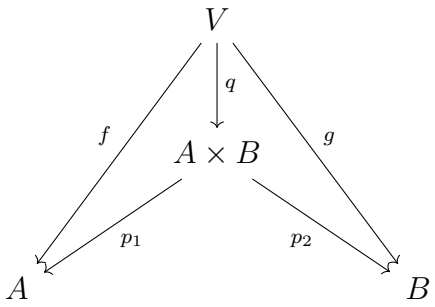


$$4 = 2 \times 2, \; 8 = 4 \times 2.$$

This seems to indicate that in 'some category related to numbers' (in fact, precisely the category of natural numbers with arrows to their multiples, that we gave as an example in the first chapter), the product would correspond to the gcd!

Example 3.3. Let us consider the product of objects in **Set**. Consider two sets A, B. We have a clear candidate for a product; the cartesian product $A \times B$. Given any element (or *pair*) $(a, b) \in A \times B$, the projections p_1, p_2 send it to a and b respectively. Is this indeed a product?

Let V be any other set, with arrows (functions) f to A and g to B. Can we construct a (unique) arrow q to $A \times B$?



Consider any element $v \in V$. It gets mapped to $f(v) \in A$, and $g(v) \in B$. Let $q : v \mapsto (f(v), g(v))$, then $(p_1 \circ f)(v) = f(v)$, and thus $p_1 \circ f = f$. Similarly $p_2 \circ g = g$.

Indeed, we have constructed an arrow that makes the above diagram commute. It is also clear that this is the *only arrow* that satisfies this, so we conclude that $A \times B$ is the product of A and B in the category **Set**. Another example of a product of sets would be $B \times A$, which is cannonically isomorphic to $A \times B$ (the isomorphism corresponds to 'swapping' the elements, which is its own inverse).

For a completely different example, we consider the category corresponding to a poset.

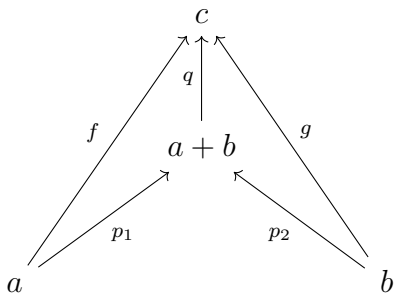
Example 3.4. Let us consider the product of objects in the category corresponding to some poset P. Consider two elements $x, y \in P$. A product $z \equiv x \times y$ would be equiped with two arrows $z \to x$ and $z \to y$, which means $z \leq x$ and $z \leq y$. Furthermore, for any element w with arrows to x, y (i.e. $w \leq x$ and $w \leq y$), there has to be an arrow $q : w \to z$ (i.e. $w \leq z$). This is the same as saying that, in addition to $z \leq x$ and $z \leq y$, we have for all elements w of the poset:

$$w \leq x \text{ and } w \leq y \implies w \leq z$$

This means that z is the "largest element that is smaller or equal to x and y", also called the *infimum* of x and y.

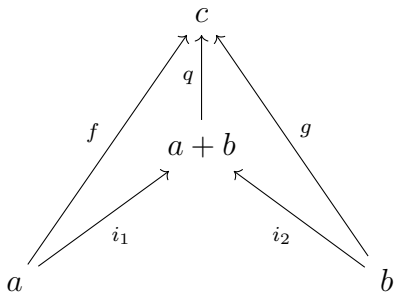
Coproducts

Let us revisit the idea of *duality*. What would be the dual-notion of the product? Let us take the product diagram, and reverse the arrows:



This already very suggestives, we have arrows going from objects a, b into the coproduct (written 'a+b', we will see why soon), and from this coproduct arrows going to arbitrary target objects c. The arrows $a \to a + b$ and $b \to a + b$ already look kind of like an inclusion. Let us see what happens when we apply duality to the product definition, and change some names.

Definition 3.5. Let $\mathcal{C}$ be a category, and let $a, b \in \mathcal{C}$ be objects in $\mathcal{C}$. A *coproduct* of a and b is an object $a + b \in \mathcal{C}$ along with two arrows $i_1 : a + b \leftarrow a$ and $i_2 : a + b \leftarrow b$ (the *inclusions*) so that for all objects $c \in \mathcal{C}$ and arrows $f : c \leftarrow a$ and $g : c \leftarrow b$ there exists a unique morphism $q : c \leftarrow a + b$ that makes the following diagram commute:



Note that this is precisely the definition of the product, with all arrows reversed and the projections renamed to i_1 and i_2.

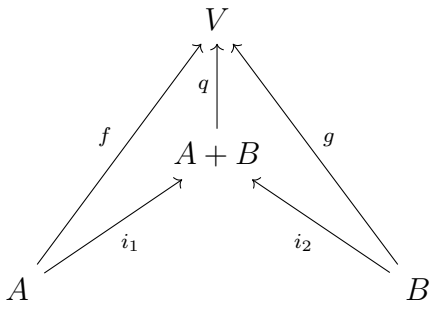
Because of the properties that we will show discover, the coproduct is also called the *sum*. Note that this dual notion is fundamentally different. Let us see what it means for the category **Set**:

Example 3.6. Consider two sets A, B. When looking at the diagram for the coproduct, we see that we need to find some kind of set in which elements of A and B are represented but completely independent; since c is now the target of the functions we want to factor through $a + b$.

This describes the *union* of A and B, but only if the two are disjoint since in the intersection of A and B we would not know whether q should represent f or g. This is easily solved by looking at the disjoint union, which has a nice representation:

$$A + B \equiv \{(a, 0) \mid a \in A\} \cup \{(b, 1) \mid b \in B\}.$$

It is clear what i_1 and i_2 are. Let V be any other set, with arrows (functions) $f : A \to V$ and $g : B \to V$.



Consider any element $a \in A$. It gets mapped to $f(a) \in V$, and to $i_1(a) = (a, 0)$ in $A + B$. Then we should set $q(a, 0) \equiv f(a)$, and similarly we should set $q(b, 1) \equiv g(b)$. This already defines q uniquely and completely, so we conclude that the disjunt union is indeed the coproduct in the category **Set**.

We note there that the coproduct (and product) of two objects, generalizes also to products of more than 2 objects (by simply adding more maps $i_1, i_2, i_3 \ldots$).

3.2 Algebraic data types

Let us apply the product (and coproduct) concepts to the category of types. Since we already saw what these constructs mean for sets, namely the cartesian product and the disjoint union respectively, it should be clear what this means for types.

Given a type a and a type b, the product corresponds to a *pair*, written `(a, b)` in Haskell. We could implement this ourselves using simply:

```
data Pair a b = Pair a b
```

Here, we give the unique value constructor the same name as its type constructor. In C this would correspond roughly to a `struct` (more specifically a POD data type), although a `Record` in Haskell corresponds more precisely to a `struct`. Note for this to make sense, the product type should (and is) be defined for more than 2 elements.

In C++ this is known as a `std::pair` (or a `std::tuple` for n-ary products). However, its implementation (and also usage) is awkward and convoluted. Functional programming (and product/coproduct types) is not yet a first-class citizen in C++.

The coproduct (or *sum type* corresponds to a value that has either type a, or type b. This is implemented as the `Either` data type:

```
data Either a b = Left a | Right b
```

Here, the two value constructors take an element of type `a`, or an element of type `b` respectively. In C and C++ this would correspond roughly to a union[1], except that it is *tagged*.

A sum type means choosing an alternative between types, while the product type is a combination of the types. Let us look at some more examples:

- In C, an `enum` represents a fixed number of alternative constants. In Haskell, this would correspond to the sum type of multiple 0-ary value constructors (implicitely the finite sum type of the type `()` with itself):

  ```
  data Enum = One | Two | Three
  ```

- A node of a binary tree of type `a` has a sum type: it is either `()` (representing a leaf), or it is the product type of:

  - Tree on the left
  - `a` for the value of the node
  - Tree on the right

[1] In C++17 there will be a standardized 'tagged union' 'std::variant' that more accurately models the coproduct

or in Haskell:

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

- Using the product and sum types, we can turn the type system into a semi-ring, where we define:
  - 0 = `Void`
  - 1 = `()`
  - $a + b$ = `Either a b = Left a | Right b`
  - $a \times b$ = `(a, b)`

  Let us check that 0 really works as 0. What happens when we add `Void` to a type:

  ```haskell
  Either a Void = Left a | Right Void
  ```

  We can never get a value for void, so the only thing we can do is to construct `Either a Void` with a value of type `a`, which means:

  $$a + 0 = a.$$

  Similarly, if we have a product with `Void`, we can never instantiate a pair (because there is no value for `Void`), so the corresponding product type is again `Void`:

  $$a \times 0 = 0.$$

  Although this is all a bit of a stretch, this analogy has some interesting properties, and we can do some real algebra with our types and try to interpret the results. Consider again the list type:

  ```haskell
  List a = Empty | Cons a (List a)
  ```

  In our 'semi-ring', writing x for `List a`, this would look like the expression:

  $$x = 1 + a \times x$$

  This is unsolvable, but we can try to iteratively substitute x into the right hand side:

  $$\begin{aligned} x &= 1 + a \times (1 + ax) \\ &= 1 + a + a^2 x \\ &= 1 + a + a^2(1 + ax) \\ &= 1 + a + a^2 + a^3(1 + ax) \\ &= \dots \end{aligned}$$

Which can be read as 'a list is either empty, or it has one element of type `a`, or it has two elements of type `a`, etc. Although this is mostly an entertaining (and, depending on your view, an overly complicated) way of looking at types, a similar correspondence from types to logical operations forms the basis of the Curry-Howard isomorphism that connects type theory to logic in a very fundamental way.

3.3 Bi-functors

Definition 3.7. Given two categories $\mathcal{C}, \mathcal{D}$ their product category $\mathcal{C} \times \mathcal{D}$ is given by:

- The objects are pairs (c, d) where $c \in \mathcal{C}$ and $d \in \mathcal{D}$.
- The arrows are pairs of arrows, $(f, g) : (c, d) \to (c', d')$ for $f : c \to c'$ in $\mathcal{C}$ and $g : d \to d'$ in $\mathcal{D}$.
- The identity arrow for (c, d) is the pair $(\mathrm{id}_c, \mathrm{id}_d)$.
- Composition of arrows happens per component, i.e. when f, g in $\mathcal{C}$ and $h, k \in \mathcal{D}$:
$$(f, h) \circ (g, k) \equiv (f \circ g, h \circ k)$$

Note that alternatively we could define this as the product of objects in the category **Cat**.

This brings us to the concept of a bifunctor, which can be seen as a 'functor of two arguments'.

Definition 3.8. Let $\mathcal{C}, \mathcal{D}, \mathcal{E}$ be categories. A bifunctor is a functor:
$$F : \mathcal{C} \times \mathcal{D} \to \mathcal{E}.$$

We now ask ourselves how bifunctors relate to functors. This is summarized in the following proposition, where we denote pairs as $\langle c, d \rangle \in \mathcal{C} \times \mathcal{D}$:

Proposition 3.9. Let $F : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$ be a bifunctor. Then:
$$F\langle c, - \rangle \equiv G_c : \mathcal{D} \to \mathcal{E},\ d \mapsto F\langle c, d \rangle,\ (g : d \to d') \mapsto F\langle \mathrm{id}_c, g \rangle$$
$$F\langle -, d \rangle \equiv H_d : \mathcal{C} \to \mathcal{E},\ c \mapsto F\langle c, d \rangle,\ (f : c \to c') \mapsto F\langle f, \mathrm{id}_d \rangle$$
are functors for all $c \in \mathcal{C}$ and $d \in \mathcal{D}$ respectively, and furthermore they satisfy:
$$G_c d = H_d c \tag{3.1}$$
$$G_{c'} g \circ H_d f = H_{d'} f \circ G_c g \tag{3.2}$$

for all $c, c' \in \mathcal{C}$ and $d, d' \in \mathcal{D}$. Conversely, let G_c, H_d be family of functors so that (3.1) and (3.2) hold, then:

$$\tilde{F} : \mathcal{C} \times \mathcal{D} \to \mathcal{E},\ \langle c, d\rangle \mapsto G_c d,\ \langle f, g\rangle \mapsto H_{d'} f \circ G_c g$$

is a bifunctor, and satisfies $\tilde{F}\langle c, -\rangle = G_c$ and $\tilde{F}\langle -, d\rangle = H_d$.

Proof. Let us first show that we can construct the functors G_c and H_d from a bifunctor F. We show that G_c is a functor, H_d follows similarly.

$$\begin{aligned} G_c(\mathrm{id}_d) &= F\langle \mathrm{id}_c, \mathrm{id}_d\rangle = \mathrm{id}_{F\langle c,d\rangle} \\ G_c(g \circ g') &= F\langle \mathrm{id}_c, g \circ g'\rangle = F(\langle \mathrm{id}_c, g\rangle \circ \langle \mathrm{id}_c, g'\rangle) \\ &= F\langle \mathrm{id}_c, g\rangle \circ F\langle \mathrm{id}_c, g'\rangle = G_c g \circ G_c g' \end{aligned}$$

and clearly the mapped arrows have the correct (co)domains, hence G_c is a functor for all c. Showing (3.1) is simply, by definition both sides are equal to $F\langle c, d\rangle$. To show (3.2) we compute:

$$\begin{aligned} G_{c'} g \circ H_d f &= F\langle \mathrm{id}_{c'}, g\rangle \circ F\langle f, \mathrm{id}_d\rangle \\ &= F(\langle \mathrm{id}_c, g\rangle \circ \langle f, \mathrm{id}_d\rangle) = F(\langle f, g\rangle) = F(\langle f, \mathrm{id}_{d'}\rangle \circ \langle \mathrm{id}_c, g\rangle) \\ &= F\langle f, \mathrm{id}_{d'}\rangle \circ F\langle \mathrm{id}_c, g\rangle = H_{d'} f \circ G_c g \end{aligned}$$

To show the converse statement, we compute:

$$\begin{aligned} F\langle \mathrm{id}_c, \mathrm{id}_d\rangle &= G_c \mathrm{id}_d \circ H_d \mathrm{id}_c = \mathrm{id}_{G_c d} \circ \mathrm{id}_{H_d c} = \mathrm{id}_{F\langle c,d\rangle} \circ \mathrm{id}_{F\langle c,d\rangle} = \mathrm{id}_{F\langle c,d\rangle} \\ F(\langle f, g\rangle \circ \langle f', g'\rangle) &= F\langle f \circ f', g \circ g'\rangle = G_{c'} g \circ G_{c'} g' \circ H_d f \circ H_d f' \\ &= G_{c'} g \circ H_{d'} f \circ G_c g' \circ H_d f' = F\langle f, g\rangle \circ F\langle f', g'\rangle \end{aligned}$$

□

In Haskell the bifunctor is implemented as a type class, which is implemented in the standard library as follows:

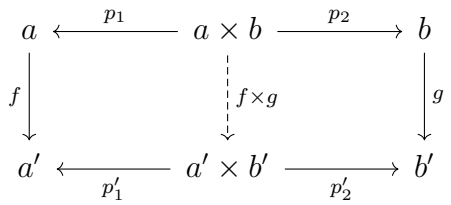
```
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
    bimap g h = first g . second h
    first :: (a -> c) -> f a b -> f c b
    first g = bimap g id
    second :: (b -> d) -> f a b -> f a d
    second = bimap id
```

Here you see a circular definition. This means it is enough to *either* provide the `bimap`, or the `first` and `second` functions, powered by Proposition 3.9.

Example 3.10. Whenever you have a category $\mathcal{C}$ where the product of two objects exists for all pairs of objects, then this gives rise to a bifunctor:

$$\begin{aligned}\times &: \mathcal{C} \times \mathcal{C} \to \mathcal{C} \\ &: (a, b) \mapsto a \times b \\ &: (f : a \to a', g : b \to b') \mapsto (f \times g : a \times b \to a' \times b')\end{aligned}$$

where we find $f \times g$ by looking at the diagram:



By definition of the product $a' \times b'$, we have that for any object c that has arrows to a' and b', there should be a *unique* arrow $c \to a' \times b'$. Note that $f \circ p_1$ and $g \circ p_2$ are arrows from $a \times b$ to a' and b' respectively, meaning that we can set $f \times g$ to the unique arrow going between $a \times b$ and $a' \times b'$.

By duality, there is also a bifunctor corresponding to the coproduct if it is defined everywhere. What would these two examples mean in Haskell? The product is the 'pair functor' `(,)`, and the coproduct is the sum type `Either`.

```haskell
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)

instance Bifunctor Either where
    bimap f _ (Left x)  = Left (f x)
    bimap _ g (Right y) = Right (g y)
```

These are examples of type constructors (or algebraic data types, as we have seen). Since functors compose, we could ask ourselves: "Are all algebraic data types functors?". The answer is positive, and this allows the Haskell language to derive an implementation of `fmap` for all ADTs!

3.4 Exercises

Exercise 3.1. In the category **Vect**, show that the product corresponds to the direct sum.

3.5 References

- 2.1, 2.2, 3.1, 3.3 (partially) and 3.4 (partially) of Mac Lane
- 1.5, 1.6 and 1.8 of the 'Category Theory for Programmers' blog by Bartosz Milewski
- 2.6.7, 5.1, 5.2, 5.4 of Barr and Wells
- *Catsters*: Products and coproducts https://www.youtube.com/watch?v=upCSDIO9pjc

Chapter 4

The Yoneda Lemma

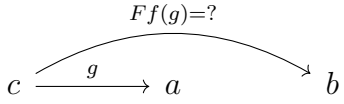
The Yoneda Lemma relates a category $\mathcal{C}$ with the functors from $\mathcal{C}$ to **Set**. Before we can introduce the lemma's we will introduce a number of concepts; first we introduce a class of functors called *hom-functors*, we introduce the notion of *representable functors*, we will discuss the *Yoneda embedding* and finally we will move on to the Yoneda Lemma; one of the important tools in category theory

4.1 Hom-functors

The *hom-functor* for some fixed object c, is a functor that sends any object a to the hom-set $\mathrm{Hom}(c, a)$. It is clear that for each object we get an associated object in **Set**, but what should this functor do with arrows? We will denote the candidate functor with $F = \mathrm{Hom}(c, -)$. Say we have an arrow $f : a \to b$:

$$\begin{array}{ccc} a & \xrightarrow{F} & \mathrm{Hom}(c, a) \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle ?} \\ b & \xrightarrow{F} & \mathrm{Hom}(c, b) \end{array}$$

The arrow marked with a question marked is an arrow in **Set**. Arrows in sets are functions, which we can define by saying what it does on elements. The elements of the hom-sets are arrows in $\mathcal{C}$. Given some element of $\mathrm{Hom}(c, a)$, i.e. an arrow in $\mathcal{C}$: $g : c \to a$, we need to obtain an element of $\mathrm{Hom}(c, b)$, i.e. an arrow from $c \to b$. We have the following picture



We can go to a from c using g, but then we need a way to get from a to b. We actually have a way to do this, namely the arrow $f : a \to b$ that we started with. We need only to compose! This motivates the following definition:

Definition 4.1. Let $\mathcal{C}$ be a category, and let $c \in \mathcal{C}$ and $f : a \to b \in \mathcal{C}$. We define the (covariant) **hom-functor** $\mathrm{Hom}(c, -) : \mathcal{C} \to \mathbf{Set}$ as:

$$\begin{aligned} \mathrm{Hom}(c,-)(a) =& \mathrm{Hom}(c,a) \\ \mathrm{Hom}(c,-)(f) :& \mathrm{Hom}(c,a) \to \mathrm{Hom}(c,b), \\ & g \mapsto f \circ g \end{aligned}$$

Clearly the identity arrow gets mapped to the identity map. To show that compositions are preserved, we compute for any arrow $h : c \to a$:

$$\begin{aligned} \mathrm{Hom}(c,-)(g \circ f)(h) &= (g \circ f) \circ h \\ &= g \circ (f \circ h) \\ &= g \circ (\mathrm{Hom}(c,-)(f)(h)) \\ &= \mathrm{Hom}(c,-)(g)\,(\mathrm{Hom}(c,-)(f)(h)) \\ &= (\mathrm{Hom}(c,-)(g) \circ \mathrm{Hom}(c,-)(f))\,(h) \end{aligned}$$

We can also define the contravariant hom-functor: $\mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ by *precomposing with* f, and we denote it as $\mathrm{Hom}(-, d)$.

Let us introduce a term; functors are called **naturally isomorphic** if there is a natural transformation between them for which all components are isomorphisms. Hom-functors are such an important class of functors from $\mathcal{C} \to \mathbf{Set}$, that they motivate the following definition:

Definition 4.2. A functor $F : \mathcal{C} \to \mathbf{Set}$ is called **representable** if it is naturally isomorphic to a hom-functor.

To simplify the notation in the upcoming sections, we will denote the covariant hom-functor $\mathrm{Hom}(a, -) = h^a$ and the contravariant hom-functor $\mathrm{Hom}(-, b) = h_b$.

4.2 Yoneda Embedding

For any category $\mathcal{C}$ the Yoneda embedding is a functor between the opposite category and the category of functors between $\mathcal{C}$ and **Set**. Let us first introduce this target category.

Definition 4.3. Let $\mathcal{C}$ and $\mathcal{D}$ be two categories, then we define $\mathbf{Fun}(\mathcal{C}, \mathcal{D})$ as the category with as objects functors $\mathcal{C} \to \mathcal{D}$, and as arrows natural transformations between these functors.

Now, we are ready to describe the Yonedda embedding. Note that because it is a functor between *the opposite of* $\mathcal{C}$ and the category of *functors* between $\mathcal{C}$ and **Set**, it should take objects to functors, and arrows to natural transformations. For all objects, we have introduced a functor associated to it in the previous section; the *hom-functor*.

$$\begin{array}{ccc} a & \xrightarrow{Y} & h^a \\ {\scriptstyle f}\downarrow & & \Downarrow{\scriptstyle Yf} \\ b & \xrightarrow{Y} & h^b \end{array}$$

The natural transformation Yf should have components which are arrows in **Set**, indexed by objects in $\mathcal{C}$. Let $k : d \to c$ (note the reversed order), the corresponding naturality square looks like this:

$$\begin{array}{ccc} \mathrm{Hom}(a,c) & \xrightarrow{(Yf)_c} & \mathrm{Hom}(b,c) \\ {\scriptstyle h^a(k)}\downarrow & & \downarrow{\scriptstyle h^b(k)} \\ \mathrm{Hom}(a,d) & \xrightarrow{(Yf)_d} & \mathrm{Hom}(b,d) \end{array}$$

So the natural components should be maps between hom-sets, and again we can find these maps by composition! This is summarized in the following definition:

Definition 4.4 (Yoneda embedding)**.** The **Yoneda functor** $Y : \mathcal{C}^{\mathrm{op}} \to \mathbf{Fun}(\mathcal{C}, \mathbf{Set})$, is defined as follows. Let $a \in \mathcal{C}$ and $f : b \to c$ in $\mathcal{C}$.

$$\begin{aligned} Ya =& h^a \\ Yf^{\mathrm{op}} :& h^c \to h^b \\ (Yf^{\mathrm{op}})_a :& \mathrm{Hom}(c,a) \to \mathrm{Hom}(b,a) \\ :& (g : c \to a) \mapsto (g \circ f : b \to a) \\ =& h_a f \end{aligned}$$

Note that the component is defined using *pre-composition*, it is a contravariant hom-functor, whereas the objects Ya are *covariant* hom-functors, i.e. use *post-composition*.

Let us check that Yf is indeed a natural transformation by looking at the naturality square introduced above, let $\ell : a \to c$, and lets trace it through the diagram for some $k : c \to d$ and $g : b \to a$:

$$\begin{array}{ccc} \ell \in \mathrm{Hom}(a,c) & \xrightarrow{(Yg^{\mathrm{op}})_c} & \mathrm{Hom}(b,c) \ni \ell \circ g \\ \downarrow{\scriptstyle h^a(k)} & & \downarrow{\scriptstyle h^b(k)} \\ k \circ \ell \in \mathrm{Hom}(a,d) & \xrightarrow{(Yg^{\mathrm{op}})_d} & \mathrm{Hom}(b,d) \ni k \circ (\ell \circ g) = (k \circ \ell) \circ g \end{array}$$

In other words, the naturality condition corresponds simply to the associativity in $\mathcal{C}$. We say that Yf is the *induced natural transformation* of f.

The reason that the Yoneda functor is of such interest is because of the following:

Theorem 4.5. The Yoneda functor Y is *full* and *faithful*.

We will prove this in the next section, after we state and prove the Yoneda lemma. Theorem 4.5 has the following corollary:

Corollary 4.6. Let $\mu : h^a \to h^b$ be a natural transformation between hom-functors, then it is given by composition with a unique arrow $f : b \to a$. Furthermore, μ is a (natural) isomorphism if and only if f is an isomorphism.

This means in particular that if a set-valued functor F is represented by both a and b, then there is an isomorphism $a \xrightarrow{\sim} b$.

Again, by duality, there exists also a full and faithful functor from $\mathcal{C} \to \mathbf{Fun}(\mathcal{C}^{\mathrm{op}}, \mathbf{Set})$.

4.3 The Yoneda Lemma

Corollary 4.6 tells us that any natural transformation between covariant hom-functors h^a and h^b is given by composition with an arrow in the reverse direction $f : b \to a$. Note that this arrow is an element of $h^b a = \mathrm{Hom}(b, a)$.

Less obviously, this result holds also for natural transformations between h^a and any other set-valued functor F.

What would a function between h^a and F look like? We see that a component of the natural transformation should take an element from $h^a b$, i.e. an arrow $g : a \to b$,

to some element of Fb. We can do this by *evaluating* the lifted arrow Fg , which is a map between the sets Fa and Fb, at a fixed $x \in Fa$.

This gives us an idea for a natural transformation corresponding to an element of Fa. We summarize this in the following proposition:

Proposition 4.7. Let $F : \mathcal{C} \to \mathbf{Set}$ be a functor, and $a \in \mathcal{C}$. Any element $x \in Fa$ induces a natural transformation from h^a to F, by evaluating any lifted arrow in x.

Proof. We have to show that this induces a natural transformation, i.e that the following diagram commutes:

$$\begin{array}{ccc} h^a b & \xrightarrow{F\_(x)} & Fb \\ {\scriptstyle h^a f}\downarrow & & \downarrow{\scriptstyle Ff} \\ h^a c & \xrightarrow[F\_(x)]{} & Fc \end{array}$$

Here we denote:

$$F\_(x) : h^a b \to Fb,\ f \mapsto Ff(x).$$

To show that the diagram commutes, fix an arrow $g : a \to b \in h^a b$. If we start taking it along the top side we obtain:

$$Ff(Fg(x)) = (Ff \circ Fg)(x) = F(f \circ g)(x) = (F\_(x))(f \circ g) = (F\_(x))((h^a f)(g))$$

which is equal to taking it along the bottom, hence the diagram commutes. □

The Yoneda lemma states that *all natural transformations between h^a and F are of this form*.

Theorem 4.8 (The Yoneda lemma)**.** Let $\mathcal{C}$ be a category, let $a \in \mathcal{C}$, and let $F : \mathcal{C} \to \mathbf{Set}$ be a set-valued functor. There is a one-to-one correspondence between elements of Fa, and natural transformations:

$$\mu : h^a \Rightarrow F.$$

Proof. We already saw that each element of Fa induces a natural transformation, so we have a map:

$$\Phi : Fa \to \mathrm{Nat}(h^a, F).$$

Here, $\mathrm{Nat}(h^a, F)$ denotes the set of natural transformations between h^A and F. We now need to show show that Φ has an inverse. Let μ be any natural transformation,

then we can obtain an element of Fa by looking at the component μ_a and let it act on the identity arrow $\mathrm{id}_c \in h^a a$, i.e.:

$$\Psi : \mu \mapsto \mu_a(\mathrm{id}_a).$$

Now let us show that Φ and Ψ are inverses of each other. First, we compute:

$$\Psi(\Phi(x)) = \Psi(F\_(x)) = F\mathrm{id}_a(x) = \mathrm{id}_{Fa}(x) = x,$$

so Ψ is a left inverse of Φ. To show that it is also a right inverse, we need to show that:

$$\Phi(\Psi(\mu)) = \mu,$$

or in components:

$$\Phi(\Psi(\mu))_b = \mu_b.$$

We note that by definition, for any $f : a \to b$ in $h^a b$:

$$\Phi(\Psi(\mu))_b(f) = (\Phi(\mu_a(\mathrm{id}_a)))_b(f) = Ff(\mu_a(\mathrm{id}_a)).$$

Since μ is a natural transformation we have that the following diagram commutes:

$$\begin{array}{ccc} h^a a & \xrightarrow{\mu_a} & Fa \\ {\scriptstyle h^a f}\downarrow & & \downarrow{\scriptstyle Ff} \\ h^a b & \xrightarrow[\mu_b]{} & Fb \end{array}$$

In particular, consider the element $\mathrm{id}_a \in h^a a$. Tracing this along bottom this gets mapped to $\mu_b(f)$, while along the top it gives precisely $Ff(\mu_a(\mathrm{id}_a))$, so we have shown that:

$$\Phi(\Psi(\mu))_b(f) = Ff(\mu_a(\mathrm{id}_a)) = \mu_b(f).$$

And hence, Ψ is also a right inverse of Φ, and thus Φ is a bijection, as required.

□

One can also show, that this correspondence is 'natural' in $a \in \mathcal{C}$ and F.

Let us now prove Theorem 4.5.

proof of Theorem 4.5. By Yoneda's Lemma there is a bijection between the sets:

$$\mathrm{Nat}(h^b, h^a) \simeq h^a b = \mathrm{Hom}(a, b)$$

for all objects a and b of $\mathcal{C}$, which directly implies that the functor Y is full and faithful. □

Let us recap what we have seen so far. We discussed a special class of set-valued functors called *hom-functors*. These hom-functors, like hom-sets, relate objects directly with the arrows between them.

Next we showed that we can *embed* any category into the category of contravariant set-valued functors of this category, sending objects to their hom-functors. We also showed that this embedding, as a functor, is *full* and *faithful*, which suggests that all the information of the category and its objects, is contained in its hom-functors.

When looking at what this means for the arrows, we noted that in particular any natural transformation between hom-functors is given by composition with arrows of our category.

To prove this, we stated and proved the Yoneda lemma – which is an important result in its own right. It shows that for an arbitrary set-valued functor, there is a bijection between elements of the set Fa and natural transformations from h^a to F,

All functors in Haskell are set-valued, since that is our category of interest. We first show two simple applications of Yoneda's lemma in mathematics, and next we see some initial applications of the Yoneda lemma to Haskell. In later parts we will see more advanced uses.

4.4 Examples of applications

Example 4.9 (Matrix row operations)**.** In linear algebra, row operations can be performed without changing the solutions of the linear system. Examples are row permutations, adding the j-th row to the i-th row, or multiplying a row by a (non-zero) scalar. We will show that these *row operations* are natural, in the following sense.

Let $\mathcal{C}$ be the category where the objects are natural numbers $1, 2, 3, \ldots$, and where arrows $n \to m$ correspond to $m \times n$ matrices. Composition is given by matrix multiplication, indeed if we have arrows:

$$n \xrightarrow{A_{m\times n}} m \xrightarrow{B_{k\times m}} k$$

then the composite $B_{k\times m}A_{m\times n} = C_{k\times n}$ is an arrow from n to k, as required. Consider contravariant hom-functors h_n for this category. The hom-set $h_n k = \mathrm{Hom}(k, n)$ consists of $n \times k$ matrices. To show that row operations can be seen as natural transformations $\mu : h_n \Rightarrow h_n$, we fix some $k \times m$ matrix B, and look at the following naturality square:

$$\begin{array}{ccc} h_n k & \xrightarrow{\mu_k} & h_n k \\ {\scriptstyle h_n B}\downarrow & & \downarrow{\scriptstyle h_n B} \\ h_n m & \xrightarrow{\mu_m} & h_n m \end{array}$$

Considering some $n \times k$ matrix A, the naturality condition states:

$$\mu(A)B \stackrel{?}{=} \mu(AB).$$

To show this, we observe that for all row transformations we have:

$$\mu(A) = A + \tilde{A}$$

where the rows of $\tilde{A}$ are either empty, or are multiples of rows of A, or:

$$\mu(A) = A + \Lambda A.$$

Where Λ is a matrix whose elements Λ_{ij} represent how many times row j should be added to row i. This means we have

$$\mu(A)B = (A + \Lambda A)B = AB + \Lambda AB = \mu(AB).$$

as required. By Corollary 4.6 we have that any natural transformation $\mu : h_n \Rightarrow h_n$ is given by postcomposition (in this category: left-multiplication) with a unique arrow $D : n \to n$. The Yoneda lemma allows us to identify this arrow; it is equal to:

$$D = \mu_n(\mathrm{Id}_n),$$

so to perform row operations on a matrix, one can equivalently left multiply with a matrix obtained by applying these operations to the identity matrix. This powers the technique for manually inverting a matrix A, where you perform row operations to the matrix A and simultaneously to another matrix B that is initially the identity matrix, until you reduce A to the identity matrix. The resulting matrix B, when left multiplied with the original A will perform the row operations, and hence $BA = \mathrm{Id}$, or $B = A^{-1}$.

Example 4.10. Another application of Yoneda is the following classic result from group theory:

Corollary 4.11 (Cayley's Theorem)**.** Any group G is isomorphic to a subgroup of a permutation group.

Proof. Recall that we can view a group G as a category $\mathcal{C}_G$ with a single object $\{\bullet\}$ and with arrows $\bullet \to \bullet$ corresponding to the elements of g. Consider the Yoneda embedding Y of this category into $\mathbf{Fun}(\mathcal{C}_G^{\mathrm{op}}, \mathbf{Set})$, and in particular we consider the shape of the image of $\bullet$ under the contravariant hom-functor $h_\bullet$:

$$G \dashleftarrow \bullet \xrightarrow{Y} h_\bullet \dashrightarrow \mathrm{Nat}(h_\bullet, h_\bullet)$$

The arrows on the left (displayed collectively using a dashed arrow), corresponding to the elements of G, get mapped *fully and faithfully* (by Theorem 4.5) to the natural transformations between $h_\bullet$ and itself (natural endomorphisms).

The natural endomorphisms $h_\bullet$ are characterized, by Corollary 4.6, (at the only component G) by left-multiplication of elements G on the set $h_\bullet \bullet \simeq G_{\mathrm{set}}$ which is the underlying set of G (since it is $\mathrm{Hom}(\bullet, \bullet)$). For each element $g \in G$ we obtain an automorphism $G_{\mathrm{set}} \to G_{\mathrm{set}}$ given by $h \mapsto gh$.

Recall that $\mathrm{Aut}(G_{\mathrm{set}})$ is a group (a permutation group), and note that the collection of automorphisms defined by left multiplication of elements of G is indeed a subgroup of this permutation group. The correspondence between G and the "automorphisms by left-multiplication" is easily seen to be a group isomorphism. □

4.5 Yoneda in Haskell

We will discuss a hopefully intuitive way of looking at the Yoneda lemma in Haskell, by pinpointing a function with a single evaluation. In later parts we will discuss many more applications of Yoneda to Haskell, in particular when we discuss *generalized ADTs* and *lenses*.

Let us first see how we can translate the relevant tools of Yoneda to Haskell. We have the following concepts:

- *hom-sets*: the hom-set of types `a` and `b` are the arrows between `a` and `b`, i.e. functions of the type `(a -> b)`. Note that this hom-set is again in the category of types.
- The *hom-functor* corresponding to a type `a` should be a functor, i.e. a type constructor, that produces the hom-set `(a -> b)` when given a type `b`, for some fixed type `a`. On functions `(b -> c)` it should get a function between the hom-sets of `a` and `b, c` respectively, i.e.:

```haskell
instance Functor (HomFunctor a) where
    fmap :: (b -> c) -> (a -> b) -> (a -> c)
    fmap f g = f . g
```

And indeed, we see that we can simply use composition.

- Yoneda's lemma says that for any other functor `F`, we can produce a natural transformation (i.e. polymorphic function in a type `b`) from the hom-functor for a fixed `a` by looking at elements of `F a`.

Next we look at a simple example of how to apply this final point in Haskell.

4.5.1 Reverse engineering machines

We set `F` equal to `Id`, the identity functor, and consider a natural transformation between `HomFunctor a` and `Id`, this has the form (at the component `b`):

```
--    (HomFunctor a) b      Id b
--             |              |
machine :: (a -> b)     ->    b
```

Say we are given any function with this signature, and we want to know how it is implemented. We can actually do this in a *single evaluation*, using the Yoneda lemma. The Yoneda lemma says precisely that such a *machine* is given uniquely by any element of `Id a = a`, i.e. some value of the type `a`. This makes a lot of sense in this context, since we can be given *any* `b`, and the only tool that we have to produce a value for `b` is to use the function `f :: a -> b` that is supplied to us. Furthermore, the polymorphic function should behave the same for any type, so it can only be implemented as:

```
machine :: (a -> b) -> b
machine f = f x
```

where `x` is some fixed element of type `a`. Now, the Yoneda lemma also tells us a way to obtain `x`, we simply supply `f = id`:

```
x <- machine id -- obtain the 'hidden element'
```

What if `F` is not the identity function, but say the `List` functor. The story actually does not change much, we now have a function with the signature:

```
--    (HomFunctor a) b      List b
--             |              |
machine :: (a -> b)     ->   [b]
```

the Yoneda lemma says that internally, any function of this signature should maintain a list of the type `[a]`, and when given a function `f :: a -> b` it fmaps this over the internal list to produce a value of the type `[b]`. Again, we can get this list by feeding the `id` function into the machine.

4.5.2 Continuation Passing Style

In programming, there is an equivalence between what is called *direct* style, where functions return values, and *continuation passing style* (CPS), where each *called function* takes an additional argument which is a *handler function* that does something with the result of the called function.

Say we have some function

```
T add(T a, T b) {
    return a + b;
}
```

Which we can use by calling e.g. `auto x = add(1, 2)`. The CPS version of this function looks like

```
void add_cps(T a, T b, F cont) {
    cont(a + b);
}
```

and the way it is used is:

```
add_cps(1, 2, [](auto result) {
    // ...
});
```

In other words, the CPS version of the function does not *return a value*, but rather passes the result to a handler. We do not bind the result of a function to a value, but rather to the *argument of a handler function*.

You may recognize this style of programming from writing concurrent programs, where continuations can be used to deal with values produced in the future by other threads without blocking. Continuations are also often used in UI frameworks, where a handler is used whenever e.g. a button is pressed, or the value of a slider has changed.

This CPS passing style can also be used to implement exceptions. Say we have a function that can throw:

```
void can_throw(F raise, G cont) {
    // ...
}
```

Here, the idea is that `raise` gets called if an error occurs, while `cont` gets called when a result has been computed succesfully. What is also interesting is that CPS can be used to implement *control flow*. For example, the called function can call cont multiple times (loops), or only conditionally.

Let us show that the *continuation passing transform* (CPT), i.e. going from direct style to CPS, is nothing more then the Yoneda embedding. Say we have a function:

```
f :: a -> b
```

Let us remind ourselves that the Yoneda embedding takes such an arrow, and produces a map $(Yf)_c = \text{Hom}(c, b) \to \text{Hom}(c, a)$ for all $c \in \mathcal{C}$. In Haskell, this embedding could be implemented like this:

```
yoneda :: forall x. (a -> b) -> (b -> x) -> (a -> x)
yoneda f = \k -> k . f
```

Going the other way around is easy, we simply pass `id` as our continuation `k`.

We will revisit continuations when we discuss monads.

- https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style
- https://golem.ph.utexas.edu/category/2008/01/the_continuation_passing_trans.html
- https://github.com/manzyuk/blog/blob/master/yoneda-embedding-is-cps.org

4.6 References

- 2.3, 2.4 and 2.5 of the 'Category Theory for Programmers' blog by Bartosz Milewski

- 2.2, 3.2 of Mac Lane.
- 3.1, 4.5 of Barr and Wells
- 2.2 of Riehl
- *Catsters*: Yoneda and representables : https://www.youtube.com/playlist?list=PLUWfjhrIRed_PgAmTFFyuEtFRdJzgcOZE
- Blogs:
  - http://www.haskellforall.com/2012/06/gadts.html
  - http://blog.sigfpe.com/2006/11/yoneda-lemma.html
  - https://www.schoolofhaskell.com/user/bartosz/understanding-yoneda#yoneda-lemma

Chapter 5

Cartesian closed categories and λ-calculus

In Haskell, functions that take two arguments can be written as:

```
-- 1) idiomatic haskell
f :: a -> b -> c
-- 2) more conventional style
f :: (a, b) -> c
```

the first style can be read as "for any fixed value `x` of type `a`, you are given a function `b -> c` which sends `y` to `f(x, y)`". In this part we will discuss why we are allowed to do this, and see the theory that underpins this. The process of converting the second to the first style is called *currying* (the reverse is called *uncurrying*) and can be described in the context of category theory.

In the language of category theory, we are trying to show the equivalence between arrows of the form $a \times b \to c$ and arrows of the form $a \to [b \to c]$, where $[b \to c]$ is some 'function object'. We will first state what it means to *curry* a function between sets.

Definition 5.1. Let A, B, C be sets. We define $[A \to B]$ to be the *set of functions* between A and B. Given a function of two variables:

$$f : A \times B \to C,$$

we have a function:

$$\lambda f : A \to [B \to C],$$

defined by $\lambda f(a)(b) = f(a, b)$. We say that λf is the *curried* version of f, and going from f to λf is called *currying*.

Going the other way around is called *uncurrying*. Let

$$g : A \to [B \to C],$$

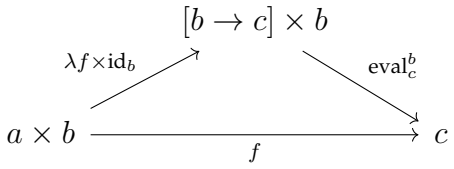
be a function, then we can define $\lambda^{-1}g : A \times B \to C$ by setting $\lambda^{-1}g(a, b) = g(a)(b)$. In other words, we have an isomorphism λ between the hom-sets:

$$\mathrm{Hom}_{\mathbf{Set}}(A \times B, C) \simeq \mathrm{Hom}_{\mathbf{Set}}(A, [B \to C]).$$

We are now ready to discuss this process more generally, but for this we need to specify what properties our category should have in order for this to work.

Definition 5.2 (Cartesian closed category)**.** A category $\mathcal{C}$ is called *cartesian closed* (or a CCC), if the following conditions are satisfied:

1. It has a terminal object 1.
2. For each pair $a, b \in \mathcal{C}$ there exists a product $a \times b$.
3. For each pair $a, b \in \mathcal{C}$ there exists an object $[a \to b]$ called the *exponential* such that:
   - there exists an arrow: $\mathrm{eval}_b^a : [a \to b] \times a \to b$.
   - For any arrow $f : a \times b \to c$ there is a unique arrow $\lambda f : a \to [b \to c]$ so that the following diagram commutes:



Here, the product of arrows $f \times g$ is as given in Example 3.10.

Wherever possible, we will denote eval_b^a simply as eval. Another common notation for the exponential $[a \to b]$ is b^a.

Note that the commutative diagram that shows up in the definition directly implies that we indeed have a bijection of hom-sets (i.e. it makes sense to curry). That is to say, let $\mathcal{C}$ be a CCC:

$$\mathrm{Hom}_{\mathcal{C}}(a \times b, c) \simeq \mathrm{Hom}_{\mathcal{C}}(a, [b \to c])$$

are isomorphic, by sending a $f : a \times b \to c$ using:

$$\lambda : f \mapsto \lambda f,$$

and vice versa:

$$\lambda^{-1} g = \text{eval}_b^c \circ (g \times \text{id}_b).$$

which is an isomorphism by the commutativity of the diagram and the uniqueness of λf.

To prove that curried and uncurried version of binary functions are actually *equivalent* we would have to show something stronger, that there is an arrow between $[a \times b \to c] \to [a \to [b \to c]]$ that is *iso*, but for this we some more complicated machinery which for now would be too big of a diversion.

One can show also show that exponentials are unique up to unique isomorphism, but this also requires some machinery that we have not yet developed. We may revisit this when when we get to discuss adjunctions.

We have already seen that **Set** is a CCC. Before we give some additional properties of CCCs and the exponential objects in them, let us look at some additional examples of CCCs:

Example 5.3 (Boolean algebras as CCCs)**.**

Definition 5.4. A **Boolean algebra** is a partially ordered set B such that:

- For all $x, y \in B$, there exists an infimum $x \wedge y$ and a supremum $x \vee y$.
- For all x, y, z we have a distributive property:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z).$$

- There exists a *smallest* element 0, and a *greatest* element 1, which satisfiy e.g. $0 \vee x = x$, $1 \vee x = 1$.
- There exists a complement $x \vee \neg x = 1$, $x \wedge \neg x = 0$.

Let us check that a Boolean algebra is a CCC.

- It has a terminal object 1.
- It has infimums (i.e. products) for all pairs of elements.

- We define the exponential as Boolean implication, i.e. $[a \to b] = \neg a \vee b$. Since $\text{eval}_b^a : [a \to b] \times a \to b$ and arrows between objects of posets are unique, we simply have to show that $[a \to b] \wedge a \leq b$ to obtain evaluation arrows:

$$\begin{aligned}[a \to b] \wedge a &= (\neg a \vee b) \wedge a = (\neg a \wedge a) \vee (b \wedge a) \\ &= 0 \vee (b \wedge a) = b \wedge a \leq b\end{aligned}$$

  Where we used a distributive property, and in the final step the definition of an infimum.

- Next we need to be able to curry, i.e. show that λf exists. Note that indeed we only have to show that such an arrow exists, by definition every diagram in a poset category commutes, since arrows between objects are unique. Say we have an arrow from $a \times b \to c$, i.e. we have $a \wedge b \leq c$. Then:

$$\begin{aligned}a &= a \wedge 1 = a \wedge (b \vee \neg b) = (a \wedge b) \vee (a \wedge \neg b) \leq c \vee (a \wedge \neg b) \\ &\leq c \vee \neg b \equiv \neg b \vee c \equiv [b \to c]\end{aligned}$$

  So there is indeed an arrow from $a \to [b \to c]$, as required.

Example 5.5 (Small categories as a CCC)**.** Before, we briefly discussed $\mathbf{Cat}$, the category of small categories. Let $\mathcal{C}, \mathcal{D} \in \mathbf{Cat}$, then we can define:

$$[\mathcal{C} \to \mathcal{D}] \equiv \mathbf{Fun}(\mathcal{C}, \mathcal{D}).$$

So the exponentials correspond to the *functor categories* between the categories in question.

Let $F : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$ be a functor, then we want to construct a functor $\lambda F : \mathcal{C} \to [\mathcal{D} \to \mathcal{E}]$. This functor should sends each object c to a functor $\lambda F(c)$ between $\mathcal{D}$ and $\mathcal{E}$, and arrows of $\mathcal{C}$ to natural transformations between $\mathcal{D}$ and $\mathcal{E}$. We define this, using F, as:

- The functor for $c \in \mathcal{C}$:
  - For objects $d \in \mathcal{D}$ we have $\lambda F(c)(d) = F(c, d)$.
  - For arrows $g : d \to d'$ we have $\lambda F(c)(g) = F(\text{id}_c, g)$.
- The natural transformations for $f : c \to c'$ in $\mathcal{C}$ should be between the functors $F(c), F(c')$:
  - For $d \in \mathcal{D}$, the component of $\lambda F f \equiv \mu$ at d is given by:

$$\begin{aligned}\mu &: F(c) \Rightarrow F(c') \\ \mu_d &: F(c)(d) \to F(c')(d) \\ &: F(c, d) \to F(c', d) \\ \mu_d &\equiv F(f, \text{id}_d)\end{aligned}$$

Let us check that this indeed defines a natural transformation. Let $g : d \to d'$ in $\mathcal{D}$:

$$\begin{array}{ccc} F(c,d) & \xrightarrow{\mu_d} & F(c',d) \\ {\scriptstyle F(c)(g)}\downarrow & & \downarrow{\scriptstyle F(c')(g)} \\ F(c,d') & \xrightarrow{\mu_{d'}} & F(c',d') \end{array}$$

To show that this commutes, we compute:

$$\begin{aligned} \rightarrow\downarrow &= F(c')(g) \circ \mu_d = F(\mathrm{id}_{c'}, g) \circ F(f, \mathrm{id}_d) \\ &= F(\mathrm{id}_{c'} \circ f, g \circ \mathrm{id}_d) \\ &= F(f \circ \mathrm{id}_{c'}, \mathrm{id}_{d'} \circ g) \\ &= F(f, \mathrm{id}_{d'}) \circ F(\mathrm{id}_c, g) = \mu_{d'} \circ F(c)(g) = \downarrow\rightarrow \end{aligned}$$

Where we used the definition of composition in a product category, and the fact that F is a functor so it plays nice with composition.

So we can indeed 'curry' in the category of small categories, the other properties (e.g. that it has a terminal object, the category with one object and its identity arrow) are easy to check.

5.1 λ-calculus and categories

One of the interesting features of a CCC is that it can model a λ-calculus, which is one of the universal models of computations, and corresponds to underlying computational model for functional programming (whereas imperative languages are based on Turing machines).

In this section we will give a brief and incomplete introduction to (typed) λ-calculus. Our reason to discuss them is to better understand functional languages, and to give further motivation to the definition of CCCs.

Expressions, or λ*-terms*, form the key component of λ-calculus. In these expressions there can be *variables* which are identifiers that can be seens as placeholders, and *applications*. An expression is defined recursively as one of the following:

- a *variable* x.
- if t is an expression and x a variable, $\lambda x.t$ is an expression (called an *abstraction*).
- if t and s are expressions, then so is ts (called an *application*).

The only 'keywords' that are used in the λ-calculus language are the λ and the dot. Multiple applications can be dismbiguated using parentheses, where the convention is that they associate from the left if these are omitted, i.e.

$$t_1 t_2 t_3 \dots t_n = (\dots((t_1 t_2)t_3)\dots t_n).$$

Abstractions can model functions, for example the identity function could be written as:

$$\lambda x.x$$

Note that the choice for the name x is completely arbitrary, equivalently we could have written

$$\lambda y.y \equiv \lambda z.z$$

and so on. This is called α**-conversion**.

Note that we do not have to give this function any name, but is simply defined to be the given expression. This is why anonymous functions in programming are often called λ-functions. On the left of the dot we have the *arguments* preceded by a λ. On the right of the dot we have the *body* expression.

Functions like this can be *applied* to expressions, by **substituting** the expressions as the 'value' for the argument, i.e. say we have a function evaluated at some point:

$$f(x) = ax,\ f(y)$$

then the corresponding expression would be:

$$(\lambda x.ax)y \equiv ay$$

This substitution process is called β**-reduction**, and can be seen as a computational step.

A *variable* can be free or bound, for example in our example

$$\lambda x.ax,$$

a is free, while x is bound – i.e. associated to an argument. We can make this formal:

Definition 5.6 (Free and bound variables)**.** A variable x is **free** only in the following cases:

- x is free in x.
- x is free in $\lambda y.t$ if $y \neq x$ are not the same identifier, and x is free in t.
- x is free in st if it is free in either s or t.

A variable x is **bound** in the following cases:

- x is bound in $\lambda y.t$ if $y = x$ is the same identifier, or if x is bound in t.
- x is bound in st if it is bound in either s or t.

Note that a variable can be both bound *and* free in the same expression. For example, y is both bound and free in:

$$(\lambda y.y)(\lambda x.xy).$$

Also, note that this implies that the same identifiers may be used indepently in multiple expressions, but should not be mixed up. We should rename identifiers wherever necessary when applying functions.

Say f does not contain x as a free variable, then we can equivalently write:

$$\lambda x.fx \equiv f,$$

this is called η-conversion.

Example 5.7 (Natural numbers)**.** Since the λ-calculus forms a very minimal programming language, we may expect it to be able to perform basic mathematical tasks. Indeed it can, and as an example we will see how we can model the natural numbers as expressions in λ-calculus.

We define:

$$0 \equiv \lambda s.(\lambda z.z) \equiv \lambda sz.z,$$

where we also introduced syntax for functions of multiple parameters. Note that by convention these associate from the right, contrary to expressions.

The natural numbers are defined recursively by applying s to the body of the function corresponding previous number:

$$\begin{aligned} 1 &= \lambda sz.sz \\ 2 &= \lambda sz.s(sz) \\ 3 &= \lambda sz.s(s(sz)) \\ &\dots \end{aligned}$$

This leads naturally to the *succesor function*, which correspond to the following expression:

$$S = \lambda wyx.y(wyx).$$

Writing $s^k z = s(s(s(s \dots (sz)))))$, with k occurences of s, we can see that:

$$\begin{aligned} Sk &= (\lambda wyx.y(wyx))(\lambda sz.s^k z) \\ &= (\lambda yx.y((\lambda sz.s^k z)yx)) \\ &= (\lambda yx.y(y^k x)) \\ &= (\lambda yx.y^{k+1} x) \\ &\equiv (\lambda sz.s^{k+1} z) \\ &\equiv k+1 \end{aligned}$$

In similar ways, one can define addition and multiplication, logical operations, equality, and ultimately even simulate a Turing machine using λ-calculus.

5.2 Typed λ-calculus

In the context of typed functional languages, we are interested in *typed* λ-calculus. This is an extension of λ-calculus, where each expression has a *type*. We will sketch this extension and the associated category here, but note that we will freely glance over some technicalities and will not prove anything in too much detail. The goal of this part is to give an idea of how CCCs can be applied more broadly to functional program than just formalizing the notion of function types and currying.

To define what a type is, we introduce the set of *symbols* as:

$$\mathcal{S} = \{S_1, S_2, S_3, \dots\}.$$

A *type* is defined recursively as either:

- A *symbol* S_i
- If T_1, T_2 are types, then so is $T_1 \to T_2$.

If t is an expression then we write $t : T$ to indicate that its type is T. Types corresponding to expressions have to obey a number of rules, e.g.:

- $c : T$ denotes a constant of type T.
- For each type, there is a countable number of variables $x_1 : T$, $x_2 : T$, ...
- If $t : T_1 \to T_2$ and $s : T_1$ then $ts : T_2$
- For a variable $x : T_1$, given an expression $t : T_2$, we obtain a *function* $\lambda x.t : T_1 \to T_2$.
- There is a singleton type 1 with an expression $* : 1$. Any other expression of this type is equal (see below) to $*$ as seen from $\Gamma = \emptyset$.

Equations, or *equality judgements* in this calculus have the form:

$$\Gamma | t = s : T.$$

Here, Γ is some set of variables that at least contains *all* the free variables in both t and s. Such an equation means that according to Γ (i.e. with respect to its variables), the expressions t and s of type T are equal. These equations are subjects to some rules, e.g. for fixed Γ they define an equivalence relation of expressions of type T, but we will not list these here. For an overview, see the suggested literature at the end of this chapter.

5.3 Typed λ-calculus as a CCC

We can go back and forth between CCCs and λ-calculus. Let us describe how we can obtain a CCC from a typed λ-calculus.

Definition 5.8. Given a typed λ-calculus $\mathcal{L}$, we associate it to a category $\mathcal{C}(\mathcal{L})$ where:

- The objects are types T.
- The arrows $T \to T'$ are pairs of
  - an equivalence class of expressions of types T'. The equivalence of two expressions t, s where t may contain the variable x, and s may contain the variable y, is defined as follows:
    - both s, t are of the same type T
    - x has the same type as y and is substitutable for it in s (this means that occurence of x becomes bound after substituting it for y in s)
    - $\{x\} | (\lambda y.s)x = t : T$.

    There are multiple reasons for needing this relation, e.g. we want all the expressions of a type T that correspond to single variables to correspond to the same identity arrow of the type T. Also, together with the properties of the singleton type 1, this ensures that we get a terminal object corresponding to the type 1.
  - a free variable x of type T (that does not necessarily have to occur in the expression(s))

You can prove $\mathcal{C}(\mathcal{L})$ is indeed cartesian closed, and also that any CCC defines a λ-calculus, but we will not do this here primarily because the definition given here is incomplete and would be overly long otherwise.

There are a number of advantages of viewing a λ-calculus from the viewpoint of a CCC. For example, often variables (identifiers) clash between expressions, and this requires carefully renaming variables where necessary. When considering the arrows of the associated CCC, all placeholders have been identified by means of the equivalence relation, and this is no longer an issue. Also, the fact that we can compose arrows means that results from category theory can be used for further reduction of expressions.

5.4 References

- 1.9 of the 'Category Theory for Programmers' blog by Bartosz Milewski
- 6.1, 6.2, 6.3 of Barr and Wells
- https://en.wikibooks.org/wiki/Haskell/The_Curry-Howard_isomorphism
- Raul Rojas: A tutorial introduction to λ-calculus
- Chapter 7 of van Oosten

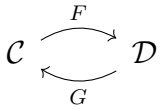
Chapter 6

Adjunctions

"Adjoint functors arise everywhere"

Saunders Mac Lane

There are multiple ways to introduce adjunctions, both in terms of the intuition behind them, as well as the actual definition. The setup is that there are two functors F, G:



that we want to relate. In particular, we want to generalize the *inverse* of a functor. We say that the functor F is an isomorphism with inverse G if:

$$\mathrm{Id}_{\mathcal{C}} = GF,\ FG = \mathrm{Id}_{\mathcal{D}}$$

where $\mathrm{Id}_{\mathcal{C}}$ is the identity functor on $\mathcal{C}$, and GF denotes $G \circ F$. A weaker notion is *isomorphism up to natural isomorphism*, where we require that there exists some natural isomorphisms

$$\mathrm{Id}_{\mathcal{C}} \tilde{\Rightarrow} GF,\ FG \tilde{\Rightarrow} \mathrm{Id}_{\mathcal{D}}$$

Even weaker is that we only require that there exists natural transformations:

$$\mathrm{Id}_{\mathcal{C}} \Rightarrow GF,\ FG \Rightarrow \mathrm{Id}_{\mathcal{D}}$$

This is what we are going to explore in this part.

6.1 Universal arrow adjunctions

Definition 6.1 (Universal arrow adjunction)**.** Let $\mathcal{C}, \mathcal{D}$ be categories. Let $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ be functors. If there exists a natural transformation:

$$\eta : \mathrm{Id}_{\mathcal{C}} \Rightarrow GF,$$

such that for all objects $c \in \mathcal{C}$ and $d \in \mathcal{D}$, and all arrows $f : c \to Gd$ there exists a unique arrow $g : Fc \to d$ such that the following diagram commutes:

$$\begin{array}{ccc} c & \xrightarrow{\eta_c} & GFc \\ & \searrow^{f} & \downarrow{\scriptstyle Gg} \\ & & Gd \end{array}$$

We call the triple (F, G, η) an *adjunction*, and η the *unit* of the adjunction. We say that F is left adjoint to G, and G is right adjoint to F, or simply $F \dashv G$.

In other words, given an adjunction and any arrow $f : c \to Gd$, i.e. from an arbitrary object of $\mathcal{C}$ to something in the image of G (so *relevant to the functor* G), we can equivalently consider an arrow $g : Fc \to d$ in $\mathcal{D}$ relating to the functor F, because we use the natural transformation η and our functors to convert them to the same arrow.

This means that the *relevant structure* of $\mathcal{C}$ with respect to the functor G, can also be found in $\mathcal{D}$ with respect to the functor F.

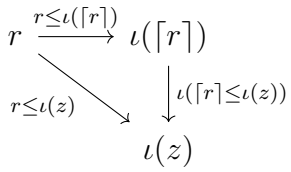
Example 6.2. View $\mathbb{Z}$ and $\mathbb{R}$ as categories, with $a \to b \iff a \leq b$. Let $I : \mathbb{Z} \to \mathbb{R}$ be the inclusion functor that sends $z \to \iota(z)$. I is left adjoint to the functor $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$ that sends $r \to \lfloor r \rfloor$. Indeed, consider the following diagram in $\mathbb{Z}$:

$$\begin{array}{ccc} z & \xrightarrow{z \leq z} & \lfloor \iota(z) \rfloor = z \\ & \searrow^{z \leq \lfloor r \rfloor} & \downarrow{\scriptstyle G(\iota(z) \leq r)} \\ & & \lfloor r \rfloor \end{array}$$

the existence of a unique $g = \iota(z) \leq r$ for such an f corresponds to the statement:

$$\iota(z) \leq r \iff z \leq \lfloor r \rfloor.$$

For the converse, consider the ceiling functor $\lceil \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ and the following diagram in $\mathbb{R}$:



Which corresponds to the statement:

$$r \leq \iota(z) \iff \lceil r \rceil \leq z,$$

showing that the inclusion functor is right adjoint to the ceil functor. So we have the adjunction chain:

$$\lceil \cdot \rceil \dashv I \dashv \lfloor \cdot \rfloor.$$

Example 6.3. An important class of adjunctions take the form **free** $\dashv$ **forgetful**. Let X be a set. The free monoid $F(X)$ is defined as:

$$F(X) = (X^*, +\!\!+ \,, ()),$$

see Example 2.1 for the definition of X^*, $+\!\!+$ denotes the concatenation of words as a binary operator, and $()$ denotes the empty word. F defines a *free functor*:

$$F : \mathbf{Set} \to \mathbf{Mon},$$

sending a set to the free monoid over that set. There is also a *forgetful functor*:

$$U : \mathbf{Mon} \to \mathbf{Set},$$

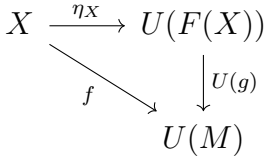
sending a monoid to its underlying set, that sends monoid homomorphisms to the corresponding function on sets. We define:

$$\eta : \mathrm{Id}_{\mathbf{Set}} \Rightarrow U \circ F,$$

as having components defining a function that sends an element $x \in X$ to a singleton word containing that element:

$$\eta_X(x) = (x).$$

To show that (F, U, η) form an adjunction, we consider some $f : X \to U(M)$ where M is a monoid, and we want to show that there is a unique monoid homomorphism $g : F(X) \to M$ that makes the following diagram commute:



We have to define:

$$\begin{aligned} g(()) &= \mathrm{id}_M \\ g((x)) &= f(x) \\ g((x_1, x_2, \ldots, x_n)) &= f(x_1)f(x_2)\ldots f(x_n) \end{aligned}$$

to make g into a monoid homomorphism that satisfies also:

$$f(x) = U(g)(\eta_X x) = U(g)((x)).$$

Before moving on, we first show that there are other definitions of adjunctions, which we will show are equivalent to the one we gave above, but are useful for describing other examples of adjunctions.

6.2 Equivalent formulations

There is an alternative way of describing adjunctions, as a natural bijection between hom-sets.

Definition 6.4 (Hom-set adjunctions)**.** Let $\mathcal{C}, \mathcal{D}$ be categories. Let $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ be functors. If there is a natural bijection:

$$\mathrm{Hom}_{\mathcal{D}}(Fc, d) \xrightarrow{\phi_{c,d}} \mathrm{Hom}_{\mathcal{C}}(c, Gd),$$

for each $c \in \mathcal{C}$ and $d \in \mathcal{D}$, then $(F, G, \{\phi_{c,d}\}_{c\in\mathcal{C}, d\in\mathcal{D}})$ is an *adjunction*. Here, the bijection should be natural in both c and d, where in $\mathcal{D}$ we have that for all $g : d \to d'$ in $\mathcal{D}$ the following diagram commutes:

$$\begin{array}{ccc} \mathrm{Hom}_{\mathcal{D}}(Fc, d) & \xrightarrow{\phi_{c,d}} & \mathrm{Hom}_{\mathcal{C}}(c, Gd) \\ {\scriptstyle g\circ\_}\downarrow & & \downarrow{\scriptstyle Gg\circ\_} \\ \mathrm{Hom}_{\mathcal{C}}(Fc, d') & \xrightarrow{\phi_{c,d'}} & \mathrm{Hom}_{\mathcal{C}}(c, Gd') \end{array}$$

while naturality in $\mathcal{C}$ means that for all $f : c' \to c$ the following diagram commutes:

$$\begin{array}{ccc} \mathrm{Hom}_{\mathcal{D}}(Fc, d) & \xrightarrow{\phi_{c,d}} & \mathrm{Hom}_{\mathcal{C}}(c, Gd) \\ {\scriptstyle \_\circ Ff}\downarrow & & \downarrow{\scriptstyle \_\circ f} \\ \mathrm{Hom}_{\mathcal{C}}(Fc', d) & \xrightarrow{\phi_{c',d}} & \mathrm{Hom}_{\mathcal{C}}(c', Gd) \end{array}$$

We can show that given a universal arrow adjunction, we can obtain a hom-set adjunction.

Proposition 6.5. Let (F, G, η) be a univeral arrow adjunction. Then the family of functions:

$$\begin{aligned}\phi_{c,d} : \ & \mathrm{Hom}_{\mathcal{D}}(Fc, d) \to \mathrm{Hom}_{\mathcal{C}}(c, Gd), \\ & (\alpha : Fc \to d) \mapsto G\alpha \circ \eta_c\end{aligned}$$

defines a hom-set adjunction $(F, G, \{\phi_{c,d}\}_{c\in\mathcal{C}, d\in\mathcal{D}})$.

Proof. First we show that $\phi_{c,d}$ is a bijection. Because (F, G, η) is an adjunction, we know that:

$$\forall f : c \to Gd, \exists!\ g : Fc \to d, \text{ s.t. } f = Gg \circ \eta_c.$$

Injectivity of $\phi_{c,d}$ is guaranteed by the uniqueness of the arrow g, while surjectivity is guaranteed by the existence of such an arrow.

Next we have to show that it is natural in both $\mathcal{C}$, and $\mathcal{D}$ which means respectively that for all $f : c' \to c$ and $g : d \to d'$:

$$G\alpha \circ \eta_c \circ f = G(\alpha \circ Ff) \circ \eta_{c'} \tag{6.1}$$

$$Gg \circ G\alpha \circ \eta_c = G(g \circ \alpha) \circ \eta_c \tag{6.2}$$

Equation 6.1 follows from the functoriality of G and the naturality of η:

$$G(\alpha \circ Ff) \circ \eta_{c'} = G(\alpha) \circ G(Ff) \circ \eta_{c'} = G(\alpha) \circ \eta_c \circ f.$$

Equation 6.2 follows directly from the functoriality of G. □

Definition 6.6 (Unit-counit adjunctions)**.** Let $\mathcal{C}, \mathcal{D}$ be categories. Let $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ be functors. If there are natural transformations:

$$\eta : \mathrm{Id}_{\mathcal{C}} \Rightarrow GF, \ \epsilon : FG \Rightarrow \mathrm{Id}_{\mathcal{D}},$$

such that the following diagrams (the *triangle identities*) commute:

$$\begin{array}{ccc} F & \xRightarrow{F\eta} & FGF \\ & \searrow^{\mathrm{id}_F} & \Big\Downarrow \epsilon F \\ & & F \end{array}$$

$$\begin{array}{ccc} G & \xRightarrow{\eta G} & GFG \\ & \searrow^{\mathrm{id}_G} & \Big\Downarrow G\epsilon \\ & & G \end{array}$$

where we use the notation (now in components) $(\eta G)_d = \eta_{Gd}$ and $(F\eta)_c = F(\eta_c)$, then (F, G, η, ϵ) is an *adjunction*. We call η the *unit* and ϵ the *counit* of the adjunction.

Note that this means that the unit is the *translated inverse* of the counit and vice versa.

Proposition 6.7. We can construct a unit-counit adjunction (F, G, η, ϵ) from a hom-set adjunction.

Proof. We define η and ϵ as having components:

$$\eta_c : c \to GFc = \phi_{c,Fc}(\mathrm{id}_{Fc}) \tag{6.3}$$
$$\epsilon_d : FGd \to d = \phi^{-1}_{Gd,d}(\mathrm{id}_{Gd}) \tag{6.4}$$

Let us prove that η is a natural transformation, the proof of the naturality of ϵ is dual to this. We want to show that the following diagram commutes for all $f : c \to c'$:

$$\begin{array}{ccc} c & \xrightarrow{\eta_c} & GFc \\ \downarrow{\scriptstyle f} & & \downarrow{\scriptstyle GFf} \\ c' & \xrightarrow{\eta_{c'}} & GFc' \end{array}$$

i.e. that:

$$\rightarrow_{\downarrow} = GFf \circ \eta_c = \eta_{c'} \circ f = {}_{\downarrow}\!\rightarrow$$

Plugging in our definition for η_c, and using the naturality of $\phi_{c,d}$ we see:

$$\begin{aligned} GFf \circ \phi_{c,Fc}(\mathrm{id}_{Fc}) &= \phi_{c,Fc'}(Ff \circ \mathrm{id}_{Fc}) \\ &= \phi_{c,Fc'}(\mathrm{id}_{Fc'} \circ Ff) \\ &= \phi_{c',Fc'}(\mathrm{id}_{Fc'}) \circ f = \eta_{c'} \circ f \end{aligned}$$

To show the first triangle identity, i.e. that for all $c \in \mathcal{C}$:

$$\epsilon_{Fc} \circ F(\eta_c) = \mathrm{id}_{Fc},$$

we use naturality of $\phi^{-1}_{GFc,Fc}$:

$$\begin{aligned} \phi^{-1}_{GFc,Fc}(\mathrm{id}_{GFc}) \circ F(\phi_{c,Fc}(\mathrm{id}_{Fc})) &= \phi^{-1}_{c,Fc}(\mathrm{id}_{GFc} \circ \phi_{c,Fc}(\mathrm{id}_{Fc})) \\ &= \phi^{-1}_{c,Fc}(\phi_{c,Fc}(\mathrm{id}_{Fc})) = \mathrm{id}_{Fc} \end{aligned}$$

For the second triangle identity, i.e. for all $d \in \mathcal{D}$:

$$G(\epsilon_d) \circ \eta_{Gd} = \mathrm{id}_{Gd},$$

we use the naturality of $\phi_{Gd,FGd}$:

$$\begin{aligned} G(\phi^{-1}_{Gd,d}(\mathrm{id}_{G}d)) \circ \phi_{Gd,FGd}(\mathrm{id}_{FGd}) &= \phi_{Gd,d}(\phi^{-1}_{Gb,b}(\mathrm{id}_{Gb}) \circ \mathrm{id}_{FGb}) \\ &= \phi_{Gd,d}(\phi^{-1}_{Gb,b}(\mathrm{id}_{Gb})) = \mathrm{id}_{Gb} \end{aligned}$$

□

To complete the cycle of equalities, we show that we can retrieve our original universal arrow adjunction from the unit-counit adjunction.

Proposition 6.8. Let (F, G, η, ϵ) be a unit-counit adjunction. Then (F, G, η) forms a universal arrow adjunction.

Proof. Let $f : c \to Gd$. We need to show that there is a unique solution to the equation $G(?) \circ \eta_c = f$.

From the second triangle identity, naturality of η, and functorality of G, we have:

$$\begin{aligned} G(\epsilon_d) \circ \eta_{Gd} &= \mathrm{id}_{Gd} \\ G(\epsilon_d) \circ \eta_{Gd} \circ f &= f \\ G(\epsilon_d) \circ GFf \circ \eta_c &= f \\ G(\epsilon_d \circ Ff) \circ \eta_c &= f \end{aligned}$$

So that the required $g \equiv \epsilon_d \circ Ff : Fc \to d$ exists. To show that it is unique, let:

$$\begin{aligned} f &= G(g) \circ \eta_c \\ Ff &= FG(g) \circ F\eta_c \\ \epsilon_d \circ Ff &= \epsilon_d \circ FG(g) \circ F\eta_c \\ \epsilon_d \circ Ff &= g \circ \epsilon_{Fd} \circ F\eta_c \\ \epsilon_d \circ Ff &= g \circ \mathrm{id}_{Fc} \\ \epsilon_d \circ Ff &= g \end{aligned}$$

So g must be of this form, as required. □

Summarizing what we saw so far, adjunctions can be defined either as:

1. *Universal arrow adjunction*: As a triple (F, G, η) together with a universal mapping property.
2. *Hom-set adjunction*: As a natural bijection between hom-sets
3. *Unit-counit adjunction*: As (F, G, η, ϵ) satisfying the triangle identities.

And we showed $1 \implies 2 \implies 3 \implies 1$, meaning that all these definitions are equivalent.

6.3 Uniqueness of adjoints

You can show that adjoints are unique up to natural isomorphism. Say $F, F' : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$. Assume $F \dashv G$ and $F' \dashv G$, with natural bijections $\phi_{c,d}$ and $\phi'_{c,d}$ respectively. Then we have for all $c \in \mathcal{C}$:

$$\mathrm{Hom}_{\mathcal{D}}(Fc, -) \simeq \mathrm{Hom}_{\mathcal{C}}(c, G-) \simeq \mathrm{Hom}_{\mathcal{D}}(F'c, -),$$

through natural isomorphisms in **Set** defined by $\phi_{c,-}$ and $\phi'_{c,-}$ respectively, by composing them we obtain:

$$\mathrm{Hom}_{\mathcal{D}}(Fc, -) \simeq \mathrm{Hom}_{\mathcal{D}}(F'c, -),$$

but the Yoneda embedding then says that Fc and $F'c$ are isomorphic (see Corollary 4.6). To show that these isomorphisms $Fc \to Fc'$ define the components of a natural isomorphism $F \Rightarrow F'$ we have to show that the following diagram commutes:

$$\begin{array}{ccc} Fc & \xrightarrow{\simeq} & F'c \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle F'f} \\ Fc' & \xrightarrow[\simeq]{} & F'c' \end{array}$$

Because the Hom-functor $\mathrm{Hom}_{\mathcal{C}}(-, d)$ is faithful, the above diagram commutes if[1]:

$$\begin{array}{ccc} \mathrm{Hom}(d, Fc) & \xrightarrow{\simeq} & \mathrm{Hom}(d, F'c) \\ {\scriptstyle h^d(Ff)}\downarrow & & \downarrow{\scriptstyle h^d(F'f)} \\ \mathrm{Hom}(d, Fc') & \xrightarrow[\simeq]{} & \mathrm{Hom}(d, F'c') \end{array}$$

which commutes by the naturality of $\phi_{c,d}$ (in $\mathcal{D}$).

We conclude that adjoints are unique up to natural isomorphism.

6.4 Examples

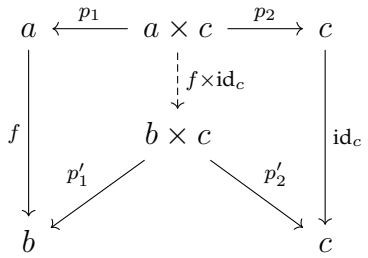
Example 6.9. The exponential object of a CCC is described by an adjunction.

[1] You can prove that faithful functors reflect commutative diagrams, by showing that it preserves non-commutative diagrams

Consider the functor:

$$\begin{aligned} - \times c : \ & \mathcal{C} \to \mathcal{C}, \\ & a \mapsto a \times c, \\ & f : a \to b \mapsto f \times \mathrm{id}_c. \end{aligned}$$

Here, $f \times \mathrm{id}_c$ is the unique arrow from $a \times c \to b \times c$ that makes the following diagram commute:

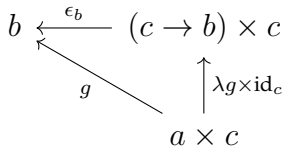


If $- \times c$ has a right adjoint, which we will suggestively denote:

$$(- \times c) \dashv (c \to -),$$

then for this adjunction, the universal property in Exercise 6.1 states:

For any $g : a \times c \to b$ there exists a unique arrow $f \equiv \lambda g : a \to (c \to b)$ such that the following diagram commutes:



so that the universal property for the counit is identical to the universal property of the evaluation function, compare also with Definition 5.2 of a CCC. Since adjoints are essentially unique, the exponential is determined by the adjunction.

You can show that adjunctions preserve (among other constructions involving universal properties) initial objects, terminal objects and products, which can be used to prove many useful and familiar equalities in a CCC. For example, we have $R_a(b \times c) \simeq R_a(b) \times R_a(c)$ which in the notation $a \to b \equiv b^a$ says:

$$(b \times c)^a \simeq b^a \times c^a.$$

Conversely, the product functor preserves coproducts, in that $(- \times c)(a + b) \simeq (- \times c)a + (- \times c)b$, or:

$$(a + b) \times c \simeq (a \times c) + (b \times c),$$

which shows that CCC's are distributative.

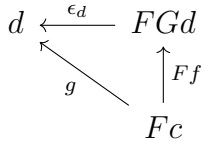
Other examples:

- Free/forgetful functor pairs.
- Groups G and their abelianizations $G^{ab} \equiv G/[G, G]$ form an adjunction.
- As an interesting application that we will see shortly, adjunctions also give rise to monads.

6.5 Exercises

Exercise 6.1. Argue using duality that the counit satisfies the following universal mapping property:

For any $g : Fc \to d$ there is a unique arrow $f : c \to Gd$ such that the following diagram commutes:



Exercise 6.2. Let $\Delta : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$ be the *diagonal functor* defined as:

$$\Delta a = (a, a)$$
$$\Delta(f : a \to b) = (f, f) : (a, a) \to (b, b)$$

Show that if the category $\mathcal{C}$ has binary products if and only if Δ has a right adjoint Π. Here, the functor $\Pi : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ should send $(a, b) \mapsto a \times b$.

Hint: write the components of the counit and the arrows that arise in the universal arrow property of the counit (see Exercise 6.1), in terms components of $\mathcal{C} \times \mathcal{C}$, i.e. $\epsilon_d = (p_1, p_2)$, $f = (q_1, q_2)$.

Use that a diagram in $\mathcal{C} \times \mathcal{C}$ commutes if and only if the diagrams for each component commute, and show that you obtain the definition for the binary product.

Exercise 6.3. Although we proved almost everything equationally in this part, some parts can be proved more efficiently using the Yoneda lemma, for example we consider the natural bijection in the definition of a hom-set adjunction as a natural transformation between the hom-functors:

$$\mathrm{Hom}(F-,-) \Rightarrow \mathrm{Hom}(-,G-)$$

from $\mathcal{C}^{\mathrm{op}} \times \mathcal{D} \to \mathbf{Set}$. Think about this.

6.6 References

- https://www.youtube.com/watch?v=K8f19pXB3ts
- Chapter 13 of Barr and Wells
- Chapter 4 of Riehl
- Chapter 4 of Mac Lane

Chapter 7

Monads

> "Mathematics is the art of giving the same name to different things"
>
> *Henri Poincaré*

Monads are used all throughout functional programming. In this part, we will try to understand them by first studying their mathematical definition and properties. Afterwards, we describe their use in functional programing by giving a number of motivating examples.

Any endofunctor $T : \mathcal{C} \to \mathcal{C}$ can be composed with itself, to obtain e.g. T^2 and T^3 (which are both again endofunctors from $\mathcal{C}$ to $\mathcal{C}$. A monad concerns an endofunctor, together with natural transformation between this functor and its composites that give it a "monoid-like structure".

7.1 Monads over a category

Say α is a natural transformation $T \Rightarrow T'$, where T, T' are endofunctors of $\mathcal{C}$, then note that α_x is a morphism from $Tx \to T'x$ in the category $\mathcal{C}$. Since this is a morphism, we can use T or T' to lift it, i.e. we obtain arrows at components $(T\alpha)_a \equiv T(\alpha_a)$ and $(\alpha T)_a \equiv \alpha_{Ta}$.

In particular, note that this defines natural transformations between the appropriate composite functors since the image of any commutative diagram under a functor is again commutative.

We are now ready to dive into the definition of a monad:

Definition 7.1. A **monad** $M = (T, \eta, \mu)$ over a category $\mathcal{C}$, consists of an endofunc-

tor $T : \mathcal{C} \to \mathcal{C}$ together with natural transformations:

$$\eta : \mathrm{Id} \Rightarrow T$$
$$\mu : T^2 \Rightarrow T$$

so that the following diagrams commute:

$$\begin{array}{ccc} T^3 & \xRightarrow{\mu T} & T^2 \\ {\scriptstyle T\mu}\Downarrow & & \Downarrow{\scriptstyle \mu} \\ T^2 & \xRightarrow{\mu} & T \end{array}$$

$$\begin{array}{ccccc} T & \xRightarrow{\eta T} & T^2 & \xLeftarrow{T\eta} & T \\ & \searrow^{\mathrm{id}} & \Downarrow{\scriptstyle \mu} & \swarrow_{\mathrm{id}} & \\ & & T & & \end{array}$$

The first of these is called the *associativity square* while the two triangles in second diagram are called the *unit triangles*.

We call η the *unit*, and μ the *multiplication*. Let us look at a familiar example:

Example 7.2 (Power-set monad)**.** Let $\mathcal{P}$ be the power-set functor that we defined before. We define η as the natural transformation:

$$\eta : \mathrm{Id} \Rightarrow \mathcal{P},$$

with components that send elements to the singleton set corresponding to that element:

$$\eta_A : A \to \mathcal{P}(A),\ a \mapsto \{a\}.$$

We define μ as the natural transformation:

$$\mu : \mathcal{P}^2 \to \mathcal{P},$$

with components that send each set of sets, to the union of those sets.

$$\mu_A : \mathcal{P}(\mathcal{P}(A)) \to \mathcal{P}(A),\ \{B_1, B_2, \ldots\} \mapsto \bigcup B_i,$$

where $B_i \subseteq A$.

7.1.1 Adjunctions give rise to monads

Let (F, G, η, ϵ) be a unit-counit adjunction. We have a functor:

$$T \equiv GF : \mathcal{C} \to \mathcal{C}.$$

We can define a natural transformation:

$$\mu : T^2 \Rightarrow T, \mu_c \equiv G(\epsilon_{Fc}).$$

Let us show that (T, η, μ) indeed forms a monad, first the associtivity square:

$$\begin{array}{ccc} GFGFGF & \xRightarrow{\mu GF} & GFGF \\ {\scriptstyle GF\mu}\Downarrow & & \Downarrow{\scriptstyle \mu} \\ GFGF & \xRightarrow{\mu} & GF \end{array}$$

Looking at this diagram in terms of components and substituting in the definition of μ we obtain

$$\begin{array}{ccc} GFGFGFc & \xrightarrow{G(\epsilon_{FGFc})} & GFGFc \\ {\scriptstyle GFG(\epsilon_{Fc})}\downarrow & & \downarrow{\scriptstyle G(\epsilon_{Fc})} \\ GFGFc & \xrightarrow{G(\epsilon_{Fc})} & GFc \end{array}$$

written more suggestively we write: $a \equiv FGFc, b \equiv Fc$ and $\tilde{G} \equiv GFG$,

$$\begin{array}{ccc} \tilde{G}a & \xrightarrow{G(\epsilon_a)} & Ga \\ {\scriptstyle \tilde{G}(\epsilon_b)}\downarrow & & \downarrow{\scriptstyle G(\epsilon_b)} \\ \tilde{G}b & \xrightarrow{G(\epsilon_b)} & Gb \end{array}$$

such that the diagram reveals itself to be a naturality square under the function $f \equiv \epsilon_b : a \to b$ for the natural transformation $G\epsilon$.

For e.g. the left unit triangle we observe:

$$\begin{array}{ccc} GFc & \xrightarrow{\eta_{GFc}} & GFGFc \\ & {\scriptstyle \mathrm{id}_{GFc}}\searrow & \downarrow{\scriptstyle G(\epsilon_{Fc})} \\ & & GFc \end{array}$$

Which is just the second triangle identity of the adjunction at the object Fc.

7.1.2 Kleisli categories

Every monad defines a new category, called the *Kleisli category*.

Definition 7.3. Let $\mathcal{C}$ be a category, and let (T, η, μ) be a monad over this category. Then the Kleisli category $\mathcal{C}_T$ is the category where:

- The *objects* of $\mathcal{C}_T$ a_T correspond directly to the objects a of $\mathcal{C}$.
- The *arrows* of $\mathcal{C}_T$ are the arrows of the form $f : a \to Tb$ in $\mathcal{C}$, and will be denoted f_T. In other words,

$$\mathrm{Hom}_{\mathcal{C}_T}(a_T, b_T) \simeq \mathrm{Hom}_{\mathcal{C}}(a, Tb).$$

- Composition between two arrows $f_T : a_T \to b_T$ and $g_T : b_T \to c_T$ in $\mathcal{C}_T$ is given by:

$$g_T \circ_T f_T \equiv (\mu_c \circ Tg \circ f)_T.$$

- The *identity arrows* id_{a_T} are equal to $(\eta_a)_T$.

Let us show that this indeed forms a category. In particular we have to show that the composition operator is associative and unital. For the former, we look at the following situation

$$a_T \xrightarrow{f_T} b_T \xrightarrow{g_T} c_T \xrightarrow{h_T} d_T$$

the left associative and right associative expressions are:

$$\begin{aligned}(h_T \circ_T g_T) \circ_T f_T &= (\mu_d \circ Th \circ g)_T \circ_T f_T \\ &= (\mu_d \circ T(\mu_d \circ Th \circ g) \circ f)_T, \\ h_T \circ_T (g_T \circ_T f_T) &= h_T \circ_T (\mu_c \circ Tg \circ f)_T \\ &= (\mu_d \circ Th \circ \mu_c \circ Tg \circ f)_T,\end{aligned}$$

so it is enough to show that:

$$\mu_d \circ T\mu_d \circ T^2 h = \mu_d \circ Th \circ \mu_c,$$

which holds because of the associativity square and the naturality of μ:

$$\begin{aligned}\mu_d \circ T\mu_d \circ T^2 h &= \mu_d \circ \mu_{Td} \circ T^2 h \\ &= \mu_d \circ Th \circ \mu_c\end{aligned}$$

To show that it is e.g. left-unital we compute:

$$\mathrm{id}_{b_T} \circ_T f_T = (\mu_b \circ T(\eta_b) \circ f)_T = f_T$$

where we use the right unit triangle of the monad:

$$\mu_b \circ T\eta_b = \mathrm{id}_b$$

Understanding Kleisli composition can be a convenient stepping stone to understanding how to work with Monads in Haskell. The composition operator $\circ_T$ is usually denoted `>=>` (the fish operator) in Haskell.

7.1.3 Every monad is induced by an adjunction

Let $\mathcal{C}$ be a category, (T, η, μ) a monad over $\mathcal{C}$, and $\mathcal{C}_T$ the associated Kleisli category. Here, we will show that there are functors $F : \mathcal{C} \to \mathcal{C}_T$ and $G : \mathcal{C}_T \to \mathcal{C}$ so that $F \dashv G$ and T is equal to the monad induced by that adjunction.

We define:

$$\begin{array}{lll} F_T : \mathcal{C} \to \mathcal{C}_T, & a \mapsto a_T, & (f : a \to b) \mapsto (\eta_b \circ f)_T \\ G_T : \mathcal{C}_T \to \mathcal{C}, & a_T \mapsto Ta, & (f : a \to Tb)_T \mapsto \mu_b \circ Tf \end{array}$$

Let us check that e.g. F_T is actually a functor. Consider two arrows in $\mathcal{C}$, $f : a \to b$ and $g : b \to c$.

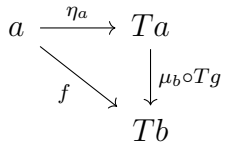
$$\begin{aligned} F_T(\mathrm{id}_a) &= (\eta_a)_T \equiv \mathrm{id}_{a_T} \\ F_T(g \circ f) &= (\eta_c \circ g \circ f)_T \\ F_T(g) \circ_T F_T(f) &= (\eta_c \circ g)_T \circ_T (\eta_b \circ f)_T = (\mu_c \circ T(\eta_c \circ g) \circ \eta_b \circ f)_T \end{aligned}$$

So we have to show that:

$$\mu_c \circ T(\eta_c) \circ Tg \circ \eta_b \stackrel{?}{=} \eta_c \circ g,$$

which is immediate from the right unit triangle, and the naturality of η.

Next we show that $F_T \dashv G_T$, in that (F_T, G_T, η) (we take the unit of the adjunction to be equal to the unit of the monad) forms an adjunction in the universal arrow sense. We have to show that for each $f : a \to Tb$ there is a unique $g_T : a_T \to b_T \equiv (g : a \to Tb)_T$ so that the following diagram commutes:



Using the left unit triangle, we obtain that it is sufficient and necessary to take simply $g_T \equiv f_T$!

The counit of the adjunction is given by $\epsilon_{b_T} \equiv (\mathrm{id}_{Tb})_T : (Ta)_T \to a_T$. We have $T \equiv G_T F_T$, and we have that

$$G_T(\epsilon_{F_T a}) = G_T(\epsilon_{a_T}) = G_T((\mathrm{id}_{Ta})_T) = \mu_a \circ T(\mathrm{id}_{Ta}) = \mu_a$$

as required.

7.2 Monads and functional programming

Because the brutal purity of Haskell is restrictive, we need non-standard tools to perform operations that we take for granted in imperative languages. In this section, we will explore what this means for some real world programs, and discover what problems and difficulties pop up. In particular, we will see how we can use monads to overcome some of these problems, by showing that functions of the type `a -> T b` are common, and hence that we are in need of a nice way to compose them.

7.2.1 IO

Consider the type of some functions in Haskell regarding input and output in Haskell:

```
print :: Show a => a -> IO ()
putStr :: String -> IO ()
getLine :: IO String
getChar :: IO Char
```

this allows us to do I/O as in the following snippet:

```haskell
main = do
  x <- getLine
  y <- getLine
  print (x ++ y)
```

Let us consider this snippet of code carefully. If `main` should *behave* purely, then it should return the same function every time. However, since we would like to support user input (`cin, scanf, getLine, ..`) so what should be its type if it should 'behave mathematically'? Similarly, for `print`, what would be its type? It should take a value, convert it to a string, and output this in a terminal somewhere. What is the *type* of *printing to screen*?

In Haskell, this is done using *IO actions*. This monadic style of doing IO is not limited to input/output for terminal, it can also be network related, file related, or mouse/keyboard input for a video game!

An IO action is a *value* with a type of `IO a`. We can also have an 'empty IO action', if the result is not used. The way to look at these actions is as a *recipe* of producing an `a`. While the actual value produced by the action depends on the outside world, the *recipe* itself is completely *pure*.

Let us consider our examples:

- The print function has the signature from a String to an IO action:

  ```haskell
  putStrLn :: String -> IO ()
  ```

  To print the value of any type, we precompose this function with `show :: a -> String`.

- The function `main` itself is an IO action! So the type is `main :: IO ()`.
- The `getLine` function is an IO action `getLine :: IO String`.

Case study: Handling input

Let us consider a very simple example using `getLine` and `print`.

```haskell
f :: Int -> Int
f x = 2 * x

-- attempt 1
main = print $ f (read getLine :: Int)
```

But this does not type check! First, the action `getLine` has type `IO String`, while `read` expects `String`. Then to work on the IO action, we want to *lift* `read` to take an `IO String` and produce an `IO Int`. This sounds like an `fmap`, and indeed `IO` provides fmap, it is a functor!

```
-- attempt 2
main = print $ f (read <$> getLine :: IO Int)
```

Next, we have that `f` expects an `Int`, not an `IO Int`, so we lift it again

```
-- attempt 3
main = print $ f <$> (read <$> getLine :: IO Int)
```

The `print`[1] statement we used here has signature `a -> IO ()`. Bringing this into the `IO` context using an `fmap` gives us:

```
fmap print :: IO a -> IO (IO ())
```

Since `main` should corespond to `IO ()`, we need either a way to remove a 'nested IO tag', or we need a function for functors that only lifts the first argument. In other words, let `F` be a functor, then we require either:

```
join :: F (F a) -> F a
(=<<) :: (a -> F b) -> (F a -> F b)
-- the above function is more commonly used with swapped arguments
-- and is then pronounced 'bind'
(>>=) ::  F a -> (a -> F b) -> F b)
```

Note, that we can define:

```
join :: F (F a) -> F a
join x = x >>= id
```

so that implementing >>= is enough. Conversely, we can also retrieve bind from `join` and `fmap`:

[1] `print` actually corresponds to `(putStrLn . show)` in Haskell

```
x >>= f = join (f <$> x)
```

Note also that we can pack an object inside an IO 'container':

```
return :: a -> IO a
```

Let us return to IO, and see what this notation gives us:

```
main = getLine >>= putStrLn
```

This code results in an empty action `IO ()`, so the 'bind' function can be used to chain `IO` operations together! For our little toy program we can `fmap` print into the `IO` context, and `join` the result afterwards to obtain:

```
main = join $ print <$> f <$> (read <$> getLine :: IO Int)
```

Using a more idiomatic style of writing this programming we get:

```
main = read <$> getLine >>= (\x -> print (f x))
```

which in `do`-notation becomes:

```
main = do
    x <- read <$> getLine
    print (f x)
```

To summarize, `>>=`, `join` and `return` allow us to **compose functions that may or may not require IO operations**.

7.2.2 Other examples

Now that we have seen how to compose functions of the form `a -> T b`, let us look at some other examples of contexts where this structure can be found.

Data structures

- `a -> Maybe b`: a function that **may fail**.
- `a -> [b]`: a function that **may produce zero or more results**.

Logging

All of `IO`, `Maybe` and `[]` may be seen as 'functional containers', let consider a different kind of example.

```
data Logger m a = Logger (a, m)
```

The data type `Logger` consists of a *composable log* (in the form of a monoid, e.g. `(String, (++))`) `m`, and an embedded value `a`.

- `a -> Logger String b`: a function that **may log a string**.

State

```
data State s a = State (s -> (a, s))
```

In this view, a value of type `State s a` is a function that takes some state, and produces an `a` in addition to a (possibly modified) state. For example, `s` could be some *environment* (say a `Map`) containing information that can be used to produce an `a`, and the state function can manipulate this `Map` when producing the `a`.

- `a -> State s b`: a function that **uses and/or manipulates a state**.

In these examples, the *contexts* are

- `Maybe`: failure that gets propagated
- `[]`: arbitrary number of results that are gathered
- `Logger s`: a log of type `s` that is maintained
- `State s`: a state of type `s` that is passed around

The bind `>>=` implementation for these *monads* pass around this context, and can change the control depending on the result after a step. For example, it can short-circuit a computation inside the `Maybe` monad in case some function fails.

7.2.3 The `Monad` type class

The triplets `(F, return, join)` that we have seen in this section, correspond to monads (T, η, μ) over the category of types. The type class in Haskell is defined as[2]:

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

We have seen that `>>=` can be defined in terms of `join`, which has the familiar type:

```
join :: m (m a) -> m a
```

Indeed, `return` corresponds to a natural transformation `Identity -> m`, while `join` corresponds to a natural transformation between `m m -> m`.

7.3 Exercises

Exercise 7.1. Show that the image of any commutative diagram under a functor F is again commutative.

Exercise 7.2. Show that G_T is a functor.

7.4 References

- https://golem.ph.utexas.edu/category/2012/09/where_do_monads_come_from.html
- 6.1 and parts of 6.3 and 6.4 of Mac Lane
- Blogs:
  - https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/
  - https://bartoszmilewski.com/2016/11/30/monads-and-effects/
  - http://www.stephendiehl.com/posts/monads.html
- Catsters

[2] We simplify the definition slightly here, the actual class also defines a `fail` method which is seen as an historical flaw

About IO:

- https://wiki.haskell.org/Introduction_to_IO

Some posts dealing specifically with Monads from a Haskell perspective:

- http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html
- https://bartoszmilewski.com/2013/03/07/the-tao-of-monad/
- http://www.stephendiehl.com/posts/adjunctions.html
- https://www.reddit.com/r/haskell/comments/4zvyiv/what_are_some_example_adjunctions_from_monads_or/

Chapter 8

Recursion and F-algebras

- Eilenberg-Moore category of algebras over a monad, can be used to show that every monad arises from an adjunction.
- Initial algebras, Lambek's theorem, `Fix f`, recursion.

Definition 8.1. Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. An **F-algebra** is a pair (a, α) where $a \in \mathcal{C}$ and $\alpha : Fa \to a$ is an arrow in $\mathcal{C}$. The object a is called the *carrier* of the algebra.

A homomorphism between F-algebras (a, α) and (b, β) is an arrow $h : a \to b$ such that the following diagram commutes:

$$\begin{array}{ccc} Fa & \xrightarrow{\alpha} & a \\ {\scriptstyle Fh}\downarrow & & \downarrow{\scriptstyle h} \\ Fb & \xrightarrow[\beta]{} & b \end{array}$$

For every endofunctor F, the collection of F-algebras together with homomorphisms of these F-algebras form a category which we will denote $\mathbf{Alg}_F$.

A fixed point for a function f is an x such that $f(x) = x$. Considering this, a sensible definition for a fixed point of an endofunctor is an object a such that $F(a) = a$, but we are a bit more lenient, and only require that $F(a) \simeq a$.

Definition 8.2. A **fixed point** of F is an algebra (a, α) for which α is an isomorphism.

Considering a *least fixed point*, we take inspiration from partially ordered sets, where the least point is an initial object, and define it as follows.

Definition 8.3. A **least fixed point** of F is an initial algebra (a, α), i.e. an algebra that is an initial object in the category $\mathbf{Alg}_F$.

An immediate issue that we have to resolve is to show that any *least* fixed point is indeed a fixed point.

Lemma 8.4 (Lambek)**.** Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor. If (t, τ) is initial in $\mathbf{Alg}_F$, then τ is an isomorphism.

Proof. Let (t, τ) be an initial object in $\mathbf{Alg}_F$, and consider the algebra $(Ft, F\tau)$. Since (t, τ) is initial there is a unique homomorphism $h : t \to Ft$ such that the following diagram commutes:

$$\begin{array}{ccc} Ft & \xrightarrow{\tau} & t \\ {\scriptstyle Fh}\downarrow & & \downarrow{\scriptstyle h} \\ F^2t & \xrightarrow{F\tau} & Ft \\ {\scriptstyle F\tau}\downarrow & & \downarrow{\scriptstyle \tau} \\ Ft & \xrightarrow[\tau]{} & t \end{array}$$

Here, the top square commutes because h is a homomorphism, and the bottom square commutes trivially. First, we note that by commutativity of this diagram, $h \circ \tau$ is a homomorhism between $(t, \tau) \to (t, \tau)$, and since (t, τ) is initial it is the unique homomorphism, i.e. the identity, and hence:

$$\tau \circ h = \mathrm{id}_t,$$

i.e. h is a right inverse to τ. To show that it is also a left inverse (and hence that τ is an isomorphism) we compute using the commutativity of the top square:

$$h \circ \tau = F\tau \circ Fh = F(\tau \circ h) = F(\mathrm{id}_t) = \mathrm{id}_{Ft}.$$

which shows that τ is an isomorphism, and hence that (t, τ) is a fixed point. □

Let (a, α) be an F-algebra. In the functional programming literature, the unique homomorphism from the initial algebra (t, τ) to (a, α) is called a *catamorphism* and is denoted $(\!|\alpha|\!)$.

Proposition 8.5 (Fusion law)**.**

$$h \circ f \simeq g \circ Fh \implies h \circ (\!|f|\!) = (\!|g|\!).$$

8.1 Determining the least fixed point

When does a least fixed point exist? Lambek's theorem implies that e.g. the $\mathcal{P}$ power-set endofunctor does not have an initial algebra (because $\mathcal{P}(X)$ is never isomorphic to X (Cantor's theorem).

Definition 8.6 (Polynomial functor)**.** Let $\mathcal{C}$ be a category with finite (co-)products. A **polynomial functor** from $\mathcal{C} \to \mathcal{C}$ is defined inductively as:

- The identity functor $\mathrm{Id}_{\mathcal{C}}$ is a polynomial functor.
- All constant functors $\Delta_c : \mathcal{C} \to \mathcal{C}$ are polynomial functors.
- If F and F' are polynomial functors, then so are $F \circ F'$, $F + F$ and $F \times F$.

8.1.1 Limits

- Cones
- Limits
- $'\omega$-limits

8.1.2 Limits in Set

- Initial algebra if F ω-cocontinuous...

8.2 Least fixed points in Haskell

In Haskell, the point of using F-algebras is to convert functions with signature:

```
alpha :: f a -> a
```

for a given functor `f`, to functions that look like:

```
alpha' :: Fix f -> a
```

where `Fix f` is the fixed point of `f`. Compared to our category theory notatation we have:

$$\texttt{alpha} \equiv \alpha$$
$$\texttt{alpha}' \equiv (\!|\alpha|\!)$$

So the universal property corresponding to a least fixed point in Haskell, expressed in Haskell, is the existence of a function that does this conversion of α to $(\!|\alpha|\!)$. Let us call it `cata`:

```
cata :: (f a -> a) -> Fix f -> a
```

Whenever you see a universal property like this in Haskell, and you want to find out what the type of `Fix f` should be, there is an easy trick, we simply define the type to have this universal property.

```
-- we look at
flip cata :: Fix f -> (f a -> a) -> a
-- which leads us to define
data Fix f = Fix { unFix :: (f a -> a) -> a }
-- now we have
unFix :: Fix f -> (f a -> a) -> a
-- so we can define
cata = flip unFix
```

Now we have our `cata` function, but it is of no use if `Fix f` is not inhabited. We want to be able to convert any value of type `f a` into a `Fix f` value. We first introduce the following equivalent description of `Fix f`:

```
-- for a fixed point, we have `x ~= f x`
-- the conversion between x and f x, where x == Fix' f
-- can be done using Fix' and unFix'
data Fix' f = Fix' { unFix' :: f (Fix' f) }
-- for Fix, we can define cata as:
cata' :: Functor f => (a -> f a) -> Fix' f -> a
cata' alpha = alpha . fmap (cata' alpha) . unFix'
-- to show that we can convert between the Fix and Fix':
iso :: Functor f => Fix' f -> Fix f
iso x = Fix (flip cata' x)
invIso :: Functor f => Fix f -> Fix' f
invIso y = (unFix y) Fix'
```

`Fix' f` is sometimes written as μF (or `Mu f` in Haskell).

So, in summary, catamorphing an algebra can be done recursively using:

```haskell
type Algebra f a = f a -> a
data Fix f = Fix { unFix :: f (Fix f) }

cata :: Functor f => Algebra f a -> Mu f -> a
cata a = f . fmap (cata a) . unFix
```

8.3 Using catamorphisms in Haskell

To give an interpretation of `cata`, we first show how we usually construct values of `Fix f`. Say we have a very simple expression language, with constants and addition:

```haskell
data Expr' = Cst' Int | Add' (Expr', Expr')
-- we introduce 'holes' in the add, instead of recurring
data ExprR b = Cst Int | Add (b, b)
-- we reobtain the original expression by finding the 'fixed point'
type Expr = Fix' ExprR
-- we make wrappers to construct values of type Expr
cst = Fix' . Cst
add = Fix' . Add
-- we turn ExprR into a fucntor
instance Functor ExprR where
  fmap _ (Cst c) = Cst c
  fmap f (Add (x, y)) = Add (f x, f y)
```

We can use this in the following way:

```haskell
eval = cata algebra where
    algebra (Cst c) = c
    algebra (Add (x, y)) = x + y

printExpr = cata algebra where
    algebra (Cst c) = show c
    algebra (Add (x, y)) = "(" ++ x ++ " + " ++ y ++ ")"
```

And it allows us to perform our optimizations independently, during the same traversal, e.g.:

```haskell
optimizeLeftUnit :: ExprR Expr -> Expr
optimizeLeftUnit (Add (Fix' (Cst 0), _)) = cst 0
```

```haskell
optimizeLeftUnit e = Fix' e

optimizeRightUnit :: ExprR Expr -> Expr
optimizeRightUnit (Add (_, Fix' (Cst 0))) = cst 0
optimizeRightUnit e = Fix' e

comp f g = f . unFix' . g
optimize = cata (optimizeLeftUnit `comp` optimizeRightUnit)
```

8.4 References

- http://files.meetup.com/3866232/foldListProduct.pdf
- https://deque.blog/2017/01/17/catamorph-your-dsl-introduction/
- https://www.schoolofhaskell.com/user/edwardk/recursion-schemes/catamorphisms
- https://www.schoolofhaskell.com/user/bartosz/understanding-algebras
- http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt
- http://web.cecs.pdx.edu/~sheard/course/AdvancedFP/notes/CoAlgebras/Code.html

Part II

Advanced theory and applications

Chapter 9

Adjunctions in Haskell

The only monad over **Hask** arising from an adjunction that goes through **Hask** itself is the `State` monad:

```
(, f) -| (->) e
```

You can show this using: https://en.wikipedia.org/wiki/Representable_functor#Left_adjoint. Witnessed by `curry` and `uncurry`.

We have:

```
(-> e) -| (-> e)
```

as an adjunction through $\mathbf{Hask}^{\mathrm{op}}$. Witnessed by `flip`. This leads to the continuation monad, which we should talk about.

- http://www.stephendiehl.com/posts/adjunctions.html

Chapter 10

Lenses; Yoneda, adjunctions and profunctors

- https://skillsmatter.com/skillscasts/4251-lenses-compositional-data-access-a
- https://github.com/ekmett/lens

Chapter 11

Purely functional datastructures

- http://apfelmus.nfshost.com/articles/monoid-fingertree.html
- https://www.amazon.com/Purely-Functional-Structures-Chris-Okasaki/dp/0521663504

Chapter 12

Applicative functors

Applicative ~= Monoidal

Is strong lax functor

- McBride, Paterson; Applicative Programming with Effects
  - http://www.staff.city.ac.uk/~ross/papers/Applicative.pdf

Chapter 13

Monad transformers

'Translation of a monad along an adjunction'

- https://oleksandrmanzyuk.files.wordpress.com/2012/02/calc-mts-with-cat-th1.pdf

Chapter 14

Proof assistants

Curry-Howard isomorphism

Chapter 15

Further Ideas

15.1 Limits and colimits

15.2 Ends and co-ends

15.3 'Theorems for free!'

15.4 'Fast and loose reasoning is morally correct'

- Note that `newtype` and bottom cause issues.

15.4.1 References

- About **Hask**: http://www.cs.ox.ac.uk/jeremy.gibbons/publications/fast+loose.pdf
- http://math.andrej.com/2016/08/06/hask-is-not-a-category/
- https://wiki.haskell.org/Newtype

15.5 Homotopy type theory

15.6 Quantum computations?

(Bert Jacobs)

15.7 Haskell tricks and gems

- https://deque.blog/2016/11/27/open-recursion-haskell/

Chapter 16

Literature

16.1 Blogs

1. *Bartosz Milewski*: "Category Theory for Programmers", a blog post series that gives a good overview of interesting topics. https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/

16.2 Papers

2. Free theorems: http://ttic.uchicago.edu/~dreyer/course/papers/wadler.pdf (also Reynold: http://www.cse.chalmers.se/edu/year/2010/course/DAT140_Types/Reynolds_typesabpara.pdf).
3. Recursion as initial objects in F-algebra: http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt

16.3 Books

1. Conceptual Mathematics: A first introduction to categories.
2. S. Mac Lane, Category Theory for the working mathematician
3. Barr and Wells, Category Theory for Computer Scientists
4. E. Riehl, Category theory in context,
5. T. Leinster, Basic Category Theory
6. J. van Ooosten, Basic Category Theory

Part III

Exercises

Parser

This exercise is based on the parser exercises of (1) and the blog post series of evaluating DSLs (spoilers in the article!) (2).

Description

The goal of this exercise is to parse, and evaluate expressions such as:

```
"((x + 3) * (y + 5))"
"(((x + 3) * (y + 5)) * 5)"
"(x + y)"
...
```

it is up to you to define precisely the rules of this language, and to enforce (or not) the use of parentheses.

Preliminaries

Assume that we have the following definitions:

```haskell
type Id = String

data OperatorType = Add | Multiply
  deriving (Show, Eq)

data Expression =
  Constant Int
  | Variable Id
  | BinaryOperation OperatorType (Expression, Expression)

data Parser a = Parser { runParser :: String -> Maybe (a, String) }
```

A) Parser

1. Implement:

   ```
   charParser :: Char -> Parser Char
   ```

2. Implement:

   ```
   satisfy :: (Char -> Bool) -> Parser Char
   satisfy predicate = ...
   -- such that
   charParser c = satisfy (== c)
   ```

   Useful predicates on characters are `isAlpha`, `isAlphaNum`, `isDigit`, `isSpace`, and are found in the `Data.Char` library.

3. Implement:

   ```
   intParser :: Parser Int
   ```

   (possibly) useful library functions are:

   ```
   null :: [a] -> Bool
   read :: Read a => String -> a -- specialize to Int
   span :: (a -> Bool) -> [a] -> ([a], [a])
   ```

4. Provide instances for `Parser` for the following type classes:

   - `Functor`: (`<$>`) *given a function* `a -> b` *and a parser for* `a`*, return a parser for* `b`*.*
   - `Applicative`: (`<*>`) *given a parser for a function* `a -> b` *and a parser for* `a`*, return a parser for* `b`*.*
   - `Alternative`: (`<|>`) *given parsers for* `a` *and* `b`*, try to parse* `a`*; if and only if it fails, try to parse* `b`*.*

   Hint: Use the corresponding instances of `Maybe`*. It is also a good exercise to implement these instances for* `Maybe` *yourself.*

5. Implement:

   ```
   oneOrMore :: Parser a -> Parser [a]
   zeroOrMore :: Parser a -> Parser [a]
   ```

   Use the alternative instance of `Parser`. In fact, these functions are already implemented for you in the Alternative type class as `many` and `some` respectfully.

   Hint: implement both in terms of the other. For example, `oneOrMore` *can be seen as parse one, then parse zero or more.*

6. Implement

   ```
   spaces :: Parser String
   ```

   that parses zero or more whitespace characters (use `isSpace`).

7. Implement

   ```
   idParser :: Parser Id
   ```

   A valid identifier is (in most language) a string that starts with an alpha character, followed by zero or more alphanumeric characters (remember to use the character predicates available!).

8. Implement

   ```
   operatorParser :: Parser OperatorType
   ```

9. Combine the different parsers that you have made to make an expression parser:

   ```
   expressionParser :: Parser Expression
   ```

   It may be useful for debugging to implement `show` for `Expression`:

   ```
   instance Show Expression where
       -- show :: Expression -> String
       show expr = ...
   ```

   Also look at the functions `(*>)` and `(<*)` for `Applicative` instances, which ignore the result of a computation but keep the side effect (use this to ignore whitespace).

B) Evaluation

We define the `Environment` as a map that holds (integer) values for variables.

```
type Environment = Map Id Int
```

`Map` is found in the `Data.Map` library. See the documentation for usage.

1. Implement:

```
evaluate :: Environment -> Expression -> Maybe Int
```

2. Implement:

   ```
   optimize :: Expression -> Expression
   ```

   for example, `(0 + x)` can be replaced with just `x`, and `(1 + 3)` can just be evaluated to produce `4`, and so on (think of other optimizations).

3. Implement:

   ```
   partial :: Environment -> Expression -> Expression
   ```

   that replaces all the variables in the epxression with those that have values in the environment, and leaves the others intact.

4. Observe that you can implement `evaluate` in terms of `partial` followed by `optimize`, and do this.

5. Make a function:

   ```
   dependencies :: Expression -> [Id]
   ```

   returning the variables that occur in expression. Use the `Data.Set` library along with the functions `singleton, union, empty, toList`.

6. Use `dependencies` to improve your error messages by implementing a function

   ```
   result :: Expression -> Either String Int
   ```

   That returns the result of an expression, or a string containing an error message along with the dependencies that are missing.

C) Monadic parser

1. Write the `Monad` instance of `Parser`.

2. Observe that `do`-notation for the Parser reads very naturally:

   ```
   threeInts :: Parser [Int]
   threeInts = do
     x <- parseOneInt
     y <- parseOneInt
     z <- parseOneInt
   ```

```
    return [x, y, z]
    where
      parseOneInt = spaces *> intParser
```

D) Catamorphisms

We will revisit our parser when we talk about ***catamorphisms****.*

References

- (1): http://cis.upenn.edu/~cis194/spring13/lectures.html
- (2): https://deque.blog/2017/01/17/catamorph-your-dsl-introduction/

Monads

A) IO: Hangman

Taken from (1)

Description

The goal is to make an interactive 'hangman' game in Haskell, so that:

```
./hangman
Enter a secret word: *******
Try to guess: h
h______
Try to guess: ha
ha___a_
Try to guess: hang
hang_an
Try to guess: hangman
You win!!!
```

Preliminaries

Assume that we have the following definitions:

```
...
```

1. Implement: ...

B) State: Simulating Risk battles

Taken from (2)

Description

Simulate Risk

Preliminaries

Assume that we have the following definitions:

```
...
```

References

- (1): http://www.haskellbook.com
- (2): CIS194

Folds

For this exercise it is good to hide the fold operations from the Prelude so that you can implement them yourself.

```
import Prelude hiding (foldr, foldl)
```

A) Lists

1. Implement the functions:

   ```
   foldl :: (b -> a -> b) -> b -> [a] -> b
   foldr :: (a -> b -> b) -> b -> [a] -> b
   ```

2. Implement the following functions on lists using a fold:

   ```
   sum :: Num a => [a] -> a
   product :: Num a => [a] -> a
   length :: Num b => [a] -> b
   and :: [Bool] -> Bool
   or :: [Bool] -> Bool
   elem :: Eq a => a -> [a] -> Bool
   min :: (Bounded a, Ord a) => [a] -> a
   max :: (Bounded a, Ord a) => [a] -> a
   all :: (a -> Bool) -> [a] -> Bool
   any :: (a -> Bool) -> [a] -> Bool
   concat :: [[a]] -> [a]
   reverse :: [a] -> [a]
   filter :: (a -> Bool) -> [a] -> [a]
   map :: (a -> b) -> [a] -> [b]
   ```

3. use foldMap to implement the following functions

```
sum :: Num a => [a] -> a
product :: Num a => [a] -> a
concat :: [[a]] -> [a]
asString :: Show a => [a] -> String
```

B) Folds over either, maybe and binary trees

1. Implement:

   ```
   foldm :: (a -> b -> b) -> b -> Maybe a -> b
   folde :: (a -> b -> b) -> b -> Either c a -> b
   ```

   Other useful 'fold-like' functions of `Maybe` and `Either` are

   ```
   maybe :: (a -> b) -> b -> Maybe a -> b
   either :: (a -> c) -> (b -> d) -> Either a b -> Either c d
   ```

   Implement them. They are also in the prelude.

2. Define a binary tree as:

   ```
   data Tree a = Node (Tree a) a (Tree a) | Leaf
   ```

   and implement the function

   ```
   foldt :: (a -> b -> b) -> b -> Tree a -> b
   ```

   using this implement e.g.

   ```
   sumTree :: Num a => Tree a -> a
   productTree :: Num a => Tree a -> a
   ```

C) Peano numbers

Modelled after section 1.2 of 'Algebra of programming' from Bird and de Moor.

Natural numbers can be represented a la Peano as:

```
data Nat = Zero | Succ Nat
```

Or mathematically, we can say that any natural number can be represented recursively as $k = 0 \sqcup (n + 1)$, where n is again a natural number. Using this notation we can write a typical recursion formula:

$$f(m) = \begin{cases} c & \text{if } m = 0 \\ h(f(n)) & \text{if } m = n + 1 \end{cases}$$

or in code:

```
f :: Nat -> b
f Zero = c
f (Succ x) = h (f x)
```

Here, `f` is completely determined by the functions `h :: b -> b` and `c :: b`.

1. Write a function `foldn` that encapsulates this pattern:

   ```
   foldn :: (b -> b) -> b -> Nat -> b
   -- to see the similarity with foldr, write it as
   -- foldn :: (() -> b -> b) -> b -> Nat -> b
   -- (the discrepency is because `:k Nat = *`)
   ```

2. Implement the following functions using foldn

   ```
   sum :: Nat -> Nat -> Nat
   product :: Nat -> Nat -> Nat
   exponential :: Nat -> Nat -> Nat
   factorial :: Nat -> Nat
   fibonacci :: Nat -> Nat
   ```

   It may be convenient during testing to make some aliases for numbers:

   ```
   zero = Zero
   one = Succ Zero
   two = Succ one
   three = Succ two
   ```

   and to implement `instance Show Nat where ...`.

   Hint: Use `(Num, Num)` as `b` for `fact` and `fib`, and compose with `snd`.

3. Using `foldn`, implement:

```haskell
square :: Nat -> Nat
-- `last p n` returns the last natural number <= n that satisfies p
last :: (Nat -> Bool) -> Nat -> Nat
```

4. The Ackermann function is defined as:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

   Implement `curry ack` using `foldn`, where `ack :: (Nat, Nat) -> Nat`.

D) Finding distinct elements of a list

From the Data61 Haskell course

Recall the `State` monad, defined as `data State s a = State { runState :: s -> (a, s) }`.

```haskell
import qualified Data.Set as S
```

1. Implement a function

   ```haskell
   filtering :: Applicative f => (a -> f Bool) -> [a] -> f [a]
   ```

   that takes a list, and an *effectful predicate*, and produces an effectful list containing only those elements satisfying the predicate.

2. Using `filtering`, with `State (Set a) Bool` as the `Applicative f`, implement:

   ```haskell
   distinct :: Ord a => [a] -> [a]
   ```

Appendix A

Short introduction to Haskell

Here we will give an introduction to programming using Haskell. It will not be an extensive introduction, in fact it will be very brief. However, studying this section should be enough to allow you to follow along with the rest of the text even if you have no experience with Haskell. You are encouraged to look for additional material online, see also the references at the end of this section. You are assumed to have access to the Glasgow Haskell Compiler (GHC) and its interactive REPL GHCi.

To follow along, open `ghci` and play around with the code snippets that we provide.

We will dicuss te topics suggested by the NICTA Haskell course[1].

Values and assignment

A value can be assigned to a variable as follows:

```
let x = 'a'
let y = 3
let xs = [1,2,3]
let f x = x * x
let g x y = x * y
```

We note that these variables are only valid inside an expression, using a:

```
let [variable = value] in [expression]
```

syntax, but you can also use this style of variable definition inside `ghci`.

[1] https://github.com/NICTA/course

Type signatures

In GHCi, you can see the type of the variable using:

```
:t x -- x :: Char
:t y -- y :: Num a => a
:t xs -- g :: Num t => [t]
:t f -- f :: Num a => a -> a
:t g -- g :: Num a => a -> a -> a
```

Here `::` means "has the type of".

The `->` in a type is right associative, i.e.

```
a -> a -> a == a -> (a -> a)
```

and so on. You can read this as 'for an `a`, we get a function from `a` to `a`'.

Functions are values

Functions can be used as arguments to other (higher order) functions. E.g.

```
:t (2*) -- Num a => a -> a
map :: (a -> b) -> [a] -> [b]
map (2*) xs -- [2,4,6]
```

Here we *map* a function over a list.

Functions take arguments

On thing to notice about the `map` example, is that it although it is a function that technically takes a single argument (and produces a function from a list to a list), it can also be viewed as a function of two arguments. We will not explicitely distinguish between these two views.

We can also make anonymous 'lambda' functions:

```
map (\x -> x * x) xs -- [1,4,9]
```

The backslash is inteded to look like a λ.

Functions can be composed

In Haskell there are three alternative ways of composing functions (to prevent overuse of parenthesis):

```
g(f 123)
g $ f 123
(g . f) 123
```

Here, `$` makes sure that all the functions on the right have been evaluated before statements on the left come in to play.

Infix operators

An operator starts with a non-alphanumeric character, e.g. `+`, `++`, `>>=`, `:` are all operators, and they use *infix* notation by default. For example:

```
1 + 2 -- 3
[1,2] ++ [3,4] -- [1,2,3,4]
1 : [2,3] -- [1,2,3]
```

To use them with *prefix* notation, we surround them with parenthesis:

```
(+) 1 2 -- 3
```

Any function (which by default uses prefix notation) can be used infix as well using backticks:

```
let f x y = x * x + y * y
2 `f` 3 -- 13
```

this can make code significantly more clear when defining e.g. operations that act on multiple lists, sets, or maps.

Polymorphism

We already saw the type signature of `map`:

```
map :: (a -> b) -> [a] -> [b]
```

This is an example of a polymorphic function, it is defined for any type `a` and `b`. We refer to these 'wildcard types' as *type variables*. These always start with a lowercase letter.

Data types

To work with custom data structures, we create new *data types*. These are declared as follows:

```
data DataTypeName a b = Zero | One a | One' b | Both a b
```

A data type is declared using the `data` keyword, and the *type constructor* is given a name (here `DataTypeName`). A data type depends on a number of type variables, here `a` and `b`. After the `=` sign, there are zero or more *data constructors*, here `Zero`, `One`, `One'`, and `Both`, each depending on one or more of the type variables of the type constructor and separated by a pipe `|`.

Data constructors can be used for *constructing* a value of the data type, or for pattern-matching on values of the data type (i.e. retrieve which constructor was used to construct the given value).

Type classes

Type classes are a way to have *ad-hoc polymorphism* in Haskell, while the ordinary polymorphic functions discussed before are *parametric*. This means that we can have different behaviour for different types. Type classes are introduced as follows:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Here, we state that in order for a type `a` to be part of the *type class* `Eq`, we have to implement an equality function with the given signature. We can then restrict functions definitions to only work on types in this type class in the following manner:

```
(!=) :: Eq a => a -> a -> Bool
x != y = not (x == y)
```

Monoids, Functors, Applicative and Alternative

Here we give a whirlwind tour of some interesting type classes used in Haskell, the majority of the category theory that we will discuss will explain the mathematical

background and uses of these typeclasses in detail, here we summarize the resulting classes as a reference. Feel free to skip or skim them, and to come back after studying the material presented in later chapters.

Monoid

Many types have one (or even multiple) *monoidal* structure, which means that it is possible to combine two elements to a single element, and that this way of combining has some special (but common) properties.

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m -- infix operator alias: <>
```

The implementations depend on the type `m`, but when implementing a Monoid instance, it is the task of the implementor to adher to the following laws:

```
-- forall x, y, z :: m
x <> mempty == x -- identity
mempty <> x == x
(x <> y) <> z == x <> (y <> z) -- associativity
```

For example, the following are all possible Monoid instances (given as `(m, mempty, mappend)`):

- `(Int, 0, (+))`
- `(Int, 1, (*))`
- `(Int32, minBound, max)`
- `(Int32, maxBound, min)`
- `(String, "", (++))`
- `(Maybe, Nothing, (<|))`, here `(<|)` denotes the binary function that yields the left-most non-`Nothing` value if anything (obviously there is also a right-most equivalent `(|>)`).

and so on.

Functor

A functor can take an 'ordinary' function, and apply it to a `context`. This context can be a list, the result of a computation that may have failed, a value from input/output and so on. You can also view the functor itself as the context.

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b -- infix operator alias: <$>
```

Again, each instance should satisfy certain laws. For `Functor`, these are:

```haskell
-- forall f, g :: a -> b
fmap id == id
fmap (f . g) == fmap f . fmap g
```

For example, the `List` functor is implemented as:

```haskell
instance Functor [] where
    fmap = map
```

Or the 'composition' functor:

```haskell
instance Functor ((->) c) where
    -- fmap :: (a -> b) -> (c -> a) -> (c -> b)
    fmap = (.)
```

Applicative

The most obvious use of applicative is to lift functions of multiple arguments into a context. If we have a function of multiple arguments like:

```haskell
g :: a -> b -> c
```

Then we can't just list it into a functor (context), since we would obtain:

```haskell
fmap g :: f a -> f (b -> c)
```

If we compose it with a function that has the signature

```haskell
apply :: f (b -> c) -> f b -> f c
```

then we obtain:

```
apply . fmap g :: f a -> f b -> f c
```

If we implement `apply`, then we can lift functions with an arbitrary number of arguments (by iteratively calling `apply` after `fmap`). A functor with `apply` is called an *applicative functor*, and the corresponding type class is:

```
class Functor f => Applicative f where
    pure :: a -> f a
    ap :: f (a -> b) -> f a -> f b -- infix operator alias: <*>
```

we see that additionally, `pure` is introduced as a way to put any value into an applicative context. Any applicative instance has to satisfy the following laws:

```
-- forall v, w :: a; x, y, z :: f a; g :: a -> b
pure id <*> x = x -- identity
pure (.) <*> x <*> y <*> z = x <*> (y <*> z) -- composition
pure g <*> pure v = pure (g v) -- homomorphism
y <*> pure v = pure ($ y) <*> v -- interchange
```

Alternative

Now that we have introduced some terminology, we can introduce Alternative functors as *giving an applicative context a monoidal structure*.

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
```

For example, for `Maybe` we can say:

```
instance Alternative Maybe where
    empty = Nothing
    Nothing <|> right = right
    left <|> _ = left
```

Or for `List` we have the standard concatenation monoid:

```
instance Alternative [] where
    empty = []
    (<|>) = (++)
```

Monads

A functor lets you lift functions to the functorial context. An applicative functor lets you untangle functions caught in a context (this can be e.g. an artifact of currying functions of multiple arguments) to functions in the functorial context. Another useful operation is to compose functions whose *result lives inside the context*, and this is done through bind `>>=` (with its flipped cousin `=<<`).

To illustrate the similarities between the typeclasses `Functor => Applicative => Monad`:

```
(<$>) :: (a -> b)   -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
(=<<) :: a -> f b   -> f a -> f b
```

For us, the interesting part of the definition is:

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

The default implementation of `return` is to fall back on `pure` from applicative. The bind operation has to satisfy the following laws:

```
-- forall v :: a; x :: m a; k :: a -> m b, h :: b -> m c
return v >>= k = k v
x >>= return = x
m >>= (\y -> k y >>= h) = (m >>= k) >>= h
```

Thus bind takes a monadic value, and shoves it in a function expecting a non-monadic value (or it can bypass this function completely). A *very* common usage of bind is the following.

```
x :: m a
x >>= (\a -> {- some expression involving a -})
```

which we can understand to mean that we *bind* the name `a` to whatever is inside the monadic value `x`, and then we can reuse it in the expressions that follow. In fact, this is so common that Haskell has convenient syntactic sugar for this pattern called `do`-notation. This notation is recursively desugared according to the following rules (taken from Stephen Diehl's "What I wish I knew when learning Haskell"):

```
do { a <- f; m }  ~>  f >>= \a -> do { m }
do { f; m }  ~>  f >> do { m }
do { m }  ~>  m
```

Curly braces and semicolons are usually omitted. For example, the following two snippets show the sugared and desugared version of `do`-notation:

```
do
  a <- f
  b <- g
  c <- h
  return (a, b, c)

f >>= \a ->
  g >>= \b ->
    h >>= \c ->
      return (a, b, c)
```

Monads can be used to do all kinds of things that are otherwise relatively hard to do in a purely functional language such as Haskell:

- Input/output
- Data structures
- State
- Exceptions
- Logging
- Continuations (co-routines)
- Concurrency
- Random number generation
- ...

Folds

Folds[2] are an example of a *recursion scheme*. You could say that (generalized) folds in functional languages play a similar role to `for`, `while`, ... statements in imperative languages. There are two main higher-order functions for *folds* in Haskell:

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldr :: (a -> b -> b) -> b -> [a] -> b
```

[2] A *fold* is also known as *reduce* or *accumulate* in other languages

Here, `foldl` associates to the left, and `foldr` associates to the right. This means:

```
foldl (+) 0 [1, 2, 3]
-- ~> ((1 + 2) + 3)
foldr (+) 0 [1, 2, 3]
-- ~> (1 + (2 + 3))
```

E.g. `foldr` can be implemented as:

```
foldr f x xs = case xs of
    [] -> x
    (y:ys) -> y `f` (foldr f x ys)
```

Foldable

Lists are not the only data structure that can be folded. A more general signature of `foldr` would be:

```
foldr :: (a -> b -> b) -> b -> t a -> b
```

where `t` is some *foldable data structure*. There is a type class, with the following core functions:

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

Only one of these two functions has to be implemented, the one can be retrieved from the other. Here, `foldMap` maps each element of a foldable data structure into a monoid, and then uses the operation and identity of the monoid to fold. There is also a general `fold` method for each `Foldable`:

```
fold :: Monoid m => t m -> m
-- e.g. for list, if `x, y, z :: m`:
fold [x, y, z]
-- ~> x <> y <> z <> mempty
```

The more 'natural' fold in Haskell is `foldr`, to understand why we should look at the difference between `cons`-lists and `snoc`-lists:

```
List a  = Empty  | Cons a (List a) -- 'cons'
List' a = Empty' | Snoc (List a) a -- 'snoc'
```

The standard Haskell lists `[a]` are `cons`-lists, they are built *from the back of the list*. The reason `foldr` is more natural for this type of list is that the recursion structure follows the structure of the list it self:

```
h = foldr (~) e
-- h [x, y, z] is equal to:
h (x : (y : (z : [])))
--   |    |    | |
--   v    v    v v
--(x ~ (y ~ (z ~ e)))
```

This can be summarized by saying that a `foldr` *deconstructs* the list, it uses the shape of the construction of the list to obtain a value. The `(:)` operation gets replaced by the binary operation `(~)`, while the empty list (base case) is replaced by the *accumulator* `e`.

As a special case, since the value constructors for a list are just functions, we can obtain the identity operation on lists as a fold:

```
id :: [a] -> [a]
id == foldr (:) []
```

References

If you want to learn Haskell, the following resources are helpful as a first step:

- 5 minute tutorial to get an idea:
  - https://tryhaskell.org/
- The wiki book on Haskell is quite good:
  - https://en.wikibooks.org/wiki/Haskell
- There is an excellent accessible Haskell book coming out soon, but it can be found already:
  - http://haskellbook.com/
- A cult-classic Haskell book:

  - http://learnyouahaskell.com/chapters

- If you are looking to do exercises, there is a guide to different courses available here:
  - https://github.com/bitemyapp/learnhaskell
- A handy search engine for library functions is Hoogle:
  - https://www.haskell.org/hoogle/
- Advanced topics for Haskell:
  - http://dev.stephendiehl.com/hask/