Bachelor thesis
Computing Science

Radboud University

Abstract Nonsense and Programming:
Uses of Category Theory in Computer Science

Author:
Koen Wösten
K.Wosten@student.ru.nl
s4787838

Supervisor:
dr. Sjaak Smetsers
S.Smetsers@cs.ru.nl

Second assessor:
prof. dr. Herman Geuvers
H.Geuvers@cs.ru.nl

June 29, 2020

Abstract

In this paper, we try to explore the various applications category theory has in computer science and see how from this application of category theory numerous concepts in computer science are analogous to basic categorical definitions. We will be looking at three uses in particular: Free theorems, recursion schemes and monads to model computational effects. We conclude that the applications of category theory in computer science are versatile and many, and applying category theory to programming is an endeavour worthy of pursuit.

Contents

Preface

The original motivation for embarking on the journey of writing this was to get a better grip on categorical ideas encountered as a student assistant for the course *NWI-IBC040 Functional Programming* at Radboud University. When another student asks ”What is a monad?” and the only thing I can explain are its properties and some examples, but not its origin and the framework from which it comes. Then the eventual answer I give becomes very one-dimensional and unsatisfactory.

With this thought I set out on the journey of learning category theory; and it wasn't immediately clear why involving category theory was useful, nor was it clear how understanding categories, morphisms, functors and monads actually lead to building better systems.

Maybe it is because it is not really what category theory provides.

The relevance of the theory isn't really debatable, as with enough trying you could make anything relevant to a particular subject. If somebody tried hard enough they could make teddy bears relevant to quantum mechanics. However, whether it is worthwhile, is a completely different question. Rather than asking how category theory is relevant to computer science, we might need to ask how it is better than what we already have. Or at least what nuts and bolts it has to offer, its unique qualities that other theories don't boast.

So then, when is an abstraction, such as category theory useful? A good abstraction allows us to see patterns and give new insights. Whether an abstraction is good is generally only known after a while. If it did give many useful results, then it probably was. Which leaves us with the goal of the paper; we will explore what uses category theory has come to offer in the realm of computer science.

Hence the aim of this paper is not to push for a specific application of category theory – a task better undertaken by more experienced practitioners. Instead, I would like to familiarise the reader with the fundamental vocabulary and highlight a few of the more prominent uses in the field of computer science. The paper is meant to be self-containing. Therefore an overview of the theory is given in the first part. In the second part, we illustrate a few concrete uses of category theory.

I would like to thank my thesis supervisor Sjaak Smetsers for his feedback and his thoughts in establishing the current format of the paper.

Introduction

Category theory was originally constructed by Samuel Eilenberg and Saunders Mac Lane in the 1940s to study the relation between homology and algebra [10]. In the following decades, category theory underwent rapid development to mature into its own autonomous branch of mathematics, studied on its own as well as used as a unified language to express mathematical ideas linking different fields [15] [18].

Category theory, amongst other theories, is a study of structure, but what makes it distinct is that in some way the concept of a category can abstract a *kind of mathematics*. In particular, it provides a language that unifies many concepts in different parts of math.

Category theory's "thing" is that it generalizes structure. To understand this, we first take a step back and see what "structure" means. Just pick up a dictionary, and search for the definition of "structure". We do the same, for example, Merriam-Webster dictionary defines structure like the following:

> "The aggregate of elements of an entity in their relationships to each other" [1, (5)]

Whichever definition found always speaks of elements, parts, components or something of the sort. Simply put, structure arises only when we compose parts. Now category theory is all about composition as you will find in the coming sections, so it isn't astounding to hear that category theory becomes a convenient way to express underlying structure of all kinds.

In specific we try to structure mathematical theories in such a way that it makes sense to us. Only theories that are palatable and well learnable are theories that hold any significance in the long run, inasmuch as something that others can not learn, will likely not be learned. Not entirely convenient is that we learn using our brains, those of a naked monkey, adjusted to survive, hunt and flee. We are not particularly fine-tuned to do math. We are simply not made to think abstractly, we are made to throw spears, find food and run when facing danger. Our ability to think abstractly is a rather newly acquired ability, just a fraction in the span of our whole evolution. Since it is so new it is also still rather primitive, we accept that it is simply not our strongest suit. Ultimately, theories that work for us, those in which we can reason, are those tuned to our primitive brains. This explains the fact that theories generally build from basic concepts easily graspable on their own, combine these concepts, alter previous definitions just slightly and give just the appropriate amount of "new" stuff such that it is still manageable.

But what does this have to do with category theory? Well... category theory encompasses the idea of composition, the process of building things using smaller things. Which is exactly the way our primitive brains want to think! So, of course, it generalizes all our other theories! Because we made those exactly so that we can reason about individual parts, and compose our ideas to form grander ideas.

At the end of the day, the definitions are the important part of category theory. Once the definitions make sense, then the rest of the system makes sense. Category theory is all about mastering these definitions so that one can identify categorical structures in practice. The definitions thus far have had decent results: With category theory, mathematicians have discovered just how much all branches of mathematics have in common. They were then joined by computer scientists who discovered that programming can also be described by category theory [30] [29] [33].

Title of the paper might require an explanation: The term "abstract nonsense", is sometimes used jokingly to refer to a proof that relies on categoric-principles. It might even refer to the study of category theory itself. In any case it is *not* meant derogatory. The term actually pre-dates the notion of a category. In [19] Saunders Mac Lane wrote (referring to his first joint paper with Eilenberg [9]): "it introduced the very abstract idea of a 'category'—a subject then called 'general abstract nonsense'!"

Part I

Basics

The following sections of part 1 present a brief tutorial on the basic concepts of category theory. The objective of this part is to give you enough of an understanding/grasp of category theory to understand the rest of the paper. The paper assumes no prior knowledge. As the goal of the complete paper is to show the capabilities of category theory, this part is only used as a tool to understand the rest of the paper.

⌈1⌋ Categories

Undoubtedly, the notion of a category is paramount to category theory. Why else would the word "Category" make up over half of the letters in the theory's name? According to its perceived importance, this is the first particular we will be dealing with.

Categories are constructed out of *objects* and *arrows*. Arrows give rise to structure in categories by connecting objects. That is, an arrow connects a source object to a target object.

$$\texttt{source} \longrightarrow \texttt{target}$$

Objects in this sense are *just* objects, we know nothing about their internal structure, nor do we want to know about their internals. In category theory, we make no assumptions about the internal structure or workings of an object. Category theory describes objects purely in terms of their relations with other objects, therefore any categorical description of an object is data independent. Where a single description may apply to anything considered an object in a category, may it be sets, types, or graphs.

> A **category** consists of a collection of objects, a collection of arrows (often called *morphisms*) and a *composition* operator ∘.

The composition operator is central in this definition of a category. It allows us to compose arrows. So if you have an arrow `f` from object `A` to `B` and an arrow `g` from `B` to `C`, then we can combine these to form a new composite arrow `g` ∘ `f` from `A` to `C`. We read this as `g` 'after' `f`.

$$\begin{tikzcd}
\mathtt{A} \arrow[r, "\mathtt{f}"] \arrow[rr, bend right, "\mathtt{g}\circ\mathtt{f}"'] & \mathtt{B} \arrow[r, "\mathtt{g}"] & \mathtt{C}
\end{tikzcd}$$

Moreover, the operator limits what collections of objects and arrows are categories. Not just any collection of objects and arrows is a category. To be a bona fide category, the composition operator ought to satisfy some important properties.

1. Composition must be *associative*. This means that: for any arrows $\mathtt{f} : \mathtt{A} \to \mathtt{B}$, $\mathtt{g} : \mathtt{B} \to \mathtt{C}$, and $\mathtt{h} : \mathtt{C} \to \mathtt{D}$

$$\mathtt{h} \circ (\mathtt{g} \circ \mathtt{f}) = (\mathtt{h} \circ \mathtt{g}) \circ \mathtt{f}$$

   This implies that it does not matter where we place the parenthesis, the result will be the same.

2. For all objects $\mathtt{A}$ in the category, there is an *identity* arrow, which is a unit for composition. That means that this identity arrow composed with any arrow $\mathtt{f} : \mathtt{A} \to \mathtt{B}$, gives back this arrow. The identity arrow for object $\mathtt{A}$ is called $\mathrm{id}_{\mathtt{A}}$

$$\mathtt{f} \circ \mathtt{id}_{\mathtt{A}} = \mathtt{f} = \mathtt{id}_{\mathtt{B}} \circ \mathtt{f}$$

   Often the subscript of $\mathtt{id}_{\mathtt{A}}$ is dropped to form $\mathtt{id}$ when it is clear from the context which identity arrow is meant.

Remark 1 (Notation of Composition)
The usual order of composition – as used above – has an alternative. Normally, when we have a morphism $\mathtt{f} : \mathtt{A} \to \mathtt{B}$ and a morphism $\mathtt{g} : \mathtt{B} \to \mathtt{C}$, we write $\mathtt{g} \circ \mathtt{f}$. This notation is actually rather confusing. If we again look at the figure depicting composition:

$$\begin{tikzcd}
\mathtt{A} \arrow[r, "\mathtt{f}"] \arrow[rr, bend right, "\mathtt{g}\circ\mathtt{f}"'] & \mathtt{B} \arrow[r, "\mathtt{g}"] & \mathtt{C}
\end{tikzcd}$$

It would make a lot more sense to write $\mathtt{f} \circ \mathtt{g}$, following the order of the diagram.

$$\begin{tikzcd}
\mathtt{A} \arrow[r, "\mathtt{f}"] \arrow[rr, bend right, "\mathtt{f}\circ\mathtt{g}"'] & \mathtt{B} \arrow[r, "\mathtt{g}"] & \mathtt{C}
\end{tikzcd}$$

This alternative notation is called the *diagrammatic order*. When looking at papers and books on category theory it is good to first check which notation the author uses. The diagrammatic notation might look familiar to a programmer if we rewrite it as $\mathtt{f}\,;\,\mathtt{g}$. First execute $\mathtt{f}$ then $\mathtt{g}$. Some authors even prefer to write the object left of the morphism $\mathtt{(x)f}$ instead of $\mathtt{f(x)}$. This notation makes sense when considering the direction of the arrow $\mathtt{x} \xrightarrow{\mathtt{f}} \mathtt{y}$. Moreover, composition is naturally written as $\mathtt{(x)(f} \circ \mathtt{g) = ((x)f)g}$

This paper will adopt the usual way of writing composition – $\texttt{g} \circ \texttt{f}$ for $\texttt{A} \xrightarrow{\texttt{f}} \texttt{B} \xrightarrow{\texttt{g}} \texttt{C}$. Most functional programming languages follow this notation, and therefore we think using the notation programmers are most familiar with is preferable.

Remark 2 (Category Theory Abstracts)
Now categories are noticeably abstract, seeing we cold-shoulder the contents of objects when reasoning using category theory. Since it is so abstract, we have no direct applicable results. But we do want these, and to have applicable results from category theory, we still need to find constructs to which we can administer category theory. Yet if we want to verify that some structure/thing , we suspect, is a category, then we do have to look at the internals of this structure. We also need to do this if we want to see if our category has special properties.

In short, reasoning within category theory takes no assumption to the internals of the objects, but to actually verify that something is a category we need to look at the internals of that something. The suggestion of a category is just an abstract sleeve that fits many constructs.

Examples of Categories

To get some familiarity with categories, we will study a variety of example categories. After this section, we will start focussing on a single one.

Example 1 (Empty Category)
There is a category with zero objects and consequently, zero morphisms. This category is called the *empty category*, which by itself isn't much. Though it is not quite as pathetic as one might think, considering it has importance in the context of other categories, for instance, in the category of all categories. Similarly, an empty set is important in the context of other sets, so if an empty set makes sense, then why not an empty category?

Example 2 (Orders)
A *preorder* is a set P together with a transitive and reflexive relation $\leq$. A preorder gives rise to a category whose objects are the elements of P. Between each pair of objects $\texttt{p}$ and $\texttt{p}'$ with $\texttt{p} \leq \texttt{p}'$ there is a single arrow representing this fact; there is no arrow from $\texttt{p}$ to $\texttt{p}'$ when $\texttt{p} \not\leq \texttt{p}'$.

The reflexivity condition corresponds to the identity law ($\mathtt{a} \leq \mathtt{a}$). The transitivity of $\leq$ ensures the existence of composite arrows (if $\mathtt{a} \leq \mathtt{b}$ and $\mathtt{b} \leq \mathtt{c}$ then $\mathtt{a} \leq \mathtt{c}$) and since an element has only one way of being equal or less than another we have that the composition is associative.

A preorder is already enough to form a category, so that a *partial order* and a *total order* too give rise to a category.

Consider now these preordered sets as objects themselves, and we take *order-preserving functions* as arrows. Where an order-preserving function from $(P, \leq_P)$ to $(Q, \leq_Q)$ is a function $\mathtt{f} : P \to G$ such that if $\mathtt{p} \leq_P \mathtt{p}'$ then $\mathtt{f(p)} \leq_Q \mathtt{f(p')}$. These then form another category **Ord**.

The composition of two order-preserving functions $\mathtt{f} : P \to Q$ and $\mathtt{g} : Q \to R$ is a function $\mathtt{g} \circ \mathtt{f} : \mathtt{P} \to \mathtt{R}$. Furthermore, if $\mathtt{p} \leq_P \mathtt{p}'$ then, since f preserves P's ordering, $\mathtt{f(p)} \leq_Q \mathtt{f(p')}$; and since g preserves Q's ordering we have: $\mathtt{g(f(p))} \leq_R \mathtt{g(f(p'))}$. So $\mathtt{g} \circ \mathtt{f}$ is order preserving. Composition of order-preserving function is associative because each order-preserving function on preordered sets is just a function on sets and composition of functions on sets is associative. For each preorder, the identity function $\mathtt{id}_P$ preserves the ordering on P and satisfies the equations of the identity law.

Now **Ord** is a category and its objects – preorders – are categories too; ergo, **Ord** is an example of a category of categories. Then as well, the category of partial orders (**Poset**) and the category of total orders (a subset of **Poset**).

Example 3 (Monoids)
A *monoid* is an underlying set M equipped with a binary operation $\cdot$ from pairs of elements of M into M such that the binary operation is associative and has a *neutral/identity element* e such that $\mathtt{e} \cdot \mathtt{x} = \mathtt{x} = \mathtt{x} \cdot \mathtt{e}$. A monoid may be represented as a category with a single object. The elements of M are represented as arrows from this object to itself, the identity element e is represented as the identity arrow, and the operation $\cdot$ is represented as composition of arrows. Conversely, any category with a single object gives rise to a monoid. So a monoid is a single object category. In fact, the name monoid comes from Greek *mono*, which means single.

Again we find that the monoids form a category of categories if we take as objects monoids and as arrows *monoid homomorphisms*. A monoid homomorphism from $(M, \cdot_{\mathtt{M}}, \mathtt{e}_{\mathtt{M}})$ to $(O, \cdot_{\mathtt{O}}, \mathtt{e}_{\mathtt{O}})$ is a function $\mathtt{f} : M \to O$ such that $\mathtt{f(e_M)} = \mathtt{e_O}$ and $\mathtt{f(x} \cdot_{\mathtt{M}} \mathtt{y)} = \mathtt{f(x)} \cdot_{\mathtt{O}} \mathtt{f(y)}$. The verification that **Mon** is a category follows exactly the same steps as for **Poset**.

Representing categories

To understand categories better we can attempt to represent them visually. Objects can be depicted as stars or as circles, but to keep it simple, we shall represent them as points. A morphism may similarly be represented as myriad things, but we shall represent it as a simple arrow. To show off this graphic representation we give another example category:

Example 4 (The category **3**)
The category **3** is a category that has three objects: $\mathtt{A}$,$\mathtt{B}$ and $\mathtt{C}$, three identity arrows and three other arrows: $\mathtt{f} : \mathtt{A} \to \mathtt{B}$, $\mathtt{g} : \mathtt{B} \to \mathtt{C}$ and $\mathtt{h} : \mathtt{A} \to \mathtt{C}$. We can quickly check if **3** is indeed a category. To that end, we should establish if we can define composition, such that the two constraints are met. First, we observe that composition – in this example – can only be defined one way, since we can only compose two arrows if one arrow starts at the same object as the other ends. Solely $\mathtt{f}$ and $\mathtt{g}$ end and begin at the same object, with the result that only these can be composed; to form $\mathtt{g} \circ \mathtt{f} = \mathtt{h}$.

Now we check if this definition of composition fulfils the requirements:

1. Composition is associative: We need to show that for any arrows $\mathtt{f} : \mathtt{A} \to \mathtt{B}$, $\mathtt{g} : \mathtt{B} \to \mathtt{C}$, and $\mathtt{h} : \mathtt{C} \to \mathtt{D}$ the following holds.

   $$\mathtt{h} \circ (\mathtt{g} \circ \mathtt{f}) = (\mathtt{h} \circ \mathtt{g}) \circ \mathtt{f}$$

   When we try to find three arrows to compose, we find that we only have two. Therefore the condition is vacuously satisfied.

2. Identity: Every object has an identity arrow since we have 3 objects and 3 identity arrows.

So **3** is a category. When we display it graphically, it is portrayed as such:

$$\begin{tikzcd}
 & \mathtt{B} \arrow[loop above, "\mathtt{id_B}"] \arrow[dr, "\mathtt{g}"] & \\
\mathtt{A} \arrow[loop left, "\mathtt{id_A}"] \arrow[ur, "\mathtt{f}"] \arrow[rr, "\mathtt{h}"] & & \mathtt{C} \arrow[loop right, "\mathtt{id_C}"]
\end{tikzcd}$$

Diagrams

Statements about object and arrows can quickly become complicated, notably when they involve conditions like "`f ∘ a = b ∘ g`". Category theory offers the use of a graphical style of presentation to make such descriptions and the arguments involving them more manageable.

> A **diagram** in a category **C** is a collection of vertices and directed edges, labelled with objects and arrows of **C**. These labels are applied consistently, meaning if an edge in the diagram is labelled `f` and the arrow `f` has domain `A` and codomain `B`, then the endpoints of this edge must be labelled `A` and `B`.

An important distinction is made here: We must not confuse diagrams *in* categories with diagrams *of* categories.

We say that a diagram in a category **C** *commutes* if all paths in the diagram from every pair of vertices `A` and `Z` are equivalent. For example, saying that "the diagram

$$\begin{tikzcd}
A \arrow[r, "f'"] \arrow[d, "g'"] & B \arrow[d, "g"] \\
Y \arrow[r, "f"] & Z
\end{tikzcd}$$

commutes" is exactly the same as saying that `f ∘ g = g ∘ f`.

Composition and Programming (section adapted from [23, p. 7])

When we look at programming, we are solving non-trivial problems (because if they were trivial we wouldn't need the help of a computer). We solve the problems by decomposing them into smaller problems, then we write programs to solve these smaller problems and we compose these programs to solve the original problem. Once again we can compose various of these programs and out rolls another, bigger, program.

We do not program like this because it is the most efficient for a computer. No, we program like this because it is easier to keep track of what is happening if we chop-up problems into smaller pieces. We can only keep in mind so many things – I am hardly able to remember the grocery list – and can't keep track of the million things the complete program would need to keep track of. Hence by limiting the scope, our brains can still keep up.

> " As a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than to try to ignore them, for

> the latter vain effort will be punished by failure. ″
>
> (Edsger W. Dijkstra [7, p. 3])

For our simple brains to keep track of what is happening, we need to keep structure in our programs. Good code is structured code.

Half a century ago structural programming revolutionized programming because it made blocks of code composable [7]. Then object-oriented programming came along, which is all about composing objects. Now, recent developments have ideas from functional programming creeping into major languages. Lambda's and partial application in C++11 [14], polymorphic lambda's in C++14 [35], template lambda's in C++20 [8] and lambda's in Java 8 [11]. Functional programming is about composing functions, algebraic data structures and more interestingly it makes concurrency composable.

> ″ Category theory – rather than dealing with particulars – deals with structure. It deals with the kind of structure that makes programs composable. ″
>
> (Bartosz Milewski [23, p. x])

Essentially, Bartosz Milewski said category theory deals with structure, and therefore it is immensely suited to deal with programs. When reasoning purely wihtin category theory you cannot look into objects, these objects are black boxes. All you ever know about the object is how it relates to other objects. Similarly, in an OOP-language you only know about an object by its abstract interface, which stipulates the functions it contains (its morphisms, so how it relates to other objects). The moment you have to know the implementation of an object to compose it with other objects, you've lost the advantages of your programming paradigm[23].

[2] Types

In computer science and computer programming, we use types to classify data values. The type instructs the interpreter or compiler how this data is to be read and used. Additionally, it gives meaning to the data by stipulating the operations that work on this data. This interpretation of a type corresponds to a "value space and behaviour" as identified by Parnas et al. [25]. A value space means that only certain values are of that type, e.g. `1` is an integer but `'c'` is not. The observation that a type is a value space, leads us to our initial intuition about types, that they are sets of possible values.

The "behaviour" part of the classification means that there is a collection of distinct operations that work on those elements in the type's value space. For example, `sign` checks the sign of an integer, but calling `sign` on a string has little meaning. If we consider the values of a type as a set, then we can consider the operations on types as functions between these sets.

Briefly touching on sets: a set can be finite or infinite and so can a type be, respectively `Bool` and `String` are finite and infinite.

There are some subtleties that make this identification of types as sets tricky. For example polymorphic functions that involve circular definitions lead to Russell's paradox. And so does the fact that we can't have a set of all sets. We choose to ignore these problems. There are other ways to model types, that do not lead to paradoxes, but for this paper, this identification suffices.

Functional Programming

> " If we consider a program as a transformation which can be built up from "smaller" transformations. Then given a small library of basic (primitive) functions we may build our programs using functional constructors and without concern for any specific internal representation of that which is being transformed. "
>
> (Davit Pitt [28, p. 6])

We intend to use this functional programming view to introduce the basic concepts of category theory. We will develop a basic skeleton for a functional programming language. Then slowly introduce concepts from Category Theory and show how these basic categorical constructions express themselves in this functional programming language, by extending the language to incorporate the categorical constructs.

Our Functional Programming Language

We begin to show this fruitful view on types by considering a simple functional language (as adapted from [28] and [27]).

This "designed" language will use the same syntax as Hugs and Haskell [21].

We start from the basics; Functions map "values" into "values". The idea behind functional programming is then to use the functions to build programs (other functions) by somehow combining them. Before we can combine anything at all, we need to have something to combine. This work is provided by a set of initial primitive functions, these are the building-blocks for all our later programs. Initially, primitive functions are defined by application to the variables, for example:

```
id x = x
```

Which is read, 'id' applied to a constant x yields x. Another primitive function returns if a value is zero:

```
isZero x = if x == 0 then true else false
```

Which is read 'isZero' applied to x yields a boolean value that denotes if x was zero, if x is not a natural number the result is undefined. We now have two primitive functions which we can represent as arrows, working on the set of all possible values V:

$$\begin{tikzcd} \mathtt{V} \arrow[loop left, "\mathtt{id}"] \arrow[loop right, "\mathtt{isZero}"] \end{tikzcd}$$

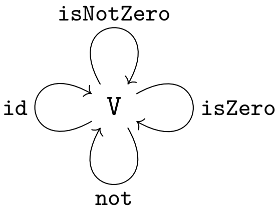
In constructing programs we are interested principally in functions (arrows) and how they can be combined. The most basic way of combining functions is composition, which is also defined in terms of application.

```
(g ∘ f) x = g(f(x))
```

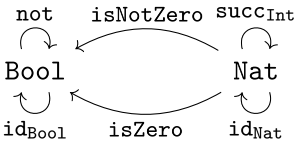
For example we can define:

```
isNotZero = not ∘ isZero
```

Now since those are all functions too, they appear as arrows in the figure:



In the shown example, we only discussed one type of constants, even though it is clearly nonsensical to apply every operation to all possible values: `isZero(x)` does not produce a sensible value unless `x` is a natural number. Hence, it would be clever to restrict functions to the values they enact on. If we introduce arithmetic functions, we should, logically, restrict these to numerical values. Logical functions such as `not` should apply to only boolean values. Thus we could introduce value spaces (types) of natural numbers (`Nat`) and booleans (`Bool`). Arrows would become characterized not by the constants that they are defined on but by their source (and target) types. This is also how we view arrows in category theory.

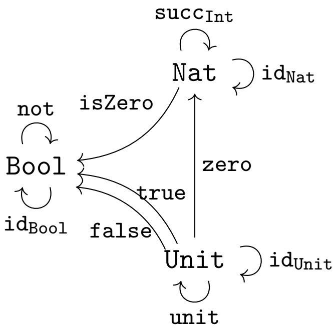

not
isNotZero
$\texttt{succ}_{\texttt{Int}}$
Bool
Nat
$\texttt{id}_{\texttt{Bool}}$
isZero
$\texttt{id}_{\texttt{Nat}}$

We finish the construction of this typed functional language, by removing the need to refer to values. We achieve this by introducing a type, `Unit`, such that functions from `Unit` to another type correspond precisely to the constants of that type.

For example, the functions from `Unit` to `Nat` are:

$\texttt{zero}$
$\texttt{succ}_{\texttt{Int}} \circ \texttt{zero}$
$\texttt{succ}_{\texttt{Int}} \circ (\texttt{succ}_{\texttt{Int}} \circ \texttt{zero})$
...

Corresponding exactly to all natural numbers. We end up with the following graphical representation of our language.


$\texttt{succ}_{\texttt{Int}}$
Nat
$\texttt{id}_{\texttt{Nat}}$
not
isZero
Bool
zero
true
$\texttt{id}_{\texttt{Bool}}$
false
Unit
$\texttt{id}_{\texttt{Unit}}$
unit

We expand the "created" language with the following components:

primitive types:

```
Int  -- integers
Real -- real numbers
Bool -- truth values
Unit -- a one-element type
```

built-in operations:

```
isZero    :: Int → Bool    -- test for zero
not       :: Bool → Bool   -- negation
succ_Int  :: Int → Int     -- integer successor
succ_Real :: Real → Real   -- real successor
toReal    :: Int → Real    -- integer to real conversion
```

constants:

```
zero  :: Int
true  :: Bool
false :: Bool
unit  :: Unit
```

Leaving out the identity arrows and composite arrows the category created from this small functional language looks as follows:

$$\begin{tikzcd}
& \mathtt{Int} \arrow[loop above, "\mathtt{succ}_{\mathtt{Int}}"] \arrow[dl, "\mathtt{isZero}"'] \arrow[dr, "\mathtt{toReal}"] & \\
\mathtt{Bool} \arrow[loop left, "\mathtt{not}"] & & \mathtt{Real} \arrow[loop right, "\mathtt{succ}_{\mathtt{real}}"] \\
& \mathtt{Unit} \arrow[uu, "\mathtt{zero}"'] \arrow[ul, shift left, "\mathtt{true}"] \arrow[ul, shift right, "\mathtt{false}"'] \arrow[loop below, "\mathtt{unit}"] &
\end{tikzcd}$$

Types and categories

If we take as our objects sets and functions as morphisms (arrows), then we end up with a category know as **Set**. As we have just seen, this could be useful as our intuitive understanding of types are sets of values. Ideally, we would just say that types are sets and functions are mathematical functions between sets. There is just one little problem: There is a difference between a mathematical function and a function in a programming language. A mathematical function just "knows" its solution, whilst a function in a programming language needs to be executed before we have a result. Now, this is not a problem if the answer can be calculated in a finite amount of steps. However, code can also be recursive, and therefore might never terminate and, hence, never have an answer. A naive solution would be to ban all non-terminating functions from a programming language, but this is impossible as that would mean we need to determine whether a function is terminating or not, the famous halting problem. To battle this problem, some really smart people came up with the idea to extend every type by a value, *bottom*, that corresponds to a non-terminating computation. Functions that may return bottom are called *partial* functions. Functions that return a valid value for every input are *total* functions.

Luckily for us, it is okay to ignore non-terminating functions and bottoms and treat types extended with bottom as **Set**. [6]

Composability of Types

Category theory is about composing, but not any two arrows are composable. For two arrows to compose the initial object of one arrow must be the same as the terminal object of the other. Similarly, in programming to compose functions, one function must accept the return type of the other function.

We can compose `zero :: Unit -> Int` and `toReal :: Int -> Real` to form `toReal∘zero :: Int -> Real`. However, we cannot compose `isZero :: Int -> Bool` and `toReal :: Int -> Real`. As `Int` $\neq$ `Real` and `Bool` $\neq$ `Int`.

$$\begin{tikzcd}
& \texttt{Int} \arrow[loop above, "\texttt{succ}_{\texttt{Int}}"] \arrow[dl, "\texttt{isZero}"'] \arrow[dr, "\texttt{toReal}"] & \\
\texttt{Bool} \arrow[loop left, "\texttt{not}"] & & \texttt{Real} \arrow[loop right, "\texttt{succ}_{\texttt{real}}"] \\
& \texttt{Unit} \arrow[loop below, "\texttt{unit}"] \arrow[uu, "\texttt{zero}"'] \arrow[ul, shift left, "\texttt{true}"] \arrow[ul, shift right, "\texttt{false}"'] \arrow[ur, dashed, "\texttt{toReal} \circ \texttt{zero}"'] &
\end{tikzcd}$$

Type Checking

With machine language, any combination of bytes will be accepted and run. Most of this will, of course, not do anything useful. With higher-level languages, we would appreciate if the compiler can detect lexical and grammatical errors. This filters out the nonsensical programs that one would be able to enter. Type checking proves yet another one of these barriers, as a type checker makes sure that once data is given a type, it does not display behaviour outside of what the type is expected to do.

Static vs. Dynamic *Dynamically* typed languages detect type mismatches at runtime, whether a *static* typed one would detect these at compile-time, eliminating lots of incorrect programs before they have the chance to run. Static typing might, however, reject programs that would run without runtime errors, and thus can result in incompleteness. Dynamic typing does have the benefit of accepting these programs that the static type checker would deem illegal.

Strong vs. Weak Often languages are referred to as *strong* or *weakly* typed. Weak typing roughly means that the language has a way to "bypass" its type system. It allows you to operate on data without considering its type. Strongly typed languages allow you to only use functions on data that are meant to be used on that type of data. However, be cautious when using these terms as there is no universal definition, and the properties that make a language strong or weak, are generally better expressed in different characteristics of the language.

Pure vs. Dirty In mathematics, a function is just a mapping of values to values. In programming, we can implement such a function. Give the function a value and it will calculate an output value. Such a function will – for the same input value – always return the same output value. The square function, for example, won't change depending on the phase of the moon or the arrangement of planets. Nor should this function have the additional effect of writing an entire bachelor thesis whilst calculating the square root of its input. A function that would do that cannot be easily modelled as a mathematical function. Functions that always give the same output given the same input, and have no additional side effects are called *pure functions*.

Haskell can be considered a strong statically-typed pure language.

Some exotic types

Void `Void` is the type that is not inhabited by any values. It is the empty set. Since it is not inhabited by any values we can not call any function which expects that type. Suppose we have:

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

What would happen if we try to call `absurd x`? `x` must be something in the value space of `Void`. But there is no such value and therefore we can never call this function.

This type does, however, have a use. It embodies the classical-logic law "from falsehood, anything follows".

Unit Unit is the type that is inhabited by only one value. It is isomorphic with every singleton (sets with only one value).

Here is a function of Unit that always returns the Integer 44.

```
f44 :: () -> Integer
f44    ()  = 44
```

We consider this a constant function as it always returns the same value. A function must be called with a value, so a constant function too. Conceptually, a constant function always takes the unit as input. This also makes sense in category theory as a morphism (function) goes from one object to another, it has a source and a target. From the Unit to the type of the constant. We do not need to mention this conceptual dummy value, because there is always only one instance of it, the single element in the unit type. So every function of Unit is equivalent to picking a single element from the target type. In fact, we could think of f44 as an alternative representation for the Integer 44. This gives us an example of how we could emit explicit mention of values in a set by talking about functions on them instead.

3 Universal Constructions

> " Mathematicians do not study objects, but the relations between objects; to them it is a matter of indifference if these objects are replaced by others, provided that the relations do not change. Matter does not engage their attention, they are interested in form alone " (Henri Poincaré [31, p. 20])

Poincaré tells us that mathematics is about relations between objects, add to this that category theory is used to formalize different areas of mathematics. Then it does not come as surprising that category theory is all about structure and relations. The only thing that matters to an object is how it relates to other objects. This also means that if we want to talk and reason about a single object, we first need to specify what its relations are to other objects. To do this, we use *universal constructions*. A universal construction gives us the "best" object satisfying a certain property. These properties of objects again only talk about their connections to other objects. Because of this we can image a property as being a pattern, made from objects and morphisms. A universal construction describes a collection or class of objects that all satisfy a certain property and picks out the most *universal*. Universal in this definition means that we are trying to pick the most general object from this collection.

Let us give an example:

Example 5 (Numbers)
Suppose we have a category, where the objects are non-negative integers $\mathbb{Z}^{\geq}$ $:= \{0, 1, 2, 3, \cdots\}$ and the morphisms specify "is smaller than or equal to" so $\leq$. This category is a poset and would look as follows.

$$\begin{tikzcd}
0 \arrow[r, "\leq"] \arrow[loop, "\leq"] & 1 \arrow[r, "\leq"] \arrow[loop, "\leq"] & 2 \arrow[r, "\leq"] \arrow[loop, "\leq"] & 3 \arrow[r] \arrow[loop, "\leq"] & \cdots
\end{tikzcd}$$

Clearly $1 \leq 3$, so one would have to imagine the composed arrows as well, but for brevity and visibility we shall leave those out.

We then want to specify a property, so we think of a pattern. The easiest pattern we can think of would be just a single object:

$$\bullet$$

As you can imagine, when we try to find all objects that satisfy this pattern, we get a lot. In fact, all objects in the category satisfy this pattern. That is a problem, as universal constructions exist to single out an object so that we can talk about the properties of that specific object. This is why we want to establish some kind of ranking on all objects that fit our pattern, we then pick the best object – as determined by our ranking – as being our universal construction.

We establish a ranking based on the morphisms between all candidates. We could say a candidate is "better" than another if there is an arrow going from it to the other. Then the "best" is the one that has one going to every other candidate.

$$\begin{tikzcd}
 & & \mathtt{A} \arrow[lld] \arrow[ld] \arrow[d] \arrow[rd] \arrow[rrd] & & \\
\mathtt{B} & \mathtt{C} & \mathtt{D} & \mathtt{E} & \mathtt{...}
\end{tikzcd}$$

What object is "best" in this ranking depends on the specific category or on the property. It could be the simplest, the smallest, or the largest. In this case every single object/number was a candidate. The best object satisfying the property was in fact the smallest number 0.

Still, we have a problem with this ranking, it is not precise enough. Suppose we had expanded the positive integers with another number 0′. That was also smaller than every other number (including 0), whilst at the same time having an arrow from 0 to 0′. So that we would have two smallest positive integers.

$$\begin{tikzcd}
0' \arrow[loop above, "\leq"] \arrow[dd, shift left, "\leq"] \arrow[dr, "\leq"] & & & & \\
 & 1 \arrow[loop above, "\leq"] \arrow[r, "\leq"] & 2 \arrow[loop above, "\leq"] \arrow[r, "\leq"] & 3 \arrow[loop above, "\leq"] \arrow[r] & \cdots \\
0 \arrow[loop below, "\leq"] \arrow[uu, shift left, "\leq"] \arrow[ur, "\leq"] & & & &
\end{tikzcd}$$

Which object `0` or `0`′ would we have to pick as our "best" object? The ranking chooses both, as both have an arrow to every other object. Evidently, our ranking doesn't pick an object *uniquely*. Now you might say: "But `0` and `0`′ are the same! They are completely interchangeable!". Sure, they *represent* the same, but they are *not* the same object. We could go back to Pointcarés quote:

> " ... it is a matter of indifference if these objects are replaced by others provided that the relations do not change. ... "
> (Henri Poincaré[31, p. 20])

We call objects like these – those that are completely interchangeable – *isomorphic*. From here on we say that it is fine if our ranking selects multiple objects, as long as they are all isomorphic. For our ranking to guarantee that all selected objects are isomorphic, we require there to be one and only one morphism to every other object.

When we use our pattern, together with our new ranking, we find `0` to fit our property *up to isomorphism*. And indeed `0` is our smallest positive integer.

Properties are like patterns and finding the universal construction, means establishing some kind of ranking among all occurrences – that satisfy that property – and picking the best fit. Another way to say this is that a universal construction is an object that satisfies a *universal property*.

Isomorphisms

We just introduced the term *isomorphism*, but the previous explanation, that the objects are completely interchangeable isn't entirely satisfactory.

Because what does it mean to be completely interchangeable in category theory? Intuitively, two isomorphic objects must look similar, they must have the same shape. In a way, the only thing that is different between the objects is their name.

We can try to better our understanding, by looking at a non-categoric setting. Suppose we were programming. We would have to have given a one-to-one correspondence between the constituents of the objects before we would call them isomorphic.

Take the two types `Bool`, having the values `true` and `false`, together with our new type `NotBool`, having the values `untrue` and `unfalse`. A bijection between both types (an invertible function) would show that they represent the same:

```
bool2notbool :: Bool -> NotBool
bool2notbool true  = unfalse
bool2notbool false = untrue
```

In category theory functions are morphisms. Then an isomorphism is an invertible morphism; or if you prefer, a morphism paired with its inverse. Then thus their composition is the identity function. Here `f` is a morphism and g is its inverse, together they form an isomorphism:

$$\begin{tikzcd}
\mathtt{A} \arrow[loop above, "\mathtt{id_A}"] \arrow[r, bend left, "\mathtt{f}"] & \mathtt{B} \arrow[loop above, "\mathtt{id_B}"] \arrow[l, bend left, "\mathtt{g}"]
\end{tikzcd}$$

```
f ∘ g = id
g ∘ f = id
```

We say that two objects, that are isomorphic to each other are *within an isomorphism* or that they are identical *up to isomorphism* [27] Likewise, if all objects that satisfy some property `P` are isomorphic, then we say that they are "unique up to isomorphism". In practice things are so often "unique up to isomorphism" that the qualification is often omitted.

> **Remark 3** (Not always a hit)
> A certain universal construction doesn't necessarily exist in a category. To illustrate that such a universal construction does not always have hits, we extend our category to include all integers as objects.
>
> $$\begin{tikzcd}
\cdots \arrow[r, "\leq"] & \mathtt{-2} \arrow[loop above, "\leq"] \arrow[r, "\leq"] & \mathtt{-1} \arrow[loop above, "\leq"] \arrow[r, "\leq"] & \mathtt{0} \arrow[loop above, "\leq"] \arrow[r, "\leq"] & \mathtt{1} \arrow[loop above, "\leq"] \arrow[r, "\leq"] & \mathtt{2} \arrow[loop above, "\leq"] \arrow[r] & \cdots
\end{tikzcd}$$
>
> As we can see, there is no object with a unique morphism to every object. So there is no number smaller than every other number.

We shall go trough a few examples of universal constructions, the first of which we have already seen:

Initial Object

The pattern and the ranking we used to find the best object in example 5, correspond with the universal construction of the initial object.

If we think of the arrow `A` → `B` as saying that `A` is "more initial" than `B`. Then *the* initial object is the object that is more initial than all other objects in the category. Therefore we can define the initial object as the object that has arrows going to all other objects in the category. In the example, we stated that the to be uniquely defined up to isomorphism we needed to add the additional requirement of having only one way of being more initial than another object.

Accordingly, we want the initial object to have only one arrow going to every other object. We define it as:

> The **initial object** is the object that has one and only one morphism going to any object in the category.

$$\begin{tikzcd}
& & \mathtt{A} \arrow[dll, "!"'] \arrow[dl, "!"'] \arrow[d, "!"] \arrow[dr, "!"] \arrow[drr, "!"] & & \\
\mathtt{B} & \mathtt{C} & \mathtt{D} & \mathtt{E} & \ldots
\end{tikzcd}$$

(Arrows from an initial object or to a terminal object are sometimes labelled "!" to highlight their uniqueness)

To show that the uniqueness of morphisms indeed ensures that the initial object is uniquely defined if it exists in a category we give a proof.

Example 6 (uniqueness of the initial object)
We give a proof by assuming the opposite.

Suppose the initial object was *not* unique up to isomorphism. Then we would have at least 2 distinct initial objects 0 and $0'$ such that they have a unique morphism to every object x in the category.

$$0 \to x$$

$$0' \to x$$

Every object x in the category includes these initial objects themselves. So that we can find morphism to make the following diagram.

$$\begin{tikzcd}
0 \arrow[loop above, "!id"] \arrow[r, bend left, "!f"] & 0' \arrow[loop above, "!id"] \arrow[l, bend left, "!g"]
\end{tikzcd}$$

So that we have $g \circ f :: 0 \to 0$, now we know from the initiality property of 0, that there is only one morphism from $0 \to 0$ and we know this to be the identity morphism. Consequently $g \circ f = id$. In much the same way, we find $f \circ g = id$. So that we find f and g to form an isomorphism.

In fact, we see that the initial object is *unique up to unique isomorphism* meaning that there is only one isomorphism relating these two isomorphic objects. Because there is only one morphism from $0 \to 0'$ and only one from $0' \to 0$.

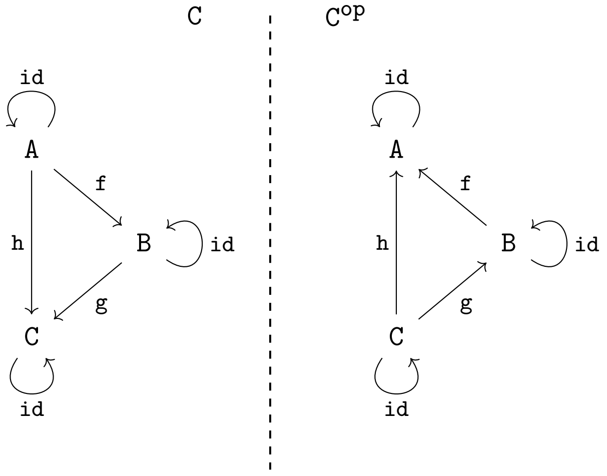


Figure 3.1: A category **C** and its dual

Duality

When dealing with the initial object we gave a ranking saying that `A` is more initial than `B` if there is an arrow `A` → `B`. Yet there is no reason we couldn't have reversed the ranking. Having `B` to be "more terminal" than `A` if there is an arrow `A` → `B`. Instead of finding the initial object, we could search for the *terminal object*.

Effectively, we just turned the arrows around. In fact, this is rather common in category theory. As it turns out, we can always reverse all morphisms in a category **C** in order to get the *opposite category* $\mathbf{C}^{op}$. This opposite category will then automatically satisfy all the requirement of a category, as long a were redefine composition.

```
f :: a -> b                       f^op :: b -> a
g :: b -> c                       g^op :: c -> b

h :: a -> c                       h^op :: c -> a
h = g ∘ f                         h^op = f^op ∘ g^op
```

The nice thing about this duality is that every time we come up with a construction, we can reverse all arrows and have the dual construction for free. These "free" constructions are often prefixed with "co". So we have products and coproducts, monads and comonads, limits and colimits, and so on. In short: for every "something" we have a co-"something".

We can make the concept of *dualizing* a category more rigorous, by defining an inductive dualisation function as seen in Figure 3.2. Though "reversing direction of all arrows" is all it does.

`dual(A)`	`= A`	for object `A`
`dual(x)`	`= x`	for morphism variable `x`
`dual(f : A- > B)`	`= f : B- > A`	(note the swap of `A` and `B`)
$\texttt{dual(g} \circ \texttt{f)}$	$= \texttt{dual(f)} \circ \texttt{dual(g)}$	(note the swap of `f` and `g`)
$\texttt{dual(id}_{\texttt{A}}\texttt{)}$	$= \texttt{id}_{\texttt{A}}$	

Figure 3.2: Dualising a category

Terminal Object

Now if we reverse the direction of the net-flow and think of `B` is “more terminal” than `A` if there is an arrow from `B` to `A`. Then we can similarly define:

> The **terminal object** is the object that has one and only one morphism going to it from every object in the category.

$$\begin{tikzcd}
\texttt{B} \arrow[drr, "!"'] & \texttt{C} \arrow[dr, "!"] & \texttt{D} \arrow[d, "!"] & \texttt{E} \arrow[dl, "!"'] & \cdots \arrow[dll, "!"] \\
 & & \texttt{A} & &
\end{tikzcd}$$

Initial Object in FPL

So let us try and add an initial object to the language, or find it in case it already exists. Yes, *an* initial object, not *the*, since there can be many representations for such an object in a category up to isomorphism.

So let us take a look at our programming language and check if it has a candidate for the initial object. Let first look at a familiar type `Bool`. First, we check if it has a function to every other type. At a first glance, it seems like it does, as for any type `A` with values, we can pick a single value `a` and create a function:

```
bool2A :: Bool -> A
bool2A    True  = a
bool2A    False = a
```

But this only works with types that actually have a value. `Void` does not have any values. Consequently, there is no function `bool2Void :: Bool -> Void`. And so `Bool` is definitely not an initial object. In fact, there is no function from any type to `Void`, except for the identity function specialized to `Void`: $\texttt{id}_{\texttt{Void}}$ `:: Void -> Void`. Now there is no function from any type to `Void`, so that any type that is not `Void` can clearly not be the initial object. So we can better hope `Void` is the initial object. Luckily `absurd :: Void -> a` is a function to every other type `a`.

You might now wonder why there exists no function to `Void`, but from `Void` there is always one. Kind of nonsense right? This actually has to do with the definition of a function in set-theory. As we are currently treating types as sets, functions between types closely resemble functions between sets. The set-theoretic definition of a function is as follows:

> A **function** is a relation between sets that associates to every element of a first set exactly one element of the second set.

We consider the existence of a function from `Void` to every other type `A`. This function might not be able to be executed, but that does not matter. It just needs to fit the definition of a function to be a morphism in **Set**. When we try to verify that there indeed is a function `Void -> A` for every `A`, we just need to look whether the definition holds. And the definition applies to `Void ->  A`, because the antecedent is not satisfied. There are no elements in the first set `Void` and therefore it is vacuously true that all elements (namely none) of the first set are associated with an element of the second set. We have no elements so we could not even associate them if we wanted to. Therefore the function `Void -> A` is set-theoretically a function. It is often referred to as the *empty function*.

Suppose we consider the possibility of a function `A -> Void`, then for an element `a` in `A`. We need to associate it to an element in `Void`, however, there are no elements in `Void`. So for every element in `A`, we cannot find an element in `Void` to associate it to. Clearly, this violates the definition of a function and so there is no function `A -> Void`.

Terminal Object in FPL

Again we attempt to add a terminal object to the language, or find it in case it already exists. Let us take another look at our programming language and check if it has a candidate for the terminal object. We know for sure that `Void` cannot be the terminal object as there were no functions `f :: A -> Void`.

Once more, we take a look at the familiar type `Bool`. This time, instead of finding a function *from* `Bool` to every other type, we need to check if there is a function from every other type *to* `Bool`. For `Void -> Bool` we have `absurd`. And for any type `A` with values we can construct a function that for every element in `A` will return `True`.

```
tautology :: A -> Bool
tautology    _  = True
```

As such we have a morphism to Bool from every object. But Bool is not a terminal object. As for every object we also have the function:

```
contradiction :: A -> Bool
contradiction    _  = False
```

This violates the condition that a terminal object has a *unique* morphism to it from every object and so Bool cannot be a terminal object. We find that a type with at least 2 values cannot be a terminal object. Then since Void is also eliminated it can only be a type holding a single value, a singleton. *The* singleton in **FPL** is Unit (or alternatively in Haskell syntax ()). Which has a unique function to it from every other type A.

```
unit :: A -> ()
unit    _  = ()
```

It is good to note that () is isomorphic to every other singleton. Say s is the single value of another singleton S. Then we can define an isomorphism between () and S. So that every singleton is a terminal object up to isomorphism.

```
unit :: S -> ()
unit    s  = ()

unit' :: () -> S
unit'    ()  = s
```

Products

We look at another interesting construction in set theory and check if we can find a universal construction that captures its properties.

From set theory, we know that the cartesian product of two sets is a set of pairs of the elements from the initial sets. However this gives mention of the elements in the set, and that cannot be imitated in category theory. Ultimately it does not tell us enough to make a categorical construction for the product. We need an "arrow-theoretic" way to describe a pair. With that in mind, we can look at the relations a product has to its constituent sets. We know that there are two functions, the projections, from the cartesian product set to its constituent sets.

We find the following construction if try to capture this pattern as objects and morphisms, where there is an object C from which we can construct objects A and B, using two projections p and q going from C to A and B respectively.

$$\begin{tikzcd}
 & \mathtt{C} \arrow[dl, "\mathtt{p}"'] \arrow[dr, "\mathtt{q}"] & \\
\mathtt{A} & & \mathtt{B}
\end{tikzcd}$$

Now all C's that fit this pattern will be a possible candidate for a product. There may be lots of them, and therefore we need to establish a way to get the best "fit" for this pattern.

Say C and $\mathtt{C}'$ are both candidates for a product. Then if C is "better" than C′, then C would be the more universal one. This means that the projections of $\mathtt{C}'$, namely, $\mathtt{p}'$ and $\mathtt{q}'$ can be factorized by p and q (the projections of C) and some other function $\mathtt{m} :: \mathtt{C}' \rightarrow \mathtt{C}$.

$$\mathtt{p}' = \mathtt{p} \circ \mathtt{m}$$

$$\mathtt{q}' = \mathtt{q} \circ \mathtt{m}$$

$$\begin{tikzcd}
 & \mathtt{C}' \arrow[dl, "\mathtt{p}'"'] \arrow[d, "\mathtt{m}"] \arrow[dr, "\mathtt{q}'"] & \\
\mathtt{A} & \mathtt{C} \arrow[l, "\mathtt{p}"] \arrow[r, "\mathtt{q}"'] & \mathtt{B}
\end{tikzcd}$$

We take at all such candidates, and pick the best one. We give this object the special name $\mathtt{A} \times \mathtt{B}$. This is *the* product (up to isomorphism) in its respective category:

> **Remark 4** (Dashed arrow)
> A dashed arrow in a diagram means that the morphism is "presumed to exits" if the rest of the diagram is filled in properly. In this case, if C is indeed 'better' than $\mathtt{C}'$ then there *must exist* $\mathtt{C}' \overset{\mathtt{m}}{\dashrightarrow} \mathtt{C}$.

A **product** of two object A and B is the object $\mathtt{A} \times \mathtt{B}$ equipped with two projections p, q such that for *any other* objects $\mathtt{C}'$ equipped with two projections $\mathtt{p}'$, $\mathtt{q}'$ there is a unique morphism $\langle \mathtt{p}', \mathtt{q}' \rangle : \mathtt{C} \rightarrow \mathtt{A} \times \mathtt{B}$ that factorizes those projections:

$$\mathtt{p} \circ \langle \mathtt{p}', \mathtt{q}' \rangle = \mathtt{p}'$$

$$\mathtt{q} \circ \langle \mathtt{p}', \mathtt{q}' \rangle = \mathtt{q}'$$

$$\begin{tikzcd}
 & \mathtt{C}' \arrow[dl, "\mathtt{p}'"'] \arrow[d, dashed, "!\langle \mathtt{p}', \mathtt{q}' \rangle"] \arrow[dr, "\mathtt{q}'"] & \\
\mathtt{A} & \mathtt{A} \times \mathtt{B} \arrow[l, "\mathtt{p}"] \arrow[r, "\mathtt{q}"'] & \mathtt{B}
\end{tikzcd}$$

> **Remark 5** (The projections are important)
> Even though we call $\mathtt{A} \times \mathtt{B}$ the product, the projection arrows are just as an important part of the definition. Strictly speaking, we should define the product as the tuple $(\mathtt{A} \times \mathtt{B}, \mathtt{p}, \mathtt{q})$.

We can also define arrows *between* product objects in terms of projection arrows:

> If $A \times C$ and $B \times D$ are product objects, then for every pair of arrows $f : A \to B$ and $g : C \to D$, the **product map** $f \times g : A \times C \to B \times D$ is the arrow $\langle f \circ p, g \circ q \rangle$.

Product in FPL

Now the current type system of our language is not powerful enough to express a product for every pair of types. For example, there is no fitting product `Int`×`Int`.

Yet we would like to have this capability and so we introduce a likely familiar construct to the type language: a pair `(,)`, together with the two projections `fst` and `snd`.

```
fst :: (A,B) -> A                snd :: (A,B)-> B
fst (a,_) = a                    snd (_,b) = b
```

Then given any type `C` with two projections `p :: C -> A` and `q :: C -> B`, there is a unique function from `C` to the product `(A,B)` that factorizes them. In fact, it just combines `p` and `q` into a pair.

```
m :: C -> (A,B)
m x = (p x, q x)
```

$$\begin{tikzcd}
 & C \arrow[dl, "p"'] \arrow[d, dashed, "!m"] \arrow[dr, "q"] & \\
A & (A,B) \arrow[l, "fst"] \arrow[r, "snd"'] & B
\end{tikzcd}$$

So that `m` properly factorizes `p` and `q`:

```
p :: C -> A                      q :: C -> A
p = fst . m                      q = snd . m
```

It appears that `(A,B)` is the product of `A` and `B` in **FPL**. To then define the product map in **FPL** we would need a function, that for `f :: A -> B` and `g :: C -> D` can give a new function `(A,C) -> (B,D)`. This demand is satisfied by:

```
mapPair :: (A -> B) -> (C -> D) -> (A,C) -> (B,D)
mapPair f g (a,c) = (f a, g c)
```

> **Remark 6** (A × B vs. B × A)
> Of course `( (A,B), fst, snd)` is not the only representative of the set of tuples `(X, p, q)` that is optimal in this sense. For example, the type `((B,A), snd, fst)` is just as good. But `(A,B)` and `(B,A)` are the same up to an isomorphism by `swap (a,b) = (b,a)`. So, categorically, we think of them as essentially the same.

Coproducts

We already defined the product, and now by dualising our previous definition, we get a definition of the coproduct for free.

> A **coproduct** of two objects A and B is the object $A + B$ equipped with two injections i, j such that for *any other* objects C' equipped with two injections i' and j' there is a unique morphism $[i', j']$ from $A + B$ to C' that factorizes those injections.

$$i' = [i', j'] \circ i$$

$$j' = [i', j'] \circ j$$

$$\begin{tikzcd}
A \arrow[r, "i"] \arrow[dr, "i'"'] & A + B \arrow[d, dashed, "{![i',j']}"] & B \arrow[l, "j"'] \arrow[dl, "j'"] \\
& C' &
\end{tikzcd}$$

To gain a bit of intuition for the coproduct, we can take a look at set theory, in the category of sets a coproduct is a disjoint union of two sets. Say the disjoint union of set A and B. Then the elements in the coproduct of A and B are either in set A or they are in set B. You can imagine this as the "sum" of two sets.

In the same manner as with product we can also define arrows *between* coproduct objects:

> If $A + C$ and $B + D$ are coproduct objects, then for every pair of arrows $f : A \to B$ and $g : C \to D$, the **coproduct map** $f + g : A + C \to B + D$ is the arrow $[i \circ f, j \circ g]$.

Coproduct in FPL

Again the current category does not have a coproduct for every two objects.

To solve this problem, We introduce yet another construct to the type language: a tagged union.

```
data A+B = (0, A) | (1, B)
```

A tagged union is a data structure used to hold a value that could take on several different, but fixed, types. A value of type `A+B` is a value of type `A` or (|) of type `B`, but not both at the same time. The arbitrary chosen numbers `0` and `1` are used to 'tag' the values of the two summands so they can be distinguished. Closely related to our type `A+B` are the injections:

```
inj1 x = (0, x)
inj2 y = (1, y)
```

`A+B` is parameterized by two types, `A` and `B` and has two injections, the constructors `inj1` that takes a value of type `A` and `inj2` that takes a value of type `B`. We see that the types in this case are tagged with either 0 or 1.

$$\begin{tikzcd}
\texttt{A} \arrow[dr, "\texttt{inj1}"'] & & \texttt{B} \arrow[dl, "\texttt{inj2}"] \\
& \texttt{A+B} &
\end{tikzcd}$$

Then given any type `C` with two injections `i :: A -> C` and `j :: B -> C`, there is a unique `m` from the coproduct `A+B` to `C` that factorizes them. Actually it just "removes" the tag and applies the proper injection.

```
m :: A+B -> C
m (0, a) = i a
m (1, b) = j b
```

$$\begin{tikzcd}
\texttt{A} \arrow[r, "\texttt{inj1}"] \arrow[dr, "\texttt{i}"'] & \texttt{A+B} \arrow[d, dashed, "\texttt{!m}"] & \texttt{B} \arrow[l, "\texttt{inj2}"'] \arrow[dl, "\texttt{j}"] \\
& \texttt{C} &
\end{tikzcd}$$

So that `m` properly factorizes `p` and `q`:

```
i :: A -> C                     j :: B -> C
i = m . inj1                    j = m . inj2
```

Remark 7 (tagged union in Haskell)
In Haskell, the role of the tagged union is covered by the `Either` type.

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

`Either a b` is parameterized by two types, `a` and `b` and has two injections, the constructors `Left` that takes a value of type `A` and `Right` that takes a value of type `B`. We see that the types in this case are tagged with either `Left` or `Right`.

$$\begin{tikzcd}
\texttt{A} \arrow[dr, "\texttt{Left}"] & & \texttt{B} \arrow[dl, "\texttt{Right}"'] \\
& \texttt{Either A B} &
\end{tikzcd}$$

[4] Algebraic Data types

In the defined language we currently have only a fixed amount of primitive types, `Void`, `Unit`, `Bool`, `Int` and `Real`, together will all their products and coproducts. Though, admittedly, it would be nice if we could define our own types. Fortunately, the constructs we already have in the language provide us with a way to do this. We can combine the current types to form new types using `A × B` and `A + B`, and we can do this indefinitely: The created type `Int × Int`, can again be combined with `Bool`, to form `(Int × Int) + Bool` and so forth. These created types are called *algebraic data types*. In programming, an algebraic data type is a kind of composite type, a type formed by combining other types. As it turns out, a lot of data structures in programming can be built using just two constructs; using just products and coproducts.

> An **algebraic data type** is a possibly recursive sum type of product types.

Type Constructors

To make use of these algebraic data types we introduce *type constructors*. As the name suggests, these construct types. A type constructor is an n-ary operator working on types. A type constructor takes zero or more types and returns another type.

We lend the syntax from Haskell:

```
data Typename [typevar₁ ··· typevarₖ] = constr₁ | ··· | constrₙ
-- where type variables distinct
```

Essentially, new data types are defined by listing all possible proper ways to construct their values. We use *data constructors* to construct the data. We write data constructors using a sans serif font, whereas Haskell would capitalize the first letter.

Looking back on the existing primitive types, these can be considered built using nullary type constructors, for instance:

```
data Bool = true | false
```

Product

The current implementation of a product in the language is a simple 2-tuple `(A,B)`. I will now introduce a more general way of defining product types in the language, we can use a data constructor with multiple arguments as an alternative definition for a product:

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

Here, `Pair a b` is the name of the type parameterized by the two types `a` and `b` and pair is the name of the data constructor. We use this data constructor to construct the data, we can create a `Pair` value by passing two values to pair:

```
aPoint :: Pair Int Int
aPoint = pair 1 2
```

It is easily seen that `aPoint` is isomorphic to the tuple `(1,2)`.

Remark 8 (Identical type and data constructors names)
Since namespaces for type and data constructors are separate in Haskell, you will often see the same name used for both, as in:

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

With a bit of imagination and a bit of squinting the eyes, we could even view the built-in Haskell pair as a variation of this definition, as in:

```
data (,) a b = (,) a b
```

In fact, we can actually construct tuple elements in Haskell using `(,)`. Similarly, we can use `(,,)` to construct triples and `(,,,)` for quadruples and so on.

Example 7 (Specific Named Products)
Instead of using a generic pair or tuples, you can also define specific named product types, as in:

```
data Person = person String Int
```

The advantage of this style of declaration is that you may define many types that have the same content but different meaning and functionality, and which cannot be substituted for each other.

```
data Address = address String Int
```

person "Jane Doe" 22 and address "Abbey Road" 6 – now having different types – are not interchangeable.

Simple 2-tuples are not the only tuples that can make use of this alternative style. No, we can analogously define any n-tuple. For illustration, here is a quadruple:

```
Quadruple a b c d = quad a b c d
```

Sum (Coproduct)

The implementation of the sum type in our language is:

```
data a + b = (0, a) | (1, b)
```

Or written a bit differently:

```
data a + b = inj1 a | inj2 b
```

We see that the injections are *constructors* for the data type a + b. Conveniently, we have the tags correspond to data constructors' names.

```
data Sum a b = left a | right b
```

So that something of type a is tagged with left and something of type b is tagged with right. The tag allows us to use *pattern matching*. It sort of allows us to manipulate these datatypes to "deconstruct" the data structure into its components:

```
whatIsIt :: Sum Int Bool -> String
whatIsIt (left _) = "is an Int!"
whatIsIt (right _) = "is a Bool!"
```

This type just holds two possible types of values. We can expand that number and create a sum type with any amount of data constructors.

```
IsOneOfThese a b c d = first a | second b | third c | fourth d
```

Example 8 (Enumerations)
First thing, the simplest of these sum types are just enumerations.

```
data Suit = clubs | diamonds | hearts | spades
```

Example 9 (Maybe)
Sum types also offer a great way the express that a value could be absent. The way this is commonly done in Haskell is using the `Maybe` type:

```
data Maybe a = nothing | just a
```

We can see this data type as a *summation* of to two other data types. One data type that holds only the value nothing (so a singleton) and another data type that simply encapsulates a value `a`.

```
data Nothing = nothing

data JustAValue a = just a
```

So that we can combine these types into `Nothing + JustAValue a` to form an alternative representation for `Maybe a`.

Another way of defining `Maybe`, would have been `Maybe a = Unit + a`.

Recursive data types

Data definitions can be *recursive* allowing to describe data structures of varying size. For instance, below are defined natural numbers and (polymorphic) lists as recursive data types:

Example 10 (Natural Numbers)
We first look at a simple inductively (= recursively) defined data type, that of the Peano naturals.

```
data Nat = zero | succ Nat
```

Note that `Nat` occurs both on the left and the right of the datatype definition. This is why it is called an inductively defined or recursive data type.

Example 11 (Lists)
The `List` data type has two constructors. One matches the representation for an the empty, nil, and the other – cons – presents a single value

> followed by the remainder of the list.
>
> ```
> data List a = nil | cons a (List a)
> ```

Algebra of Types

Taken separately, product and sum types can be used to define a variety of useful data structures, but the real strength comes from combining the two. Combing them we find that Set together with the product and coproduct form a semi-ring. Now natural numbers also form a semi-ring under multiplication and summation. We see some interesting results if we translate our semi-ring Set to the one of natural numbers. Looking at this similarity it becomes clear why types are called "algebraic".

We take a step back and inspect and look at a few of our datatypes again.

Void

Among our data types, `Void` is the smallest because it is empty. `Void` has exactly 0 values, consequently it also has 0 constructors. We noted that `Void` isn't very useful at a first glance, but it has some interesting properties.

$$\mathtt{A} + \mathtt{Void} \cong \mathtt{A}$$

$$\mathtt{A} \times \mathtt{Void} \cong \mathtt{Void}$$

Does this ring a bell? These properties seem awfully familiar to the algebraic properties of 0.

$$a + 0 = a$$

$$a * 0 = a$$

We introduce 0 as an alternative notation to `Void`. As a matter of fact, we are the odd ones here in regard to notation considering the terminal object is most often denoted by 0 in category theory.

Unit

The next datatype we find is `Unit` with exactly 1 element `()`, do you find a trend here?

We recall that `Unit` is isomorphic to every other singleton, and so a singleton and unit are "identical" as far as we care. This means that the the value or the inhabitant for the `Unit` type become irrelevant, we could replace this value with any other value without losing expressive power.

From this observation we refer to the class of all singleton types by 1. Consequently,

$$\mathtt{A} \times 1 \cong \mathtt{A}$$

We can read this as follows: If the second components of a product cannot change, then it is useless and can be ignored. We find the functions `fst` and its inverse `(id, c)` to provide an isomorphism where `c` is our inhabitant for datatype 1. In much the same way:

$$1 \times \mathtt{A} \cong \mathtt{A}$$

Which has a striking familiarly to the algebraic identity:

$$1 * a = a = a * 1$$

Finally, what can we say about $1 + \mathtt{A}$? Observing this type, we see it is either a single value or something useful of type `A`. This correspond exactly to the `Maybe` type which is either a single value nothing or an encapsulation of a useful type just `A`.

Bool

Following from previous observations on `Void` and `Unit`, these have a class of isomorphic types with a name indicating the amount of inhabitants. `Bool` has with true and false exactly 2 inhabitants. Likewise, we will use the symbol 2 to represent the class of all objects containing exactly two distinct elements.

We write:

$$1 + 1 \cong 2$$

This follows from the fact that for anytype $\mathtt{B} = \{b_1, b_2\}$ – so a type with 2 elements – we can trivially make an isomorphism to $1 + 1$. Obviously `Bool` is in this class, which enables us to use symbol 2 in places where `Bool` is expected. Clearly:

$$2 \times \mathtt{A} \cong \mathtt{A} + \mathtt{A}$$

Just think of the example $\mathtt{Bool} \times \mathtt{Nat}$, where bool indicates the sign of the natural number, then it allows us represent a number that is either negative or positive. Doubling the amount of possible values.

```
data Integer = negative Nat | positive Nat
```

Translation table

We find the following translation table for types to their algebraic representation:

Numbers	Types
0	Void
1	Unit
2	Bool
a + b	A + B
$a * b$	A × B

Counting Inhabitants

The first direct use of this algebra is knowing how many inhabitants a type has. A + B has as many as A and B, combined. Again, the number of inhabitants looks exactly the same as the algebraic form, A + B. A × B has an inhabitant for each combination of a's and b's, that is a total of A × B.

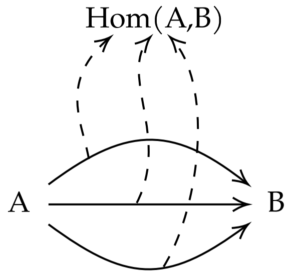
Similarly, the logical *and* and *or* also from a semi-ring, and it too can be mapped into category theory. Which is the basis for curry-howard isomorphism.

Numbers	Types
false	Void
true	()
a || b	Either a b = Left a | Left b
a && b	(a,b)

5 Function Types

So far we have tried to dodge the subject: So far we have only discussed functions as being morphisms, but functions have types too, in Haskell at least. As you are aware: types are objects in our category thus a function – which has a type – must have an associated object.

However, function types are different from other types. They are sets of morphisms from one type to some other type. These sets between objects are called *hom-sets*.



Now this is a *set* of different morphisms and so it is also an object in the category **Set**. We notice that in the hom-set has a corresponding object in the category of sets. This phenomenon is not present in every category, so we make a distinction:

External and internal Hom-sets

When a hom-set is external to a category. Then it is called an external hom-set. This means that the hom-set is not part of the category. In some categories there are ways to represent the hom-set as object within the category, a so-called hom-object. These hom-sets – those that can be represented *within* the category – are called *internal* hom-sets. The reason that there is a distinction between external and internal hom-sets, is that not every category has such a hom-object. For example, it can be shown that there is no such construction in **Mon** and evidently, such a hom-object must exist in the category of sets (it is an internal hom-*set*). So that we have, for any two sets `Hom(A, B) = {f : A → B}` is itself a set.

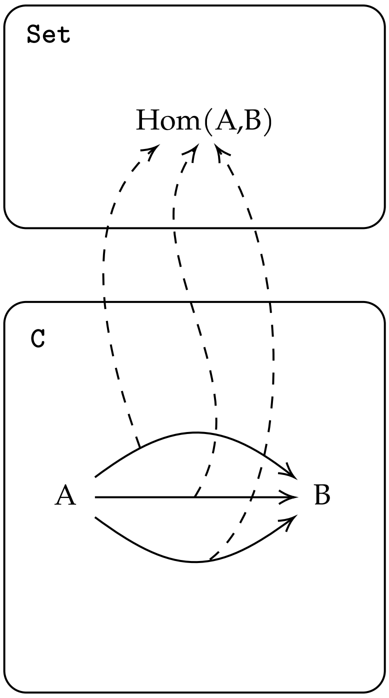


Figure 5.1: Depiction of a Hom-object *external* to the category **C**

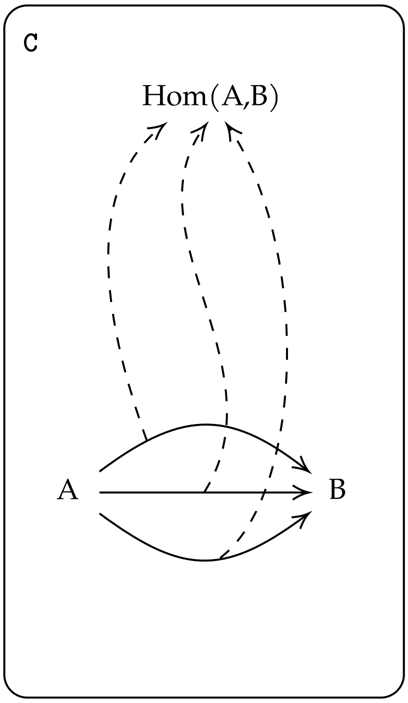


Figure 5.2: Depiction of a Hom-object *internal* to the category **C**

Universal Construction

Let us try to create a function type – that is an internal hom-object representing the morphisms between two types. We abstract away from the category of sets, this way the construction will automatically work for other categories. We will however, take our hints from set theory.

As usual, we want an arrow-theoretic way of describing a function. To do this we start with our initial understanding of a function on sets: A function is something that is *applied* to something else or *evaluated* when given an argument. This evaluation/application describes a relationship between different objects in a category. We can try to capture this relation – of the function and its argument – as a universal construction, similar to the product and coproduct.

With this in mind let us try to construct a pattern. Given a candidate for a function type, lets call it `Z`, an argument type `A` and a result type `B`, evaluation maps an element of `f` of `Z` and an element `a` of type `A` to some value `b` of type `B`: `f(a) = b`. This evaluation must work for every pair `(f, a)`. We say that this evaluation is a morphism from $\mathtt{Z} \times \mathtt{A} \to \mathtt{B}$. The following pattern emerges from this requirement:

$$\begin{tikzcd}
\mathtt{Z} & \mathtt{Z} \times \mathtt{A} \arrow[l] \arrow[r, "\mathtt{g}"] \arrow[d] & \mathtt{B} \\
 & \mathtt{A} &
\end{tikzcd}$$

We can now try and look at all possible objects that satisfy this pattern for `Z`. You can look at this construction as a sort of query looking for the right objects. Exactly those together with an evaluation function g that satisfies this pattern.

Now this pattern is not specific enough to single out the function type. This construction has a lot of hits, and we want to establish the best fit for this pattern. If we define the better function type to be the most general one, then we need to show that there exists a function type candidate that factorizes every other candidate's evaluation function uniquely. We call the best such candidate `A⇒B`. If we have another candidate `Z`. With its own evaluation morphism `eval′`. Then there must be a unique morphism `h` from $\mathtt{Z} \to \mathtt{A{\Rightarrow}B}$. That factorizes `eval′` trough `eval`: As illustrated in the following diagram:

$$\begin{tikzcd}
\mathtt{Z} \arrow[d, dashed, "\mathtt{!h}"'] & \mathtt{Z} \times \mathtt{A} \arrow[l] \arrow[d, dashed, "\mathtt{!h{\times}id_A}"'] \arrow[dr, "\mathtt{eval}'"] & \\
\mathtt{A{\Rightarrow}B} & \mathtt{A{\Rightarrow}B} \times \mathtt{A} \arrow[l] \arrow[r, "\mathtt{eval}"'] \arrow[d] & \mathtt{B} \\
& \mathtt{A} &
\end{tikzcd}$$

Recall that $\mathtt{h} \times \mathtt{id_A}$ denotes a product mapping as defined in chapter 3. So that $\mathtt{eval}' = \mathtt{eval} \circ (\mathtt{h} \times \mathtt{id_A})$. Formally:

> A **function object** from $\mathtt{A}$ to $\mathtt{B}$ is an object $\mathtt{A{\Rightarrow}B}$ together with the morphism $\mathtt{eval} :: (\mathtt{A} \Rightarrow \mathtt{B}) \times \mathtt{A} \rightarrow \mathtt{B}$.
>
> So that for any other objects $\mathtt{Z}$ with a morphism $\mathtt{eval}' :: (\mathtt{Z} \times \mathtt{A}) \rightarrow \mathtt{B}$. There is a unique morphism $\mathtt{h} :: \mathtt{Z} \rightarrow (\mathtt{A} \Rightarrow \mathtt{B})$ that factors $\mathtt{eval}'$ through $\mathtt{eval}$: $\mathtt{eval}' = \mathtt{eval} \circ (\mathtt{h} \times \mathtt{id})$.

We trim the above diagram to fit only the important parts in our definition:

$$\begin{tikzcd}
\mathtt{Z} \times \mathtt{A} \arrow[d, dashed, "\mathtt{!h{\times}id_A}"'] \arrow[dr, "\mathtt{eval}'"] & \\
\mathtt{A{\Rightarrow}B} \times \mathtt{A} \arrow[r, "\mathtt{eval}"'] & \mathtt{B}
\end{tikzcd}$$

Currying

If we once again take a look at the candidates $\mathtt{Z}$ for the function object, we can see something interesting: All candidates for the function type have some g that maps a product of two values to a single value.

$$\mathtt{g} :: (\mathtt{Z} \times \mathtt{A}) \rightarrow \mathtt{B}$$

At the same time, from the universal property of the function type, we know that there is a unique morphism $\mathtt{h}$ that maps $\mathtt{Z}$ to a function object $\mathtt{A} \rightarrow \mathtt{B}$.

$$\mathtt{h} :: \mathtt{Z} \rightarrow (\mathtt{A} \Rightarrow \mathtt{B})$$

When we think of these morphisms as functions in **Set**, our first g is the closest representation we can get to a function taking two arguments. Besides $\mathtt{h}$ is a function that takes one variable of type $\mathtt{Z}$ and maps it to a function from $\mathtt{A}$ to $\mathtt{B}$.

Then the universal construction establishes a correspondence between functions of two variables and functions of one variable returning a function. This correspondence is called *currying*, and h is called the *curried* version of g. Part of what makes this interesting is that this correspondence is a bijection.

⇒ If we have g, we know there is a unique h from the universal construction.

⇐ Conversely, if we have h, then we can create g in the following manner:

```
g = eval . (h × id)
```

This correspondence is captured by the standard Haskell functions curry and uncurry.

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f a b = f (a, b)

uncurry ::  (a -> b -> c) -> ((a,b) -> c)
uncurry f (a,b) = f a b
```

Functions as data

Using currying, we showed a relation between functions and products. In addition, we needed it to construct the universal construction. This relation actually goes deeper than that. This is best seen when you consider functions between finite types – types that have a finite number of values, like Bool, Char or even Int. Such functions, at least in principle, can be fully memoized or turned into data structures, where one can find the answer to the function, just by looking it up. This function-data correlation is the essence of the equivalence between functions, which are morphisms and function types, which are objects.

For instance we can take a function and :: Bool -> Bool -> Bool, corresponding to the logical AND and turn this into a lookup table with two keys, corresponding to the first and second argument.

a	b	result
True	True	True
True	False	False
False	True	False
False	False	False

Now this is just one such function of type `Bool -> Bool -> Bool`. For all functions `Bool -> Bool -> Bool`, we could make such a lookup table. A function is different from another function if just one result cell is different. For every row, we may change the result cell to any value of the result type. Whenever we change one such result type we have a new unique lookup table.

In this case we have $2^2 = 4$ rows and the result type `Bool` has two possible values `True` and `False`. We conclude that we have a total of $2^{2^2} = 16$ possible different tables, and so 16 possible different inhabitants of the type `Bool -> Bool -> Bool`.

Haskell's `Char` encodes the Unicode characters, which supports a total of $1,114,112$ characters. Of course, Unicode has not mapped all of these possible values. But for simplicities sake, we shall assume so. As a consequence, `Char` – for our intents and purposes – contains $1,114,112$ values.

Then for a function `Bool -> Char`, we have 2 rows, and for each row $1,114,112$ result values, hence, a total of $1,114,112^2$ possible values. We find that the amount of inhabitants to a function type is exponential, where the argument type is the exponent. A function `A -> B` has B^A inhabitants.

This is why in mathematical literature, the function object, or the internal hom-objects between two objects A and B, is often called the *exponential* and denoted by B^A.

Higher-order functions

Now that we have functions as objects, the category includes morphisms from these function objects to other objects $B^A \to C$, In addition we have morphism to these function objects $C \to B^A$. Recalling that a morphism is a function, we infer that we now have functions with a function as argument and functions with a functions as result types, otherwise known as *higher-order functions*.

> A function is called a **higher-order function** if it does one of the following:
>
> 1 takes one or more functions as arguments.
>
> 2 returns a function as its result.
>
> All other functions are *first-order functions*.

For example `curry` is a higher-order function. But what we notice it that the existence of exponentials gives rise to higher-order-functions.

Exponentials and Algebraic Data Types

The connection between function types and exponentials can be broadened when we look at the algebraic data types. The interpretation of function types as exponentials seamlessly incorporates into our algebra of algebraic datatypes.

All basic identities from high-school algebra remain true. Categorical interpretations of initial and final objects, coproducts, products, and exponentials appear consistent with simple equations relating the numbers zero and one, sums, products and exponentials – learned in high-school. We lack the tools to prove these, but fortunately, the work has been done for us in [13]. In the paper, they use the Yoneda lemma and adjunctions, two rather noteworthy ideas in category theory.

In all following identities, we replace 0 with initial objects, 1 with the terminal object and equality with isomorphism. The exponential is the internal hom-object.

Zeroth power

$$a^0 = 1$$

Translated:

```
Void -> a ≅ ()
```

The exponential a^0 represents the set of morphisms from the initial objects to an arbitrary object a. By the universal property of the initial object, we know that there is only one such arrow. Then the hom-set C(0,a) has only 1 element, a singleton. But we know that every singleton is isomorphic to the terminal object in *Set*.

We saw the function in question before: It's `absurd`.

Powers of one

$$1^a = 1$$

Translated:

```
a -> () ≅ ()
```

The intuition is similar to the previous result. From the definition of the terminal object. There is only one morphism from 1 to a. And so this hom-set is isomorphic to the terminal object.

The only function from any type `a` to `()` is the function `unit`.

First Power

$$a^1 = a$$

Translated:

```
() -> a ≅ a
```

This identity restates the finding that we can omit explicit mention of values in a type, by talking about functions on them instead. The set of such morphisms is isomorphic to the object itself.

Exponentials of sums

$$a^{b+c} = a^{b} \times a^{c}$$

Translated:

```
Either b c -> a ≅ (b -> a, c -> a)
```

The result of this statement is quite practical. It states that, categorically, the exponential from a coproduct of two objects is isomorphic to a product of two exponentials.

In our language, this means that a function from a sum of two types is equivalent to a pair of functions from individual types. We encounter this rather often when programming in Haskell. It is just the case distinction we make when defining a function over a sum type.

```
data Pet = cat | dog

f :: Pet -> String
f cat = "This is a Cat"
f dog = "This is a Dog"
```

Instead of writing one function definition with a case statement, we split the function up in two functions, one dealing with cats, the other with dogs.

Exponentials of Exponentials

$$(a^{b})^{c} = a^{b\times c}$$

Translated:

```
c -> (b -> a) ≅ (b,c) -> a
```

This is just a restatement of currying purely in terms of exponential objects.

```
curry ::  ( (b,c) -> a) -> (c -> (b -> a))
curry f a b = f (a,b)

uncurry :: (c -> (b -> a))-> ((b,c) -> a)
uncurry f (a,b) = f a b
```

Exponentials over products

$$(\mathtt{a} \times \mathtt{b})^{\mathtt{c}} = \mathtt{a}^{\mathtt{c}} \times \mathtt{b}^{\mathtt{c}}$$

Translated:

```
c -> (a,b) ≅ (c -> a, c -> b)
```

A function that returns a pair, is equivalent to a pair of two separate functions returning one element of the pair.

6 Functors

We have seen that many mathematical domains can be formulated as categories. Since categories themselves constitute a mathematical domain, it makes sense to ask whether there is a category of categories. In fact, there is: Its objects are categories, and its arrows are certain structure-preserving maps between categories, called *functors*. Something to note about these maps: A Category consists of not just objects – it consists of objects and morphisms that connect these objects. So a mapping between categories must not only map objects, but also map morphisms:

> Given two categories, **C** and **D**, a **mapping** `m` between the two categories is a operation such that:
>
> 1 `m` maps *all* objects in **C** to *an* object in **D**. If `A` is an object in **C**, then its image in **D** is `mA`.
>
> 2 `m` maps *all* morphisms `f : A` → `B` in **C** to morphisms in **D**. If `f : A` → `B` is a morphism in **C**, then its image in **D** is `mf : mA` → `mB`.

Remark 9 (mapping vs. function)
The words mapping and function are often used interchangeably. But occasionally one is chosen for emphasis. For instance in this paper, we will use the term "function" to refer specifically to a map of sets with no structure, and use the term "map" when dealing with any type of algebraic structure. Hence, the difference is that a map can also relate two objects that are not necessarily sets. Alternatively, a mapping may be described by a morphism, which generalizes the idea of a function. This rationale also fits the programmatic definition of a function, in the sense that a program function acts on sets of values, our types.

Now that we know what a mapping is, we get to introduce the functor. A functor is not too difficult to understand:

> A **functor** is just a mapping between categories. Such that it preserves connections between objects:
> Given two categories, **C** and **D**, the functor F preserves identity morphisms and composition of morphisms:
>
> **Identity:**
> Let $\mathtt{A}$ be an object in **C** then:
>
> $$\mathtt{F}\ \mathtt{id_a} = \mathtt{id_{Fa}}$$
>
> "The mapping applied to the identity morphism in **C** is the identity morphism in **D**"
>
> $$\begin{tikzcd} \mathtt{A} \arrow[loop above, "\mathtt{id_a}"] \\ \mathtt{FA} \arrow[loop above, "\mathtt{Fid_a=id_{Fa}}"] \end{tikzcd}$$
>
> **Composition:**
> For composable arrows $\mathtt{f} : \mathtt{A} \to \mathtt{B}$ and $\mathtt{g} : \mathtt{B} \to \mathtt{C}$ in **C**
>
> $$\mathtt{F}\ (\mathtt{g} \circ \mathtt{f}) = \mathtt{F}\ \mathtt{g} \circ \mathtt{F}\ \mathtt{f}$$
>
> "It does not matter if we first compose $\mathtt{f}$ and $\mathtt{g}$ and take the mapping then, or first take the mappings of $\mathtt{f}$ and $\mathtt{g}$ and compose those"
>
> $$\begin{tikzcd} \mathtt{A} \arrow[rr, "\mathtt{h}"] \arrow[dr, "\mathtt{f}"'] & & \mathtt{C} \\ & \mathtt{B} \arrow[ur, "\mathtt{g}"'] & \end{tikzcd}$$
>
> $$\begin{tikzcd} \mathtt{FA} \arrow[rr, "\mathtt{Fh}"] \arrow[dr, "\mathtt{Ff}"'] & & \mathtt{FC} \\ & \mathtt{FB} \arrow[ur, "\mathtt{Fg}"'] & \end{tikzcd}$$

Remark 10 (No tears in the fabric)
The notion that a functor preserves connections is important, frequently it is said that a functor is *structure preserving*. Bartosz Milewski gives us a fun way we can try to visualize this:

> " If you picture a category as a collection of objects held together by a network of morphisms, a functor is not allowed to introduce any tears into this fabric. It may smash objects together, it may glue multiple morphisms into one, but it may never break things apart. "
>
> (Bartosz Milewski [23, p. 72])

Remark 11 (function word)
The word *functor* comes from the term *function word* in linguistics [17,

> p. 30]. These are words that have little lexical meaning and express grammatical relationships among other words within a sentence. They signal structural relationships that words have to one another (the syntax of the sentence). So a function word for categories – a functor – expresses structural relationships between categories.

Functor in Programming

Whilst programming we are dealing with our category of types and functions. Anything we define while programming will have to do with this category. And so a functor regarding to programming only deals with this category. We can talk about functors that map the category of types into itself – such self-mapping functors are called *endofunctors*. Then an endofunctor in the category of types maps types to types. Examples of such mappings are the definitions of types that are parameterized by other types. `Maybe` is such a type constructor parameterized by an other type.

```
data Maybe a = nothing | just a
```

It is important to note that `Maybe` is not a type. It is a type constructor. To turn it into a type we have to first provide a type, say `Int`, then `Maybe Int` is a type. `Maybe` without arguments represents a function on types. Since it takes `Int` to `Maybe Int`.

Now can we turn `Maybe` into a functor? (yes, we can). The type constructor `Maybe` already deals with the mapping of objects, so the only thing left is defining the part of the functor that maps the morphisms between types. So for every function `f :: a -> b` we would like to produce a function `Maybe a -> Maybe b`. In Haskell, we implement this mapping as a higher-order function

```
f(unction)map :: (a -> b) -> (Maybe a -> Maybe b)
```

We hand it a function and it "lifts" this function to be used on `Maybe` types. "lifting" is used as a metaphor for taking something (in this case, a function) from one context to another.

```
instance Functor Maybe where
    -- We can not do anything with nothing
    fmap _ nothing = nothing
    -- The obvious mapping: We tear of the label and apply
    -- f to the "value within"
    fmap f (just a) = just (f a)
```

We should always make sure that the implementation of `fmap` for a function preserves identity and composition. These are called *the functor laws*, but they simply ensure the preservation of the structure of the category.

Identity:
We have to look at two distinct cases, when `Maybe a` is nothing or is `just a`:

```
fmap id nothing = nothing = id nothing

fmap id (just a) = just (id a) = just a = id (just a)
```

Composition:
Again we look at both cases:

```
fmap (f . g) nothing
= nothing
= fmap g nothing
= fmap f (fmap g nothing )
= (fmap f . fmap g) nothing

fmap (f . g) (just a)
= just ((f . g) a)
= just f(g(a))
= fmap f (just (g(a))
= fmap f (fmap g (just a))
= (fmap f . fmap g) (just a)
```

Example 12 (The List functor)
If we want to gain a better intuition as to the role of functors in programming, then we will need to look at more examples. To be fair, category theory is, likely, best explained by examples.

Possibly, you have heard that functors can be seen as if they are some sort of containers, this analogy provides a pretty good for trivial functors. Anyway, we shall first start with a generic container, the List. Generic containers are parameterized by the type of the elements they store, and as we have seen: any type that is parameterized by another type is a candidate for a functor.

```
data List a = nil | cons a (List a)
```

We have a mapping from any type `A` to the list type `List A`. The mapping we are talking about is the type constructor `List`. To show that this mapping is a functor we have to define the lifting of functions: given a function `f :: a -> b` define a function `List a -> List b`

```
fmap :: (a -> b) -> (List a -> List b)
```

It obvious what this function should do. It should turn every element of type a to type b using the function f. Then fmap f becomes our new function List a -> List b. Now there is a special case, what should happen if we encounter a list containing no values? Well, if there are no values then there is nothing we can apply f to, so that we retain an empty list.

```
instance Functor List where
    fmap _ nil = nil
    fmap f (cons a as) = cons (f a) (fmap f as)
```

We note that List defines the object part of a functor from the types in Haskell to the type in Haskell (so an endofunctor). And that fmap defines the arrow part of the functor.

Then let l = [s_1, s_2, ..., s_n], then fmap f maps f over the elements of l:

List f l = fmap f l = [f s_1, f s_2, ..., f s_n]

Example 13 (Identity functor)
The identity functor on a category **C** is the functor I_C that takes every **C**-object and every **C**-arrow to itself.

In Haskell its definition can be given as:

```
newtype Identity a = identity a

instance Functor Identity where
    fmap f (identity x) = identity (f x)
```

You can treat it as a container that does not do anything. It is simply a wrapper around a single value, and that is it. The identity functor really does nothing. Identity a and a are in fact isomorphic, which is trivial to see. Not very surprisingly fmap f then is just a disguised f, it strips off the Identity container, applies f, and slaps the container back on again.

Example 14 (The Diagonal functor)
The diagonal functor `∆ :: C -> C` takes each **C**-object `A` to the object `(A, A)` in the product category **C** × **C**. The reason for its name is rather obvious when you look at the following table:

	a	b	c	d	⋯
a	**(a, a)**	(a, b)	(a, c)	(a, d)	⋯
b	(b, a)	**(b, b)**	(b, c)	(b, d)	⋯
c	(c, a)	(c, b)	**(c, c)**	(c, d)	⋯
d	(d, a)	(d, b)	(d, c)	**(d, d)**	⋯
⋮	⋮	⋮	⋮	⋮	⋱

In Haskell we can try to construe the Diagonal functor like this:

```
data ∆ a = ∆ a a

instance Functor ∆ where
    fmap f (∆ a a)  = ∆ (f a) (f a)
```

We quickly check the functor laws and find that the proofs are easily formulated:
Identity:

```
fmap id (∆ a a)
= ∆ (id a) (id a)
= ∆ a a = id ∆ a a
```

Composition:

```
fmap (f . g) (∆ a a)
= ∆ ((f . g) a) ((f . g) a)
= ∆ (f (g a)) (f (g a))
= fmap f (∆ (g a) (g a))
= fmap f (fmap g (∆ a a))
= (fmap f . fmap g) (∆ a a)
```

Example 15 (The Reader functor)
Consider a mapping of type `A` to the type of a function returning `A`. This is a mapping from one type to another and thus is a candidate for a functor. Normaly we would use the type constructor to map the

objects part of the functor. But what actually is the type constructor for a function? If we give the type of a function from `A` to `B` we denote it as `A -> B`. So that must mean that `->` is simply an infix type constructor.

Just as with regular functions we can partially apply type constructors. Resulting in the partially applied form `(R ->)`. The definition of `fmap` applied to this case gives the following signature:

```
fmap :: (A -> B) -> (R -> A) -> (R -> B)
```

`fmap` takes a function from `A -> B` and a function from `R -> A` and returns us a function `R -> B`. Giving it a bit of thought we find that simple composition does the trick.

```
instance Functor (R ->) where
    fmap f g = f . g
```

And in point-free style:

```
instance Functor (R ->) where
    fmap = (.)
```

We verify the identity law:

```
fmap id h
= (.) id h
= id . h
= h
```

and the composition law:

```
fmap (f . g) h
= (.) (f . g) h
= (f . g) . h
= f . (g . h)
= fmap f (g . h)
= fmap f (fmap g h)
= (fmap f . fmap g) h
```

Functor Composition

Knowing that a functor is a mapping between categories, it is not hard to imagine that functors compose. A composition of two functors is just the composition of their respective mappings.

So that the object part of the new mapping becomes the composition of the respective mappings on objects. Correspondingly, the new mapping on morphisms is just the composition of the original mappings when acting on morphisms:

> Given functors `F :: A -> B` and `G :: B -> C` the composite functor `G∘F` maps:
>
> 1 each **A**-object `A` to the **C**-object:
> `G(F A)`.
>
> 2 each **A**-arrow `f : A` $\rightarrow$ `A`$'$ to the **C**-arrow:
> `G(F f) :: G(F A)` $\rightarrow$ `G(F A')`.

It is easy to check that this composition operation is associative and that the identity functor is the identity for the composition of functors.

So the composition of two functors is a functor whose `fmap` is the composition of the corresponding `fmap`'s. Now interestingly this composition is associative, and there is an identity functor (it maps every object to itself and maps every morphism to itself). So functors behave as morphisms in some category. This category is the category of categories, where objects are categories and morphisms are functors.

However, the category of categories would have to include itself, and this is impossible. Therefore, there is a distinction between categories. *small* and *big* categories. A category is small whenever objects within the category form a set, as opposed to something larger than a set, such as a proper class. Mind that a set can be infinite. Even an infinite uncountable set is considered small.

7 Natural Transformations

After having introduced the functor in the previous section, we once again apply the oldest trick in the book: We check if the created structures form a category.

We consider the functors as objects in a category. Then what could the morphisms between these functors be?

Mappings on Functors

Since these mappings go from one functor to the other, we at least know that the morphism must preserve the objects functorial nature. We consider two functors F and G between categories **C** and **D**. Then an object a in **C** is mapped to both Fa and Ga, using respectively the functors F and G.

$$\begin{tikzcd}
 & \mathtt{a} \arrow[dl, "\mathtt{F}"'] \arrow[dr, "\mathtt{G}"] & \\
\mathtt{Fa} & & \mathtt{Ga}
\end{tikzcd}$$

The mapping we are looking for should map every a in **C** to some morphism in **D** from Fa to Ga. We can visualize this as “sliding” the image of F onto the image of G. Now both Fa and Ga are in the same category **D**. We can not just randomly add a new morphism in this category **D**, and thus the morphism Fa to Ga must be an already existing (naturally existing) morphism in the category **D**. This morphism picks for every a in **C**, a single morphism from Fa to Ga in **D**. This morphism is called the *component* of α at a, or $\alpha_{\mathtt{a}}$.

$$\begin{tikzcd}
 & \mathtt{a} \arrow[dl, "\mathtt{F}"'] \arrow[dr, "\mathtt{G}"] & \\
\mathtt{Fa} \arrow[rr, "\alpha_{\mathtt{a}}"] & & \mathtt{Ga}
\end{tikzcd}$$

Keep in mind that a is an object in **C** while $\alpha_{\mathtt{a}}$ is a morphism in **D**. Does such a morphism in **D** not exists for some a in **C**, then a mapping as defined above does not exist. Furthermore, note that α is a collection of naturally existing morphisms in **D** that all go from F **C** to G **C**, one morphisms for every object in **C**. So that α becomes a mapping between the objects of the two functors.

Now we also need to define how this mapping works on morphisms in **C**. As it turns out we do not really have a choice when defining this mapping. To elaborate: If there is a `f :: a` $\rightarrow$ `b` in **C**. Then since `F` and `G` are functors, it is mapped to `Ff :: Fa` $\rightarrow$ `Fb` and `Gf :: Ga` $\rightarrow$ `Gb`.

$$\begin{tikzcd}
 & \mathtt{a} \arrow[dl, "\mathtt{F}"'] \arrow[dr, "\mathtt{G}"] \arrow[dd, "\mathtt{f}"] & \\
\mathtt{Fa} \arrow[dd, "\mathtt{Ff}"] & & \mathtt{Ga} \arrow[dd, "\mathtt{Gf}"] \\
 & \mathtt{b} \arrow[dl, "\mathtt{F}"'] \arrow[dr, "\mathtt{G}"] & \\
\mathtt{Fb} & & \mathtt{Gb}
\end{tikzcd}$$

α then completes this diagram by providing two additional morphisms: $\alpha_{\mathtt{a}}$ `:: Fa` $\rightarrow$ `Ga` and $\alpha_{\mathtt{b}}$ `:: Fb` $\rightarrow$ `Gb`. Where $\alpha_{\mathtt{a}}$ and $\alpha_{\mathtt{b}}$ take the endpoints of the `F`-image of `f` to the endpoints of the `G`-image of `f`.

$$\begin{tikzcd}
 & \mathtt{a} \arrow[dl, "\mathtt{F}"'] \arrow[dr, "\mathtt{G}"] \arrow[dd, "\mathtt{f}"] & \\
\mathtt{Fa} \arrow[rr, "\alpha_{\mathtt{a}}"] \arrow[dd, "\mathtt{Ff}"] & & \mathtt{Ga} \arrow[dd, "\mathtt{Gf}"] \\
 & \mathtt{b} \arrow[dl, "\mathtt{F}"'] \arrow[dr, "\mathtt{G}"] & \\
\mathtt{Fb} \arrow[rr, "\alpha_{\mathtt{b}}"] & & \mathtt{Gb}
\end{tikzcd}$$

If we then keep only the objects and morphisms in **D**, we are left with the following diagram:

$$\begin{tikzcd}
\mathtt{Fa} \arrow[r, "\alpha_{\mathtt{a}}"] \arrow[d, "\mathtt{Ff}"] & \mathtt{Ga} \arrow[d, "\mathtt{Gf}"] \\
\mathtt{Fb} \arrow[r, "\alpha_{\mathtt{b}}"] & \mathtt{Gb}
\end{tikzcd}$$

If we look closely at the diagram, we see that it contains all objects and morphisms that are related to `f` in **D**. So the candidate for the mapping of `f` must be somewhere in this diagram too. Taking a look at these candidates, two of the morphisms are images of `f` under `F` and `G`. One involves only `F`, the other only `G`, so those are no good. The other two are components of the transformation α, $\alpha_{\texttt{a}}$ and $\alpha_{\texttt{b}}$, but these are similarly bad options. The only options that are left are the composite morphisms $\texttt{Gf} \circ \alpha_{\texttt{a}}$ and $\alpha_{\texttt{b}} \circ \texttt{Ff}$. These both give ways of getting from `Fa` to `Gb`, however since we would like our mapping to be structure preserving, these should be equal. We must impose that for any `f` the *naturality condition* holds:

$$\texttt{Gf} \circ \alpha_{\texttt{a}} = \alpha_{\texttt{b}} \circ \texttt{Ff}$$

> The mapping α as we just described is a **natural transformation** from **C** to **D**.

Since we were left with little choice regarding the mapping of a morphisms in **C**, we do not have to define it, as long as it satisfies the naturality condition.

Natural Transformation in Programming

Example 16 (Reverse)
Let `reverse` be the function that reverses lists – that is

$$\texttt{reverse}_{\texttt{S}} \texttt{ :: List S -> List S}$$

takes any list with elements in `S` to its reverse. For example,

$$\texttt{reverse}_{\texttt{Int}} \texttt{ [1,2,3] = [3,2,1]}$$

`reverse` truly does not care about the specific elements in the list. We could replace them with anything and the substitution would be the same.

$$\texttt{reverse}_{\texttt{Int}} \texttt{ [4,5,6] = [6,5,4]}$$

We can apply any mapping whatsoever to the individual elements of the arguments to `reverse`, even one that changes their types.
Our previous example then was:

$$\texttt{map (+3) (reverse}_{\texttt{Int}} \texttt{ [1,2,3]) = [1+3, 2+3, 3+3] = [4,5,6]}$$

And more generally, suppose that `f : Int -> B`, then:

$$\begin{aligned}
&\mathtt{map\ f\ (reverse_{Int}\ [4,5,6])}\\
&\ \mathtt{=}\\
&\mathtt{[f(4),\ f(5),\ f(6)]}\\
&\ \mathtt{=}\\
&\mathtt{reverse_{B}\ (map\ f\ [4,5,6])}
\end{aligned}$$

or even, if $\mathtt{f\ :\ A\ {->}\ B}$:

$$\begin{aligned}
&\mathtt{map\ f} \circ \mathtt{reverse_{A}} = \mathtt{reverse_{B}} \circ \mathtt{map\ f}\\
&=\\
&\mathtt{fmap_{List}\ f} \circ \mathtt{reverse_{A}} = \mathtt{reverse_{B}} \circ \mathtt{fmap_{List}\ f}
\end{aligned}$$

But this precisely states the naturality condition, that means that `reverse` is a natural transformation.

Example 17 (Eval)
In categories with exponentiation (those things we use to model morphisms in the category **Set**) it turns out the the evaluation function forms a natural transformation.

We give the construction in **Set**:

For a fixed set A, the map taking B to $\mathtt{B}^{\mathtt{A}} \times \mathtt{A}$ can be extended to a functor $\mathtt{F_A} : \mathit{Set} \to \mathit{Set}$ as follows:

$$\begin{aligned}
&\mathtt{F_A}\ B = \mathtt{B}^{\mathtt{A}} \times \mathtt{A}\\
&\mathtt{F_A}\ \mathtt{(f\ :\ B} \to \mathtt{C)} = \mathtt{(f} \circ \mathtt{-)} \times \mathtt{id_A}
\end{aligned}$$

The fact that $\mathtt{eval\ :\ F_A\ {->}\ I}_{\mathit{Set}}$ is a natural transformation follows from the commutativity of the diagram

$$\begin{tikzcd}
\mathtt{F_A(B)} \arrow[r, "\mathtt{eval_{AB}}"] \arrow[d, "\mathtt{F_A(f)}"'] & \mathtt{I}_{\mathit{Set}}\mathtt{(B)} \arrow[d, "\mathtt{I}_{\mathit{Set}}\mathtt{(f)}"] \\
\mathtt{F_A(C)} \arrow[r, "\mathtt{eval_{AC}}"] & \mathtt{I}_{\mathit{Set}}\mathtt{(C)}
\end{tikzcd}$$

We know that $\mathtt{F_A(C)} = \mathtt{C}^{\mathtt{A}} \times \mathtt{A}$, and $\mathtt{F_A(B)} = \mathtt{B}^{\mathtt{A}} \times \mathtt{A}$, also we know $\mathtt{I}_{\mathit{Set}}\mathtt{(B)} = \mathtt{B}$ and $\mathtt{I}_{\mathit{Set}}\mathtt{(C)} = \mathtt{C}$, as $\mathtt{I}_{\mathit{Set}}$ is the identity functor. Rewriting our square we get:

$$\begin{tikzcd}
\mathtt{B}^{\mathtt{A}} \times \mathtt{A} \arrow[r, "\mathtt{eval_{AB}}"] \arrow[d, "\mathtt{(f \circ -) \times id_A}"'] & \mathtt{B} \arrow[d, "\mathtt{f}"] \\
\mathtt{C}^{\mathtt{A}} \times \mathtt{A} \arrow[r, "\mathtt{eval_{AC}}"] & \mathtt{C}
\end{tikzcd}$$

To show that the above diagram commutes we use the universal property of `eval`.

$$\begin{tikzcd}
\mathtt{B}^{\mathtt{A}} \times \mathtt{A} \arrow[d, dashed, "(\mathtt{f}\circ\mathtt{eval}_{\mathtt{AB}})\times\mathtt{id}_{\mathtt{A}}"'] \arrow[dr, "\mathtt{f}\circ\mathtt{eval}_{\mathtt{AB}}"] & \\
\mathtt{C}^{\mathtt{A}} \times \mathtt{A} \arrow[r, "\mathtt{eval}_{\mathtt{AC}}"'] & \mathtt{B}
\end{tikzcd}$$

So that our above square commutes for - = $\mathtt{eval}_{\mathtt{AB}}$.

Example 18 (Length)
Interesting cases of natural transformations appear when one of the functors is the `Const` functor. For instance, length can be though of as a natural transformation form the list functor to the `Const` functor. In practice length is defined as:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

This implementation effectively hides the fact that it's a natural transformation. However, `Int` is isomorphic to `Const Int a`, with the following isomorphism:

```
toConst :: c -> Const c a
toconst c = Const c

fromConst :: Const c a -> c
fromConst (Const x) = x
```

Making use of this isomorphism we can define `length` as:

```
length :: [a] -> Const Int a
length [] = Const 0
length (x:xs) = Const (1 + fromConst (length xs))
```

Here fromConst is used to peel off the Const constructor. Now `length` ignores the type of the elements of the list, and the `Const` functor ignores the second type argument. It then is not hard to imagine that changing these ignored types using `fmap`, will not change the behaviour of length. And so it satisfies the naturality condition.

$$\texttt{fmap}_{\texttt{ConstInt}} \circ \texttt{length} = \texttt{length} \circ \texttt{fmap}_{\texttt{List}}$$

Polymorphic function

The examples above all had something in common. They all are *polymorphic functions*, as it turns out every polymorphic function in Haskell is a natural transformation; we will touch on this in chapter 8. So all examples above were a cheap shot.

Polymorphic functions don't touch the elements, they don't use them, they don't view them, the simply leave them as is. Meaning that polymorphic function only change structure of a certain type and not any values within.

Conclusion Part I

This concludes the first part of the paper. We've learned the basic vocabulary of category theory.

We have seen categories consisting of objects and morphisms between these objects. Then we took a look at these categories and established that they form an category themselves, with functors as morphisms. Once more we looked at our new constructions and have shown that the functors form a category with natural transformation as morphisms.

We have taken a look at the basic definitions of category and what programming construct correspond to them. We found that exponentiation gives rise to higher order functions. Functors reveal themselves in a programming language as type constructors, and natural transformations correspond to the polymorphic functions.

We have looked at simple universal constructions - the product and the coproduct - and seen how they give rise to a large variety of algebraic datatypes.

Part II

Uses

This is the beginning of part 2. We shall showcase three different applications of category theory within computer science. Starting with free theorems, followed by recursion schemes and we end with a section on monads.

8 Free Theorems

The first direct result from category theory we will discuss is a magic trick:

> “ Write down the definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function’s definition. I will tell you a theorem that the function satisfies. ”
>
> (Philip Wadler [37, p. 1])

Wadler, in his paper, gives us a way to generate “free” theorems from the type signature of polymorphic functions where any term of such type must adhere to the theorem. It is “free” because we do not need to take a look at the function itself, but only at its signature.

Say that `r` is a function of type:

```
r :: [a] -> [a]
```

From just this, as we shall see, it is possible to conclude that `r` satisfies the following theorem: for all types `A` and `A'` and every total function `a : A -> A'` we have

$$\mathtt{map\ a} \circ \mathtt{r}_{\mathtt{A}} = \mathtt{r}_{\mathtt{A}'} \circ \mathtt{map\ a}$$

Here $\mathtt{r}_{\mathtt{A}}$ is the instance of `r` at type `A`. Intuitively, we can explain the result `r` must work on list of `X` for *any* type `X`. If `r` works on any list no matter the type of its elements, we know it does not touch the elements. All `r` can do with the list is to rearrange the elements in the list. And so applying `a` to each element of a list and then rearranging yields the same result as rearranging and then applying `a` to each element. Which is what the equation above states.

Examples of such rearranging functions `r` are `reverse`, `tail` and `take 3`. Take for example `r = tail` and `a = (-1)`. Then we have:

$$\begin{aligned}
& (\mathtt{map\ (-1)} \circ \mathtt{tail}_{\mathtt{Int}})\ \mathtt{[1,2,3]} \\
= \;& \mathtt{map\ (-1)\ [2,3]} \\
= \;& \mathtt{[1,\ 2]} \\
= \;& \mathtt{tail}_{\mathtt{Int}}\ \mathtt{[0,\ 1,\ 2]} \\
= \;& (\mathtt{tail}_{\mathtt{Int}} \circ \mathtt{map\ (-1)})\ \mathtt{[1,2,3]}
\end{aligned}$$

which satisfies the theorem. Wadler continues with:

> " This theorem about functions of type `[x]` → `[x]` is pleasant but not earth-shaking. What is more exciting is that a similar theorem can be derived for *every* type "
>
> (Philip Wadler [37, p. 1])

The point/aim of this chapter is to show how to derive these theorems, and show their uses. But before we explain how to derive the theorems for free, we consider the concept of polymorphism:

Polymorphism

The word *polymorphism* can roughly mean "of many forms" from the ancient greek πολύς (polús) = many, and μορφή (morphḗ) = form, figure, silhouette. Polymorphism (in computer science) is the notion that you can provide one way of interacting with many different types of entities.

To say: polymorphic languages are languages in which some values and variables may have more than one type, polymorphic functions are functions whose operands (actual parameters) can have more than one type and polymorphic types are types whose operations apply to values of more than one type.

Polymorphism comes in different flavours. The three "main flavours" of polymorphism are *ad hoc polymorphism*, *inclusion polymorphism* and *parametric polymorphism*. [5]

Ad hoc polymorphism refers to polymorphic functions that can be applied to arguments of different types, but that behave differently depending on the type of the argument to which they are applied. This is also known as function overloading or operator overloading. These functions may behave in unrelated ways for each type.

Inclusion polymorphism is often referred to as subtyping. Subtyping indicates that the interface of type `S` is compatible with the interface of some type `T`. `S` is a subtype of `T` if every function that works on an entity of type `T` can also be correctly applied to an entity of type `S`.

Using parametric polymorphism, a function or a data type can be written generically so that it can handle values identically without depending on their type. This allows a language to maintain full static type-safety while being more expressive. Functions that exhibit parametric polymorphism are also called generic functions.

We shall be be dealing with the latter.

Explaining the magic trick

Wadler's magic trick originates from Reynolds' *abstraction theorem* [32]. Reynolds' goal in establishing the abstraction theorem was to give a precise meaning to the statement that a function is ″parametrically polymorphic″. Reynold uses his abstraction theorem to show that any polymorphic function expressible in the language defined in his paper is parametric.

With *parametricity*, we mean the property that captures the intuition that all instances of a polymorphic function act in the same way. To illustrate, $\mathtt{id}_{\mathtt{A}}$ and $\mathtt{id}_{\mathtt{A}'}$ are two different instances of the polymorphic function `id`. Nevertheless, we know that they behave the same way for every instance, and that is when we call such a function *parametrically polymorphic*.

The key insight of Reynolds is that you can read types as relations between values. It gives us a way to abstract from the shape of a data structure by only looking at the relations of the values. As an example, we can look at two pairs

```
(Green, Red) :: (Colour, Colour)
(True, False) :: (Bool, Bool)
```

These pairs intuitively have the same shape, they are both pairs `(a,a)` parametric in a. But how do we know this? Well, we can relate the values, Green to True and Red to False. Then – when you look at the spaces the related values occupy – the two pairs have the same shape.

This idea is easily extended. We can establish relations between composite types in terms of relations between values of the constitute types. The trick is to inductively transform type constructors into relations. The elements of the relations are theorems about inhabitants of the original type: our "theorems for free".

Remark 12 (Why relations?)
At first, it may seem odd to want to introduce relations, especially when we are dealing with *functions*. However, when dealing with algorithms, often, the specifications of the algorithm is what really matters, and these specifications are typically relations, not total functions. For instance, an algorithm that – even for the same input – can exhibit different behaviours on different runs (a non-deterministic algorithm) has a specification that can not be captured by a total function. Other algorithms do not always yield an answer or need an infinite amount of time to compute, these algorithms are partial and these too can not be modelled using total functions.

Relational Parametricity

To make the notion of parametricity completely precise, we have to be able to extend each type constructor `T` in our chosen programming language to a function from relation to relations. More formally, if we have a polymorphic function `f` of type `T` α for all types α. That is, for each type `A` there is an instance $\mathtt{f}_{\mathtt{A}}$ of type `T A`. The action of `T` is extended to binary relations, where if relation $\mathcal{A}$ has type $\mathtt{A} \Leftrightarrow \mathtt{A}'$, then we can define a new relation of the type $\mathtt{TA} \Leftrightarrow \mathtt{TA}'$. We do as follows:

From type-constructors to relations

As you shall see, each of the rules discussed below come down to: ”Terms evaluated in related environments yield related values”. This section closely follows the explanation and notation as found in [37]

Constant Types A constant type `T` such as `Bool` or `Int`, can be read as an identity relation.

$$\mathtt{I}_{\mathtt{T}} : \mathtt{T} \Leftrightarrow \mathtt{T} = \{(\mathtt{x},\mathtt{x}) \mid \mathtt{x} \in \mathtt{T}\}$$

In other words, this identity relation states that values of constant types are related only to themselves.

$$\mathtt{If}\ \mathtt{x},\mathtt{x}' \in \mathtt{T},\ \mathtt{then}\ (\mathtt{x},\mathtt{x}') \in \mathtt{I}_{\mathtt{T}}\ \mathtt{iff}\ \mathtt{x} = \mathtt{x}'.$$

These identity relations by themselves are not very interesting. The only “free theorem” we ever get from these monomorphic types is the monomorphised `id :: T -> T` function, because that is the only inhabitant of the relation $\mathtt{I}_{\mathtt{T}}$.

Product Types For any relations $\mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A}'$ and $\mathcal{B} : \mathtt{B} \Leftrightarrow \mathtt{B}'$, we can form a product relation $\mathcal{A} \times \mathcal{B} : (\mathtt{A} \times \mathtt{B}) \Leftrightarrow (\mathtt{A}' \times \mathtt{B}')$ by the construction:

$$((\mathtt{a},\mathtt{b}),(\mathtt{a}',\mathtt{b}')) \in \mathcal{A} \times \mathcal{B}$$
$$\mathtt{iff}$$
$$(\mathtt{a},\mathtt{a}') \in \mathcal{A}\ \mathtt{and}\ (\mathtt{b},\mathtt{b}') \in \mathcal{B}$$

Wadler explains: “That is, pairs are related if their corresponding components are related.”

List Types For any relation $\mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A}'$, we can construct a relation $[\mathcal{A}] : [\mathtt{A}] \Leftrightarrow [\mathtt{A}']$:

$$([x_1, \ldots, x_n], [x'_1, \ldots, x'_n]) \in [\mathcal{A}]$$
$$\mathtt{iff}$$
$$(x_1, x'_1) \in \mathcal{A} \text{ and } \ldots \text{ and } (x_n, x'_n) \in \mathcal{A}$$

Which Wadler describes as:

> That is, lists are related if they have the same length and corresponding elements are related.

Function Types For any relations $\mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A}'$ and $\mathcal{B} : \mathtt{B} \Leftrightarrow \mathtt{B}'$, we can construct the function relation $\mathcal{A} \to \mathcal{B} : (\mathtt{A} \to \mathtt{B}) \Leftrightarrow (\mathtt{A}' \to \mathtt{B}')$ as:

$$(f, f') \in (\mathcal{A} \to \mathcal{B})$$
$$\mathtt{iff}$$
$$(x, x') \in \mathcal{A} \text{ and } (f\ x, f'\ x') \in \mathcal{B}$$

This just states that two functions are related if and only if they take related arguments from the domain to related results in the codomain.

Universally Qualified Types Finally, we also need to translate polymorphic types to relations. This brings us to types of the form

```
forall χ. F χ
```

where `F` is some type constructor. We have to give an interpretation of $\forall$ as an operation on relations:

Let $\mathcal{F}(\mathcal{X})$ be a relation depending on $\mathcal{X}$. So that $\mathcal{F}$ corresponds to a function from relations to relations, where for every relation $\mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A}'$, $\mathcal{F}$ takes it to a corresponding relation $\mathcal{F}(\mathcal{A}) : \mathtt{FA} \Leftrightarrow \mathtt{F'A'}$.

Then the relation

$$\mathtt{forall}\ \mathcal{X}.\mathcal{F}(\mathcal{X}) : \mathtt{forall\ X\ .\ F\ X} \Leftrightarrow \mathtt{forall\ X'\ .\ F'\ X'}$$

can be defined by

$$(f, f') \in \mathtt{forall}\ \mathcal{X}.\mathcal{F}(\mathcal{X})$$
$$\mathtt{iff}$$
$$\mathtt{forall}\ \mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A}'.(f_{\mathtt{A}}, f'_{\mathtt{A}'}) \in \mathcal{F}(\mathcal{A})$$

Nothing special is going on here. This just states that a polymorphic expression must preserve relationship under every substitution of its polymorphic type variable to any possible type.

$$
\begin{array}{lll}
\mathcal{T}_{\alpha,\eta} & = \eta(\alpha) \quad \alpha \text{ is a free variable} \\
\mathcal{T}_{\tau,\eta} & = \mathtt{id}_{\tau} \quad \tau \text{ is a constant type} \\
\mathcal{T}_{(\tau,\tau'),\eta} & = \{((\mathtt{a},\mathtt{b}),(\mathtt{a}',\mathtt{b}')) \mid (\mathtt{a},\mathtt{a}') \in \mathcal{T}_{\tau,\eta} \wedge (\mathtt{b},\mathtt{b}') \in \mathcal{T}_{\tau',\eta}\} \\
\mathcal{T}_{(\tau+\tau'),\eta} & = \{(\mathsf{inl}\ \mathtt{a}, \mathsf{inl}\ \mathtt{a}') \mid (\mathtt{a},\mathtt{a}') \in \mathcal{T}_{\tau,\eta}\} \\
 & \qquad\qquad \cup \\
 & \quad \{(\mathsf{inr}\ \mathtt{b}, \mathsf{inr}\ \mathtt{b}') \mid (\mathtt{b},\mathtt{b}') \in \mathcal{T}_{\tau',\eta}\} \\
\mathcal{T}_{[\tau],\eta} & = \{([\mathtt{x}_1,\mathtt{x}_2,\ldots,\mathtt{x}_n],[\mathtt{x}'_1,\mathtt{x}'_2,\ldots,\mathtt{x}'_n]) \\
 & \qquad \mid (\mathtt{x}_1,\mathtt{x}'_1) \in \mathcal{T}_{\tau,\eta} \wedge \ldots \wedge (\mathtt{x}_n,\mathtt{x}'_n) \in \mathcal{T}_{\tau,\eta}\} \\
\mathcal{T}_{\tau\to\tau',\eta} & = \{(\mathtt{f},\mathtt{f}') \mid \forall (\mathtt{x},\mathtt{x}') \in \mathcal{T}_{\tau,\eta}, (\mathtt{f}\ \mathtt{x}, \mathtt{f}'\ \mathtt{x}') \in \mathcal{T}_{\tau',\eta}\} \\
\mathcal{T}_{\forall\alpha.\tau,\eta} & = \{(\mathtt{g},\mathtt{g}') \mid \forall \mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A'}.(\mathtt{g}_{\mathtt{A}},\mathtt{g}'_{\mathtt{A'}}) \in \mathcal{T}_{\tau,\eta[\alpha\mapsto\mathcal{A}]}\}
\end{array}
$$

Figure 8.1: Inductive structure of the function $\mathcal{T}$

Coproduct Types Oddly enough Wadler's paper is completely quiet when it comes to coproduct types. Why? No idea. Leaving that aside, after reading through the above examples, we should be able to define the construction for $\mathcal{A} + \mathcal{B}$ ourselves:

For any two relations $\mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A}'$ and $\mathcal{B} : \mathtt{B} \Leftrightarrow \mathtt{B}'$, the relation $\mathcal{A} + \mathcal{B} : \mathtt{A} + \mathtt{B} \Leftrightarrow \mathtt{A}' + \mathtt{B}'$ is defined by:

$$
\begin{gathered}
(\mathtt{x},\mathtt{x}') \in \mathcal{A} + \mathcal{B} \\
\mathtt{iff} \\
\mathtt{There\ exists}\ (\mathtt{a},\mathtt{a}') \in \mathcal{A}.\mathtt{x} = \mathsf{inl}\ \mathtt{a} \ \wedge \mathtt{x}' = \mathsf{inl}\ \mathtt{a}' \\
\vee \\
\mathtt{There\ exists}\ (\mathtt{b},\mathtt{b}') \in \mathcal{B}.\mathtt{x} = \mathsf{inr}\ \mathtt{b} \ \wedge \mathtt{x}' = \mathsf{inr}\ \mathtt{b}'
\end{gathered}
$$

We use a more compact notation, inspired by the notation used in [36]. For every type τ and every relation environment η that maps all free variables of τ to relations between closed types, we define the relation $\mathcal{T}_{\tau,\eta}$ by induction on the type structure, see Figure 8.1.

Then **parametricity** as presented in [32] and [37], implies the following fundamental property of the relational interpretation of types:

If $\mathtt{t} :: \tau$ is a closed term of closed type, then:

$$(\mathtt{t},\mathtt{t}) \in \mathcal{T}_{\tau,\emptyset}$$

Parametricity gives some interesting results and has uses discussed further below.

Deriving a theorem

We have been given a systematic way to interpret types as relations and we know a parametric polymorphic term $\texttt{t}$ of type τ must satisfy the parametricity property:

$$(\texttt{t}, \texttt{t}) \in \mathcal{T}_{\tau, \emptyset}$$

Earlier we noted that our free theorems are inhabitants of these inductively constructed relations. And so to get a free theorem we must "extract" members from this relation. To continue with the derivation of a free theorem, one has to unfold the given parametricity relation using the various actions on relations as described above.

Often to obtain a more useful theorem you need to specialise relations to some well-chosen relation. The case where that relation is a relational interpretation of a function, is often fruitful.

We take a look at the example given in the introduction of this section:

```
r :: [x] -> [x]
```

Parametricity of this function gives us:

$$(\texttt{r}, \texttt{r}) \in \mathcal{T}_{\texttt{forall x . [x]->[x]}, \emptyset}$$

We start by unfolding the definition of forall on relations:

$$\begin{array}{l} \texttt{forall}\, \mathcal{A} : \texttt{A} \Leftrightarrow \texttt{A}' \\ \quad (\texttt{r}_{\texttt{A}}, \texttt{r}_{\texttt{A}'}) \in \mathcal{T}_{\texttt{[x]->[x]}, \{\texttt{x} \mapsto \mathcal{A}\}} \end{array}$$

Now we unfold the definition of → on relations:

$$\begin{array}{l} \texttt{forall}\, \mathcal{A} : \texttt{A} \Leftrightarrow \texttt{A}' \\ \quad \texttt{forall}\, (\texttt{xs}, \texttt{xs}') \in \mathcal{T}_{\texttt{[x]}, \{\texttt{x} \mapsto \mathcal{A}\}} \\ \qquad (\texttt{r}_{\texttt{A}}\, \texttt{xs}, \texttt{r}_{\texttt{A}'}\, \texttt{xs}') \in \mathcal{T}_{\texttt{[x]}, \{\texttt{x} \mapsto \mathcal{A}\}} \end{array}$$

We could unfold this even further using the definition of the list operation on relations. Another thing to do however, is to take a look at the introduced relation $\mathcal{A}$, this is the only "variable" we can play with. And ideally we would like to extract from our parametricity property a powerful statement, such as an equality. So to do this, we must introduce an equality by cleverly choosing the relation. One way of introducing this equality is by specialising it to the graph of a function `f :: A -> A'`.

$$\mathtt{graph(f)} = \{(\mathtt{x}, \mathtt{x}') \mid \mathtt{x} \in \mathtt{A}, \mathtt{x}' = \mathtt{f\ x}\}$$

The graph of a function introduces an equality, two elements are related if they are equal under an application of `f`. The result assumes the following form:

$$\begin{array}{l}
\mathtt{forall\ graph(f)} \\
\quad \mathtt{forall\ (xs, xs')} \in \mathcal{T}_{\mathtt{[x]}, \{\mathtt{x} \mapsto \mathtt{graph(f)}\}} \\
\qquad (\mathtt{r}_{\mathtt{A}}\ \mathtt{xs}, \mathtt{r}_{\mathtt{A}'}\ \mathtt{xs}') \in \mathcal{T}_{\mathtt{[x]}, \{\mathtt{x} \mapsto \mathtt{graph(f)}\}} \\
\Leftrightarrow \\
\mathtt{forall\ graph(f)} \\
\quad \mathtt{forall\ xs,} \\
\qquad \mathtt{xs'\ ==\ map\ f\ xs} \implies \mathtt{map\ f} \circ \mathtt{r}_{\mathtt{A}}\ \mathtt{xs\ ==\ r}_{\mathtt{A}'}\ \mathtt{xs'} \\
\Leftrightarrow \\
\mathtt{forall\ graph(f)} \\
\quad \mathtt{map\ f} \circ \mathtt{r}_{\mathtt{A}}\ \mathtt{xs\ ==\ r}_{\mathtt{A}'} \circ \mathtt{map\ f\ xs}
\end{array}$$

As `graph(f)` is completely defined by `f` and not used in the rest of the theorem we may remove mention of the graph entirely:

$$\begin{array}{l}
\mathtt{forall\ f\ ::\ A\ ->\ A'} \\
\quad \mathtt{map\ f} \circ \mathtt{r}_{\mathtt{A}} \mathtt{\ ==\ r}_{\mathtt{A}'} \circ \mathtt{map\ f}
\end{array}$$

That is the version as presented in the introduction of the section. More examples, as taken from [37], are given in Figure 8.2.

```
Assume f :: A -> A' and g : B -> B'

head :: [x] -> x
f ∘ head_A == head_A' ∘ map f


tail :: [x] -> x
f ∘ tail_A == tail_A' ∘ map f


(++) :: [x] -> [x] -> x
map f (xs ++_A ys) == (map f xs) ++_A' (map f ys)


concat :: [[x]] -> [x]
map f ∘ concat_A == concat_A' ∘ map ( map f)


fst :: (x,y) -> x
f ∘ fst_AB == fst_A'B' ∘ (bimap f g)


snd :: (x,y) -> y
g ∘ snd_AB == fst_A'B' ∘ (bimap f g)


zip :: ([x],[y]) -> [(x,y)]
map (bimap f g) ∘ zip_AB == zip_A'B' ∘ (bimap (map f) (map g))


filter :: (x -> Bool) -> [x] -> [x]
map f ∘ filter_A (p' ∘ f) == filter_A' p' ∘ map f


sort :: (x -> x -> Bool) -> [x] -> [x]
forall x,y ∈ A .
   x <_A y = (f x <_A' f y)
    ⟹  map f ∘ sort_A <_A  == sort_A' <_A' ∘ map f


fold :: (x -> y -> y) -> y -> [x] -> y
forall x ∈ A, y ∈ B .
   g (x ⊕ y) = (f x) ⊗ (f y) and b u = u'
    ⟹  g ∘ fold_AB (⊕) u == fold_A'B' (⊗) u' ∘ map f


id :: x -> x
f ∘ id_A == id_A' ∘ f
```

Figure 8.2: Examples of free theorems

Parametricity and Naturality

Suppose we tried to find the free theorem belonging to a polymorphic function of type:

```
r :: forall a. F a -> G a
```

Then if you accept the premise that we can define such a function for every functor `F` and `G`. Then parametricity of `r` states:

$$(\mathtt{r},\mathtt{r}) \in \mathcal{T}_{\forall\alpha.\mathtt{F}\alpha\to\mathtt{G}\alpha,\emptyset}$$

Unfolding the definition we get:

$$(\mathtt{r},\mathtt{r}) \in \mathcal{T}_{\forall\alpha.\mathtt{F}\alpha\to\mathtt{G}\alpha,\emptyset}$$

$$\Leftrightarrow [\![\mathtt{unfolding}\ \forall]\!]$$

$$\begin{array}{l}\forall\mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A}' \\ \quad (\mathtt{r},\mathtt{r}) \in \mathcal{T}_{\mathtt{F}\alpha\to\mathtt{G}\alpha,\,[\alpha\mapsto\mathcal{A}]}\end{array}$$

$$\Leftrightarrow [\![\mathtt{unfolding}\ \mathtt{A} \to \mathtt{B}]\!]$$

$$\begin{array}{l}\forall\mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A}' \\ \quad \forall(\mathtt{x},\mathtt{x}') \in \mathcal{T}_{\mathtt{F}\alpha,\,[\alpha\mapsto\mathcal{A}]} \\ \qquad (\mathtt{r}\ \mathtt{x},\mathtt{r}'\ \mathtt{x}') \in \mathcal{T}_{\mathtt{G}\alpha,\,[\alpha\mapsto\mathcal{A}]}\end{array}$$

Now we specialise the relation $\mathcal{A}$ to the graph of a function `h :: A -> A'`, i.e, setting $\mathcal{A}$ = `graph(h) := {(x, y) | h x = y}`. Additionally, we realise that if we do so, then:

$$\begin{array}{l}\mathtt{F}(\mathcal{A}) = \mathtt{F}(\mathtt{graph}(\mathtt{h})) = \mathtt{graph}(\mathtt{fmap}\ \mathtt{h}) \\ \mathtt{G}(\mathcal{A}) = \mathtt{G}(\mathtt{graph}(\mathtt{h})) = \mathtt{graph}(\mathtt{fmap}\ \mathtt{h})\end{array}$$

So that we can continue, with the following:

$$\begin{aligned}
&\forall\mathtt{(a, b)} \in \mathtt{graph(h))}\\
&\quad\forall\mathtt{(x, x')} \in \mathtt{graph(fmap\,h)}\\
&\qquad\mathtt{(r\,x, r'\,x')} \in \mathtt{graph(fmap\,h)}
\end{aligned}$$

$$\Leftrightarrow [\![\mathtt{(x, y)} \in \mathtt{graph(h)\ iff\ h\,x = y}]\!]$$

$$\begin{aligned}
&\mathtt{h\,a == b}\\
&\quad\Longrightarrow \mathtt{(fmap\,h)\ x == x'}\\
&\qquad\Longrightarrow \mathtt{fmap\,h\ (r\,x) == r'\,x'}
\end{aligned}$$

$$\Leftrightarrow$$

$$\begin{aligned}
&\mathtt{h\,a == b}\\
&\quad\Longrightarrow \mathtt{fmap\,h\ (r\,x) == r'\ (fmap\,h\,x)}
\end{aligned}$$

We find the free theorem:

```
fmap h . r == r . fmap h
```

But wait! This is exactly the naturality condition. Furthermore we had a function from one functor to another. That means that this function is a natural transformation between the two functors. Consequently, that means that naturality follows from parametricity of the function. Or said differently: every parametric polymorphic function is a natural transformation!

Uses

> “ Modern programmers are too busy to bother proving theorems about their programs. Free theorems provide a handy shortcut, as they allow us to conclude useful properties of functional programs without any proof whatsoever, provided the programs are suitably polymorphic ”
>
> (Jennifer Hackett & Graham Hutton [12])

What is the use of these theorems? Wadler himself states that the results are encouraging:

> “ In general, the equations derived from types are a useful form of algebraic manipulation. For example, many laws stated above allow us to push “map through a function”. ”
>
> (Philip Wadler [37])

This type of manipulation, where we push "map through a function" can be used to improve the efficiency of code.

Example 19 (filter)

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : filter p xs
                             else filter p xs
```

The following law can be derived *solely* from the parametric polymorphic type of `filter`.

$$\texttt{map h} \circ \texttt{filter (p} \circ \texttt{h) = filter p} \circ \texttt{map h}$$

This result might not seem very interesting. However it justifies a trivial optimization. On the left size we apply `h` twice to every element of the list and on right side of the equation only once. Clearly if we have a lot of these trivial theorems, which can be generated mechanically, whilst only needing to look at the type, then we have a rather nice collection of small optimizations. A compiler can use these optimization to rewrite code.

These improvements for now depend on the human knowledge that one must be faster than the other. But there are (promising) attempts to provide frameworks in order to calculate more efficient methods. [34, 12]

An interesting result that follows from the parametricity theorem is that certain functions can be *specified* by a parametricity property. That is the parametricity property for that type has a unique solution.

Example 20 (Uniqueness of functions)

As an example we give the type `(a -> b, a) -> b`. You might recognise this as the type of function application. The type constructor `T` belonging to this type, maps two types `A` and `B` to the type `(A -> B, A) -> B`. The extension of this type constructor to relations maps two relations $\mathcal{A}$ and $\mathcal{B}$ to the relation $\mathcal{T}_{(\texttt{A}\rightarrow\texttt{B},\texttt{A})\rightarrow\texttt{B},\emptyset}$

Now suppose `eval` is any parametrically polymorphic function with the same type. Then parametricity claims that

$$(\texttt{eval},\texttt{eval}) \in \mathcal{T}_{\forall\alpha,\beta.(\alpha\rightarrow\beta,\alpha)\rightarrow\beta,\emptyset}$$

We can unfold this definition using the rules above and we gain: For all functions `f, f' :: A -> B`, and all `a, a' :: A`:

> $$\begin{aligned}
> &\texttt{forall}\ \mathcal{A} : \texttt{A} \Leftrightarrow \texttt{A}', \mathcal{B} : \texttt{B} \Leftrightarrow \texttt{B}'\\
> &\quad \texttt{forall}\ (\texttt{x}, \texttt{x}') \in \mathcal{T}_{\alpha, \{\alpha \mapsto \mathcal{A}, \beta \mapsto \mathcal{B}\}}\\
> &\qquad \implies (\texttt{f x}, \texttt{f}'\ \texttt{x}') \in \mathcal{T}_{\beta, \{\alpha \mapsto \mathcal{A}, \beta \mapsto \mathcal{B}\}} \wedge (\texttt{a}, \texttt{a}') \in \mathcal{T}_{\alpha, \{\alpha \mapsto \mathcal{A}, \beta \mapsto \mathcal{B}\}}\\
> &\qquad\quad \implies (\texttt{eval}_{\texttt{AB}}\ \texttt{f a}, \texttt{eval}_{\texttt{A}'\texttt{B}'}\ \texttt{f}'\ \texttt{a}') \in \mathcal{T}_{\beta, \{\alpha \mapsto \mathcal{A}, \beta \mapsto \mathcal{B}\}}
> \end{aligned}$$
>
> We can see that function application satisfies this property. And now we can see that function application is uniquely defined by its parametricity property. To see this, we instantiate $\mathcal{A}$ to the singleton set $\{(\texttt{a}, \texttt{a})\}$ and $\mathcal{B}$ to the singleton set $\{(\texttt{f c}, \texttt{f c})\}$ then the above result give us: $(\texttt{eval f c}, \texttt{eval f c}) \in \mathcal{B}$. That is, `eval f c = f c`. And so we find that there is only one function of type `(a -> b, a) -> b`.

Most parametricity properties do not have a unique solution however. For example, both the identity function on lists and the reverse function satisfy parametricity property of function `f :: [a] -> [a]`.

Nonetheless, there are some for which it is unique, and that allows us to say some interesting things. Other functions that boast similar uniqueness are the identity function, projection and fmap. The latter allows us to say: If there is an implementation of fmap for a given type constructor, one that preserves identity, then it must be unique. We already knew we could derive an implementation of fmap for arbitrary types. But now we know that these are even unique! In fact this leaves little reason to not just derive them in Haskell, using `deriving(Functor)`.

⌈9⌋ Recursion Schemes

We like to reuse code and like our code to be generic, so much even that we take the level of "genericity" of a programming language as a highly-valued criterion for usability. One type of genericity we encounter is the ability to capture "programming patterns" as parametrisable abstractions. [2] One place to find such pattern is in recursive functions. They often follow a similar pattern: they pattern-match on the input and continue to recur until they hit a base-case. Another pattern we often find ourselves is a function that takes a seed value, to produce the first value in some container like datatype, and creates a new seed to produce the rest of the datatype. These patterns of recursive functions are known as recursion schemes.

To properly discuss these, we will need some more machinery to reason about them. In specific, we need a proper model for recursive types. A theory for such datatypes would need a calculus for program transformation [20]. Earlier I introduced these as algebraic types where the type appears on both sides of the equation, that, however, is not very specific and does not allow for such calculations.

We start this section with the concept of algebras and homomorphisms between these. We will use these algebras to give a better understanding of datatypes. We will see that recursive datatypes are algebras but of a special kind. Special in the way that they are "initial". We also introduce the notation of a "catamorphism" and an "anamorphism" – special kinds of homomorphisms first introduced in a programming setting by Meijer et al[22]. We show that these cata- and anamorphisms provide a nice abstraction for our recursive functions.

Algebras

> An **algebra** is a set, together with a number of operations (functions) that return values in that set. The set is called the carrier of the algebra.

Example 21 (Natural Numbers)

$(\mathbb{N}, 0, +)$ is an algebra with carrier set $\mathbb{N}$ and the functions:

$$\begin{aligned} 0 &:: () \to \mathbb{N} \\ (+) &:: \mathbb{N} \times \mathbb{N} \to \mathbb{N} \end{aligned}$$

We treat fixed elements, like $0 \in \mathbb{N}$, as nullary operations. This is indicated by the source type $()$, the initial object.

Other examples are:

$$\begin{aligned} &(\mathbb{R}, 1, *), \text{ with } 1 :: () \to \mathbb{R}, (*) :: \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ &(\mathbb{B}, \mathtt{true}, \wedge), \text{ with } \mathtt{true} :: () \to \mathbb{B}, (\wedge) :: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ &(\mathbb{B}, \mathtt{true}, \vee), \text{ with } \mathtt{false} :: () \to \mathbb{B}, (\vee) :: \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ &([A], [], ++), \text{ with } [] :: () \to [A], (++) :: [A] \times [A] \to [A] \end{aligned}$$

The above examples are all similar, they have been chosen in such a way that they all have the same number of operations with the same arities. They are all related by the following abstract algebra:

$$(A, e, \oplus), \text{ with } e :: () \to A, \oplus :: A \times A \to A$$

This abstract algebra describes a *class* of algebras. So the above examples all belong to this class. An example of another class of algebras is:

$$\begin{aligned} &(\mathbb{N}, +, +1), \text{ with } (+) :: \mathbb{N} \times \mathbb{N} \to \mathbb{N}, (+1) :: \mathbb{N} \to \mathbb{N} \\ &(\mathbb{R}, *, *2), \text{ with } (*) :: \mathbb{R} \times \mathbb{R} \to \mathbb{R}, (*2) :: \mathbb{R} \to \mathbb{R} \end{aligned}$$

which belong to this abstract class:

$$(A, \oplus, f), \text{ with } (\oplus) :: A \times A \to A, f :: A \to A$$

An algebra may belong to multiple classes, as an example the first of the above concrete algebra also belongs to the class:[1]

$$(A, \oplus, f), \text{ with } (\oplus) :: \mathbb{N} \times A \to A, f :: A \to A$$

Remark 13 (Algebras from datatype)
A recursively defined datatype determines, in a natural way, an algebra. A simple example is the datatype Nat defined by:

```
data Nat = zero | succ Nat
```

[1] See A changed to $\mathbb{N}$ in the type of $\oplus$

The corresponding algebra is: $(\mathtt{Nat}, \mathtt{zero}, \mathtt{succ})$, with

$$\begin{aligned}\mathtt{zero} &:: () \to \mathtt{Nat}\\ \mathtt{succ} &:: \mathtt{Nat} \to \mathtt{Nat}\end{aligned}$$

This shows that a datatype determines an algebra in which the carrier of the algebra is the datatype itself, and the constructors of the datatype are the operations of the algebra.

Homomorphisms

A homomorphism is a map between two algebras, which must be from the same class that preserves the structure of the class. This map takes elements between their carrier sets. For example, the function $\mathtt{exp} :: \mathbb{N} \to \mathbb{R}$ is a homomorphism from the source algebra $(\mathbb{N}, 0, (+))$ to the target algebra $(\mathbb{R}, 1, (*))$.

Where respecting the structure means that exp satisfies the following properties:

$$\begin{aligned}\mathtt{exp}(0) &= 1\\ \mathtt{exp}(x + y) &= \mathtt{exp}(x) * \mathtt{exp}(y)\end{aligned}$$

Another name for the "preservation of structure" is that exp is "compatible with the operations". One could see that exp acts as a sort of bridge between the structures, so that + and $*$ work the same way on the same elements, under application of exp.

For the class of algebras whose generic algebra is

$$(\mathtt{A}, \mathtt{e}, \oplus),\ \text{with } \mathtt{e} :: () \to \mathtt{A},\ \oplus :: \mathbb{N} \times \mathtt{A} \to \mathtt{A}$$

we have that a homomorphism $\mathtt{h} :: (\mathtt{A}, \mathtt{e}, \oplus) \to (\mathtt{B}, \mathtt{u}, \otimes)$ satisfies:

$$\begin{aligned}\mathtt{h}\ \mathtt{e} &= \mathtt{u}\\ \mathtt{h}(\mathtt{x} \oplus \mathtt{y}) &= \mathtt{x} \otimes \mathtt{h}(\mathtt{y})\end{aligned}$$

In this new case, you might wonder why h is not applied to x. The answer is: because that wouldn't make sense; h is a function from $\mathtt{A} \to \mathtt{B}$, whilst $\mathtt{x} \in \mathbb{N}$. But more importantly, it shows that the current notion of a homomorphism is dependent on the specific algebra class. We shall see how to define the idea of a homomorphism generically, being able to talk about homomorphisms generically is important in the context of for example catamorphisms – shown later in this section. To that end, we shall have to introduce another concept: the *F-algebra*.

F-algebras

If we look at the above notation for algebras we find that it quickly becomes unwieldy, simply consider an algebra with more operations. Preferably, we would want a more compact definition, as that would give a better basis for computation. Additionally, as this is a paper on category theory, so we would like a categorical definition of algebras. It would also make sense if it was possible to capture these algebraic structures, be category theory is all about structure.

What we first notice – when we take a look at a class of algebras – is that it defines an abstract pattern for a set of algebras. This pattern is a set of operation typings that depend on the carrier set. This can actually be captured by a functor (as it would take the carrier set to a set of operations that work on that set (a type constructor)). I shall use an example to show this:

> **Example 22** (Algebras and Functors)
> Let us, once more, take a look at the type of the Peano naturals.
>
> ```
> data Nat = zero | succ Nat
> ```
>
> The corresponding algebra is: (Nat, zero, succ), with
>
> $$\begin{aligned}\texttt{zero} &:: () \to \texttt{Nat}\\ \texttt{succ} &:: \texttt{Nat} \to \texttt{Nat}\end{aligned}$$
>
> We know that these function types may be represented as exponential objects:
>
> $$\begin{aligned}\texttt{zero} &\in \texttt{Nat}^{1}\\ \texttt{succ} &\in \texttt{Nat}^{\texttt{Nat}}\end{aligned}$$
>
> Now the Cartesian product of these two sets is:
>
> $$\texttt{Nat}^{1} \times \texttt{Nat}^{\texttt{Nat}}$$
>
> Then using some algebraic manipulation as introduced in section 4, we can rewrite it as:
>
> $$\texttt{Nat}^{\texttt{1+Nat}}$$
>
> Which is the exponential notation of the function function type:
>
> $$\texttt{1 + Nat} \to \texttt{Nat}$$

To say, the entire algebra $(\mathtt{Nat}, \mathtt{zero}, \mathtt{succ})$, can be captured by the function $[\mathtt{zero}, \mathtt{succ}] :: 1 + \mathtt{Nat} \to \mathtt{Nat}$. Recall that $[\,,\,]$ is the coproduct mediating arrow. That is, the triangles in the following diagram commute:

$$\begin{tikzcd}
1 + \mathtt{Nat} \arrow[dr, dashed, "{[\mathtt{zero},\mathtt{succ}]}"] & \mathtt{Nat} \arrow[l, "\mathtt{inr}"] \arrow[d, "\mathtt{succ}"] \\
1 \arrow[u, "\mathtt{inl}"] \arrow[r, "\mathtt{zero}"] & \mathtt{Nat}
\end{tikzcd}$$

What is more, we can generalize this statement, by abstracting away from the type `Nat` by replacing it with the variable `z`. In this way, we are led to consider the unary functor `N` defined by:

$$\mathtt{N}\,\mathtt{z} = 1 + \mathtt{z}$$

The functor `N` captures the pattern of the inductive formation of the Peano naturals. We find the entire algebra $(\mathtt{Nat}, \mathtt{zero}, \mathtt{succ})$ defined by:

$$[\mathtt{zero}, \mathtt{succ}] :: \mathtt{N}\,\mathtt{Nat} \to \mathtt{Nat}$$

This example shows how we come to define an *F-algebra*.

> An **F-algebra** is a triple consisting of an endofunctor `F`, an object `a`, and a morphism
> $$\mathtt{F}\mathtt{a} \to \mathtt{a}$$

The object is often called the carrier, an underlying object or, in the context of programming, the carrier type. The morphism is often called the evaluation function or the structure map. Think of the functor `F` as forming expressions and the morphism as evaluating them.

F-algebra in Haskell

Here's Haskell's definition of an F-algebra:

```haskell
type Algebra f a = f a -> a
```

In the example of Peano naturals. The functor in question ($\mathtt{N}\,\mathtt{z} = 1 + \mathtt{z}$) is:

```haskell
data NatF a = zeroF | succF a
```

And the object is the datatype `Nat`. Our evaluation function `NatF Nat -> Nat` becomes:

```
evalNatF :: NatF Nat -> Nat
evalNatF zeroF      = zero
evalNatF (succF x)  = succ x
```

This is the definition of an algebra in category theory based on the notion of a functor.

Recursion

We first look at the simple inductively defined datatype, that of the Peano naturals.

```
data Nat = zero | succ Nat
```

We can try to list all possible values using the functor that captured the construction of these members.

```
zeroF
succF zeroF
succF (succF zeroF)
succF (succF (succF zeroF))
...
```

Continuing this process, we can write a symbolic equation:

$$\texttt{succF}_{n+1}\ \texttt{zeroF} = \texttt{succF}\ (\texttt{succF}_{n}\ \texttt{zeroF})$$

Conceptually, if we have done this an infinite amount of times (because of course we cannot actually do that), then applying succF one more time, does not change anything, we end up with virtually the same type. This type "at infinity" is like the type `Nat`. As an analogy, adding 1 to ∞ is still ∞. This is of course just a hand-waving argument, an intuition, but the thought is captured by a *fixed point*, an object defined as:

$$\texttt{Fix f} = \texttt{f (Fix f)}$$

This object is fixed, because when applying `f` to it, it does not change.

> **Remark 14** (What about a type with the same pattern functor?)
> Notice how the definition of the fixed point does not mention the carrier type anywhere. The fixed point is independent of its carrier type. If we took another algebra from the same class, with a different carrier and conceptually applied our constructor functor without end, we would end up with the same fixed point. The starting point of our journey doesn't matter, we always end up in the same place. This is not always

> true for an arbitrary endofunctor in an arbitrary category, but in the category **Set** things are nice.

In Haskell, the definition of a fixed point is:

```haskell
newtype Fix f = in (f (Fix f))
```

The nice thing about this fixed point is that it is defined only using the way we construct the type from itself. And this allows us to capture the entire type `Nat`, using only the functor that captured said inductive formation. To view our Peano numbers as a fixed point definition, let us once again abstract from `Nat` on the right side by replacing it with the variable `z`:

$$\mathtt{N\ z = 1 + z}$$

The functor `N` captures the pattern of the inductive formation of the Peano naturals. The point is that we use this to rewrite the definition of `Nat` to the fixed point.

```haskell
data Nat = in (N Nat) -- Here I replaced Fix N with Nat
```

Apparently, the pattern functor `N` uniquely determines the datatype `Nat`. Then our datatype `Nat` as F-algebra for the endofunctor `N` becomes `(FixN, zero+succ)`.

Category of F-algebras

We have found that we can uniquely determine a recursive datatype using its pattern functor, we could then start to wonder how this datatype relates to other datatypes; What homomorphisms originate and which end up at this algebra? When we wonder how something relates to something else it becomes natural to use category to look at these relations. But to that end we need to see whether F-algebras with homomorphism as morphisms form a category.

Fist we check how these homomorphisms between algebras translate to homomorphism between F-algebras. A homomorphism is compatible with the operations, but now that these operations have been incorporated into the evaluation function. We say that the homomorphism `h` is compatible with the evaluator functions. A homomorphism $\mathtt{h :: (A, f) \to (B, g)}$ is compatible with the evaluator functions `f` and `g` if

$$\mathtt{g \circ F\ h = h \circ f}$$

Or diagrammatically, the following square commutes:

$$\begin{tikzcd}
\mathtt{F\,A} \arrow[r, "\mathtt{F\,h}"] \arrow[d, "\mathtt{f}"] & \mathtt{F\,B} \arrow[d, "\mathtt{g}"] \\
\mathtt{A} \arrow[r, "\mathtt{h}"] & \mathtt{B}
\end{tikzcd}$$

This is indeed a category. Identity homomorphisms form identity morphism in this category of F-algebras,

$$\begin{tikzcd}
\mathtt{F\,A} \arrow[r, "\mathtt{F\,id}"] \arrow[d, "\mathtt{f}"] & \mathtt{F\,A} \arrow[d, "\mathtt{f}"] \\
\mathtt{A} \arrow[r, "\mathtt{id}"] & \mathtt{A}
\end{tikzcd}$$

and the composition of two homomorphisms is a homomorphism:

$$\begin{tikzcd}
\mathtt{F\,A} \arrow[r, "\mathtt{F\,m}"] \arrow[d, "\mathtt{f}"] & \mathtt{F\,B} \arrow[r, "\mathtt{F\,n}"] \arrow[d, "\mathtt{g}"] & \mathtt{F\,C} \arrow[d, "\mathtt{h}"] \\
\mathtt{A} \arrow[r, "\mathtt{m}"] & \mathtt{B} \arrow[r, "\mathtt{n}"] & \mathtt{C}
\end{tikzcd}$$

Initial algebra

So how does this fixed point relate to other algebras in the category of F-algebras? As it turns out, this fixed point is the initial object in the category of F-algebras over the endofunctor. We can prove this using category theory:

What we would like to show is that an initial object in the category of F-algebra's (also called an *initial algebra*) is a fixed point of `F`. Now we first described the fixed point using equality.

$$\mathtt{F(Fix\,F) = Fix\,F}$$

However in category theory this equality is changed to an isomorphism:

$$\mathtt{F(Fix\,F)} \cong \mathtt{FixF}$$

This result is know as *Lambek's Theorem* [16]:

> Let `i` be the carrier of the initial algebra and `j :: F i → i` its evaluator. Then the evaluator of the initial algebra is an isomorphism.

The proof of Lambek's theorem is as thus:

Since (i, j) is initial there exists a *unique* homomorphism m to every other F-algebra. So lets create an algebra $(F\ i, F\ j)$ by lifting j.

$$\begin{tikzcd}
F\ (Fi) \arrow[d, "F\ j"] \\
F\ i
\end{tikzcd}$$

Now the initiality property gives us the unique homomorphism $m :: (i, j) \to (F\ i, F\ j)$ or the following commuting diagram:

$$\begin{tikzcd}
F\ i \arrow[r, "F\ m"] \arrow[d, "j"] & F(F\ i) \arrow[d, "F\ j"] \\
i \arrow[r, "m"] & F\ i
\end{tikzcd}$$

Obviously we also have that j is a morphism from (Fi, Fj) to (i, j) (the need commutativity is $j\ (F\ i) = j\ (F\ i))$. Or equivalently the following diagram commutes:

$$\begin{tikzcd}
F(F\ i) \arrow[r, "F\ j"] \arrow[d, "F\ j"] & F\ i \arrow[d, "j"] \\
F\ i \arrow[r, "j"] & i
\end{tikzcd}$$

Now we smash these two diagrams together to get:

$$\begin{tikzcd}
F\ i \arrow[r, "F\ m"] \arrow[d, "j"] & F(F\ i) \arrow[r, "F\ j"] \arrow[d, "F\ j"] & F\ i \arrow[d, "j"] \\
i \arrow[r, "m"] \arrow[rr, dashed, bend right, "id_i"'] & F\ i \arrow[r, "j"] & i
\end{tikzcd}$$

The diagram shows that $j \circ m$ is a homomorphism from (i, j) to itself. However, since (i, j) is initial id_i is the only morphism $(i, j) \to (i, j)$. Therefore $j \circ m = id_i$. We can now use this and the commuting property of the left diagram to show:

$$\begin{aligned} \mathtt{m} \circ \mathtt{j} &= \mathtt{F\ j} \circ \mathtt{F\ m} \\ &= \mathtt{F\ (j} \circ \mathtt{m)} \\ &= \mathtt{F\ id_i} \\ &= \mathtt{id_{F\ i}} \end{aligned}$$

Consequently with $\mathtt{j} \circ \mathtt{m} = \mathtt{id_i}$ and $\mathtt{m} \circ \mathtt{j} = \mathtt{id_{F\ i}}$ we have shown that `j` is an isomorphism, with inverse `m`. Therefore:

$$\mathtt{F\ i} \cong \mathtt{i}$$

But that is the same as saying that `i` is a fixed point.

Catamorphisms

So what did we gain by doing this? We can now say that the fixed point of any pattern functor is an initial algebra. This is useful because we had shown that these fixed points correspond to certain datatypes. It also means that our datatypes receive the same properties as initial algebras.

Then let's rewrite the initiality condition using our datatypes. We call our initial algebra `FixF`, as the fixed point of the pattern functor `F`. Our evaluator function becomes the constructor `in :: F(Fix F)` → `Fix F`. The initiality condition then tells us that there is a unique homomorphism `m` to any other algebra `(a, alg)` over the same pattern functor.

$$\begin{tikzcd}
\mathtt{F\ (Fix\ F)} \arrow[r, "\mathtt{fmap\ m}"] \arrow[d, "\mathtt{in}"] & \mathtt{Fa} \arrow[d, "\mathtt{alg}"] \\
\mathtt{Fix\ F} \arrow[r, "\mathtt{m}"] & \mathtt{a}
\end{tikzcd}$$

Remember how Lambek's theorems told us that `in` is an isomorphism, we shall call the inverse `out`. Accordingly, we can just flip the isomorphism to gain the commuting diagram:

$$\begin{tikzcd}
\mathtt{F\ (Fix\ F)} \arrow[r, "\mathtt{fmap\ m}"] & \mathtt{Fa} \arrow[d, "\mathtt{alg}"] \\
\mathtt{Fix\ F} \arrow[r, "\mathtt{m}"] \arrow[u, "\mathtt{out}"] & \mathtt{a}
\end{tikzcd}$$

When we write down the the commutation condition for this diagram:

$$\mathtt{m} = \mathtt{alg} \circ \mathtt{fmap\ m} \circ \mathtt{out}$$

Notice how m occurs on both sides of the equation. It looks like our construction results in a recursive definition of m. Besides alg is a simple non-recursive function. This is rather powerful because that means we can go to every other algebra (so all other types) using our function m, which is defined using only the variable algebra alg.

Since we can do this for any algebra alg, it makes sense to defines a higher order function takes the algebra as a parameter and gives us the function we called m. This higher order function is called a catamorphism (from the Greek κατά meaning "downwards"):

```
cata :: Functor f => (f a  -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . out
```

Example 23 (Peano Naturals)
The functor that defines the positive natural numbers:

```
data NatF a = oneF | succF a
```

Let us take (Int, Int) as the carrier type and define the algebra:

```
fac :: NatF (Int, Int) -> (Int, Int)
fac oneF = (0,1)
fac (succF (n, f)) = (n+1, f*(n+1))
```

Now you an easily convince yourself that the catamorphism born from this algebra, cata fac calculates the n-th factorial. In general an algebra for the datatype NatF a captures a recurrence relation. It defines the next value in terms of the previous element. A catamorphism then evaluates the n-th element of that sequence.

```
cata fac (in (succF (in (succF (in (succF (in oneF))))))) == (3,6)
```

Example 24 (Lists)
We find the list datatype. if we consider the fixed point of the following functor.

```
data ListF e a = nilF | consF e a
```

Indeed, replacing the variable a with the result of recursion, which we'll call List e, we get:

```
data List e = nil | cons e (List e)
```

An algebra for the list functor, pattern matches on both constructors of the datatype. The algebra defines one step in the recursion, its result for NilF defines how we evaluate an empty list, and its result for ConsF specifies how the current element should be combined with the result of the evaluated tail of list.

As an example, the algebra that calculates the length of a list:

```
lenAlg :: ListF e Int -> Int
lenAlg (consF  e n) = n + 1
lenAlg nilF = 0
```

Indeed, the resulting catamorphism cata lenAlg calculates the length of a list. Here n is the length of the entire tail, and when we return n + 1 we return our new evaluated length (as tail plus one element is one element longer), which is used in the next step of evaluation. Notice how this evaluator lenAlg is a combination of two functions. The first takes two values a list element and the accumulator (nicely packaged in the pair consF e n) and returns a new accumulator. The second defines a starting value.

Compare that to the traditional Haskell definition:

```
length = foldr (\e n -> n + 1) 0
```

We find the components of the algebra represented as the two arguments to foldr. We observe foldr is just a catamorphism specialized to lists.

Anamorphisms

As we know, for every construction in category theory we have one dual to it. The categorical dual to a catamorphism is called an *anamorphism*, from the Greek $\alpha\nu\alpha$ meaning "upwards". As in a anamorphism builds up from a single seed value as opposed to a catamorphism, that builds down the structure into a single value.

The anamorphism is born from the construction dual to an F-algebra, an F-coalgebra. where the direction of morphisms is reversed:

$$\mathtt{a} \to \mathtt{F\ a}$$

Coalgrabras for a given functor together with homomorphism between them also form a category. The homomorphisms now preserve the coalgebraic structure. e.g. the following diagram commutes for the homomorphism $\mathtt{m :: (B, g)} \to \mathtt{(A, f)}$.

$$\begin{tikzcd}
\mathtt{F\,A} & \mathtt{F\,B} \arrow[l, "\mathtt{F\,m}"'] \\
\mathtt{A} \arrow[u, "\mathtt{f}"] & \mathtt{B} \arrow[l, "\mathtt{m}"'] \arrow[u, "\mathtt{g}"']
\end{tikzcd}$$

The terminal object (t, u) in that category is called the terminal (or final) coalgebra. For every other algebra (A, f) there is a unique homomorphism m to the *terminal algebra*. Similarly, to the initial algebra in the category of F-algebras, a terminal coalgeba is a fixed point of the functor, in the sense that the morphism u :: t → F t is an isomorphism (Lambek's theorem for coalagebras). Which leaves us the following commuting diagram – dual to the definition of a catamorphism.

$$\begin{tikzcd}
\mathtt{F\,(Fix\,F)} \arrow[d, "\mathtt{inn}"] & \mathtt{Fa} \arrow[l, "\mathtt{fmap\,m}"'] \\
\mathtt{Fix\,F} & \mathtt{a} \arrow[l, "\mathtt{m}"'] \arrow[u, "\mathtt{coalg}"]
\end{tikzcd}$$

The commuting property of this diagram then leaves us with the following definition for an anamorphism:

```haskell
ana :: Functor f => (a -> f a) -> a -> Fix f
ana coalg = inn . fmap (ana coalg) . coalg
```

A terminal coalgebra is usually interpreted in programming as a recipe for generating (possibly infinite) data structures or transition systems just like a catamorphism can be used to evaluate an initial algebra, an anamorphism can be used to coevaluate a terminal coalgebra.

Example 25 (Streams)
The following functor gives us a fixed point that is an infinite stream of elements of type e:

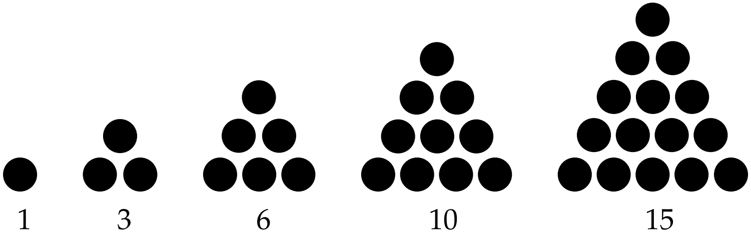
```haskell
data StreamF e a = StreamF e a
    deriving Functor
```

A coalgebra for StreamF e is a function that takes the seed of type a and produces a pair (StreamF is a fancy name for a pair) consisting of an element and the next seed.

A Show instance for our Fix datatype is needed if we wish print out the result.

```
instance (Show (f (Fix f)) ) => Show (Fix f) where
    show (in f) = show f
```

You can easily generate simple examples of coalgebras that produce infinite sequences, such as list of all natural numbers, all primes or all squares. This one for example allows us to create an infinite stream of all triangle numbers.



```
triangle :: Int -> StreamF Int Int
triangle x = streamF (x * (x + 1) `div` 2) (x+1)
```

Generating the numbers becomes the simple call:

```
ana triangle 1
```

Relating base functors and recursive datatypes

The examples above all define a new fixedpoint of some base functor. But in Haskell, a type that is isomorphic to this fixedpoint might already exist. Take for example 24. In Haskell we already have a `List` a datatype `[a]`.

We would much rather use the type that is already present in Haskell. We have to somehow tell the compiler that `[a]` is our fixed point. To this end, we can introduce a `Base` typeclass.

```
class (Functor f) => Base f where
    type Rec f :: *

    inn :: f (Rec f) -> Rec f
    out :: Rec f -> f (Rec f)
```

A functor can be a base for a fixed point. This base corresponds to our pattern functor. So that `cata` and `ana` can be rewritten to:

```
cata :: Base f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . out

ana :: Base f => (a -> f a) -> a -> Rec f
ana coalg = inn . fmap (ana coalg) . coalg
```

We find the following instances:

```
instance Functor (ListF e) where
    fmap f nilF = nilF
    fmap f (ConsF e a) = ConsF e (f a)

instance Base (ListF e) where
    type (Rec (ListF e)) = [e]

    inn nilF       = []
    inn (ConsF e a) = e:a

    out []         = nilF
    out (e:a)    = (ConsF e a)
```

With this in place we can instead call:

```
cata lenAlg [1,2,3,4,5] == 5
```

Program Calculation Laws

Generic catamorphisms and anamorphisms give closed patterns that represent inductive programming techniques and this allows for the generic expression of programming rules. An account of some of these rules is found in [22]. These laws actually birth a complete new style of programming called *the algebra of programming*[4] or *generic programming*[2]. Some of these laws prove to be free-theorems chapter 8. One such rule is the *ana-fusion* law. We use this *fusion* law for anamorphisms as a transformation rule. We can transform the composition of a function with an anamorphism into a single anamorphism so that the intermediate values can be avoided. Or we can use it the other way around to split a function, to allow for subsequent optimizations.

We show how to derive this theorem using the notation of section 8.

```
ana :: Functor f => (a -> f a) -> a -> Fix f
```

We keep in mind that `f` is a functor. Parametricity of this function then gives us:

$$(\texttt{ana},\texttt{ana}) \in \mathcal{T}_{\texttt{forall x,f . (a -> f a) -> a -> Fix f},\emptyset}$$

We start by unfolding the definition, after a few subsequent rewrites we have:

$$\begin{array}{l}
\mathtt{forall}\ \mathcal{A} : \mathtt{A} \Leftrightarrow \mathtt{A}' \\
\quad \mathtt{forall}\ \mathcal{F} : \mathtt{F} \Leftrightarrow \mathtt{F}' \\
\quad\quad (\mathtt{a}, \mathtt{b}) \in \mathcal{T}_{\mathtt{a},\{\mathtt{a}\mapsto\mathcal{A},\mathtt{f}\mapsto\mathcal{F}\}} \\
\quad\quad\quad \Longrightarrow (\mathtt{coalg}_{\mathtt{A},\mathtt{F}}\mathtt{a}, \mathtt{coalg}'_{\mathtt{A}',\mathtt{F}'}\mathtt{b}) \in \mathcal{T}_{\mathtt{f}\ \mathtt{a},\{\mathtt{a}\mapsto\mathcal{A},\mathtt{f}\mapsto\mathcal{F}\}} \\
\quad\quad\quad\quad \Longrightarrow (\mathtt{c}, \mathtt{d}) \in \mathcal{T}_{\mathtt{a},\{\mathtt{a}\mapsto\mathcal{A},\mathtt{f}\mapsto\mathcal{F}\}} \\
\quad\quad\quad\quad\quad \Longrightarrow (\mathtt{ana}_{\mathtt{A},\mathtt{F}}\ \mathtt{coalg}_{\mathtt{A},\mathtt{F}}\mathtt{c}, \mathtt{ana}_{\mathtt{A}',\mathtt{F}'}\ \mathtt{coalg}'_{\mathtt{A}',\mathtt{F}'}\mathtt{d}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \in \mathcal{T}_{\mathtt{Fix}\ \mathtt{f},\{\mathtt{a}\mapsto\mathcal{A},\mathtt{f}\mapsto\mathcal{F}\}}
\end{array}$$

We instantiate $\mathcal{A}$ to the graph of function `f`. And $\mathcal{F}$ to the identity function on functor `F` (recall how `f` is a functor!). We specify it to the identity function since we want both sides of the equation to be equal, and for that to happen we need that the return types on both sides are equal. The return type in this case is the fixed point of our polymorphic functor type `f`.

$$\begin{array}{l}
\mathtt{forall}\ \mathtt{graph(f)} : \mathtt{A} \Leftrightarrow \mathtt{A}' \\
\quad \mathtt{forall}\ \mathtt{id}_{\mathtt{F}} : \mathtt{F} \Leftrightarrow \mathtt{F} \\
\quad\quad (\mathtt{a}, \mathtt{b}) \in \mathcal{T}_{\mathtt{a},\{\mathtt{a}\mapsto\mathtt{graph(f)},\mathtt{f}\mapsto\mathtt{id}_{\mathtt{F}}\}} \\
\quad\quad\quad \Longrightarrow (\mathtt{coalg}_{\mathtt{A},\mathtt{F}}\mathtt{a}, \mathtt{coalg}'_{\mathtt{A}',\mathtt{F}}\mathtt{b}) \in \mathcal{T}_{\mathtt{f}\ \mathtt{a},\{\mathtt{a}\mapsto\mathtt{graph(f)},\mathtt{f}\mapsto\mathtt{id}_{\mathtt{F}}\}} \\
\quad\quad\quad\quad \Longrightarrow (\mathtt{c}, \mathtt{d}) \in \mathcal{T}_{\mathtt{a},\{\mathtt{a}\mapsto\mathtt{graph(f)},\mathtt{f}\mapsto\mathtt{id}_{\mathtt{F}}\}} \\
\quad\quad\quad\quad\quad \Longrightarrow (\mathtt{ana}_{\mathtt{A},\mathtt{F}}\ \mathtt{coalg}_{\mathtt{A},\mathtt{F}}\mathtt{c}, \mathtt{ana}_{\mathtt{A}',\mathtt{F}}\ \mathtt{coalg}'_{\mathtt{A}',\mathtt{F}}\mathtt{d}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \in \mathcal{T}_{\mathtt{Fix}\ \mathtt{f},\{\mathtt{a}\mapsto\mathtt{graph(f)},\mathtt{f}\mapsto\mathtt{id}_{\mathtt{F}}\}}
\end{array}$$

Which states the following in a programming way:

$$\begin{array}{l}
\mathtt{forall}\ \mathtt{f} : \mathtt{A\text{-}>}\ \mathtt{A}' \\
\quad \mathtt{forall}\ \mathtt{Functor}\ \mathtt{F} \\
\quad\quad \mathtt{f\ a\ ==\ b} \\
\quad\quad\quad \Longrightarrow \mathtt{fmap\ f\ (coalg}_{\mathtt{A},\mathtt{F}}\ \mathtt{a)\ ==\ coalg'}_{\mathtt{A}',\mathtt{F}}\ \mathtt{b} \\
\quad\quad\quad\quad \Longrightarrow \mathtt{f\ c\ ==\ d} \\
\quad\quad\quad\quad\quad \Longrightarrow \mathtt{ana}_{\mathtt{A},\mathtt{F}}\ \mathtt{coalg}_{\mathtt{A},\mathtt{F}}\ \mathtt{c\ ==\ ana}_{\mathtt{A}',\mathtt{F}'}\ \mathtt{coalg'}_{\mathtt{A}',\mathtt{F}}\ \mathtt{d}
\end{array}$$

$$\Leftrightarrow$$

$$\begin{aligned}&\texttt{forall f : A- > A}'\\ &\quad\texttt{forall Functor F}\\ &\qquad\Longrightarrow \texttt{fmap f (coalg}_{\texttt{A,F}}\texttt{ a) == coalg'}_{\texttt{A}',\texttt{F}}\texttt{ (f a)}\\ &\qquad\quad\Longrightarrow \texttt{ana}_{\texttt{A,F}}\texttt{ coalg}_{\texttt{A,F}}\texttt{ c == ana}_{\texttt{A}',\texttt{F}'}\texttt{ coalg'}_{\texttt{A}',\texttt{F}}\texttt{ (f c))}\end{aligned}$$

This gives us an equality such that we can fuse a composition of `f` with an anamorphism into a single anamorphism. That sounds like the ana-fusion rule we are trying to derive. Only there is an added requirement before we can use this rewrite rule.

And so, forall $\texttt{f : A- > A}'$, and `F`-coalgebras `(A, coalg)` and $\texttt{(A}', \texttt{coalg}'\texttt{)}$:

$$\begin{aligned}&\texttt{(fmap f)} \circ \texttt{coalg == coalg'} \circ \texttt{f}\\ &\quad\Longrightarrow \texttt{ana coalg == (ana coalg')} \circ \texttt{f}\end{aligned}$$

Note how we removed the subscript for `coalg`. It is clear from context what function this should be as being a `F`-coalgebra, specifies the polymorphic function to `F`. And the carrier type indicated in `(A, coalg)` specifies the polymorphic function `coalg` to `A`. Likewise we know the instantiation for $\texttt{coalg}'$. Similarly we find that the subscript for `ana` is redundant.

Let us put our new-found ana-fusion rule to use:

Example 26 (Roman Numerals)
The classical Romans write their numbers using the following symbols: M, D, C, L, X, V and I with the respective values of 1000, 500, 100, 50, 5 and 1. A positive integer is denoted by writing these symbols consecutively, where the highest value symbols are on the left, and the lowest value symbols are on the right. The value of the number is the sum of the individual symbols. Negative values and zero can't be represented.

Suppose we would like a function that converts a number in base 10, to its equivalent Roman version. First, we need a way to represent these Roman numerals. We will use an enumeration for the Roman numeral symbols, and a list-like datatype for the entire numbers.

```haskell
data Roman = I | V | X | L | C | D | M
    deriving(Show)

data RomanNum = zero | cons Roman RomanNum
    deriving(Show)
```

As we see `RomanNum` is a recursive datatype with the pattern functor:

```
data RomanNumF a = zeroF | consF Roman a
```

Assume the proper instances for the `Base` and `Functor` type classes are provided. Clearly, the base conversion function we want to make is a function `Int -> RomanNum`. We know the positive integers and the roman numerals represent the same thing. So it would be rather logical if there was a homomorphism from the positive integers to the Roman numerals.

```
roman :: Int -> RomanNumF Int
roman x
    | x >= 1000   = consF M (x-100)
    | x >= 500    = consF D (x-500)
    | x >= 100    = consF C (x-100)
    | x >= 50     = consF L (x-50)
    | x >= 10     = consF X (x-10)
    | x >= 5      = consF V (x-5)
    | x >= 1      = consF I (x-1)
    | otherwise   = zeroF
```

We can use `ana roman` to convert integers to their roman representation. But it is not yet perfect, as Romans used abbreviations for often occurring patterns.

$$\begin{array}{rrr} \texttt{DCCCC} \mapsto \texttt{CM} & \texttt{LXXXX} \mapsto \texttt{XC} & \texttt{VIIII} \mapsto \texttt{IX} \\ \texttt{CCCC} \mapsto \texttt{CD} & \texttt{XXXX} \mapsto \texttt{XL} & \texttt{IIII} \mapsto \texttt{IV} \end{array}$$

This reducing can again be captured by an anamorphism.

```
abbrev :: RomanNum -> RomanNumF RomanNum
abbrev zeroF = zeroF
abbrev (cons x1 (cons x2 (cons x3 (cons x4 (cons x5 ys)))))
    | [x1,x2,x3,x4,x5] == [D,C,C,C,C] = consF C (cons M ys)
    | [x1,x2,x3,x4,x5] == [L,X,X,X,X] = consF X (cons C ys)
    | [x1,x2,x3,x4,x5] == [V,I,I,I,I] = consF I (cons X ys)
abbrev (cons x1 (cons x2 (cons x3 (cons x4 ys))))
    | [x1,x2,x3,x4] == [C,C,C,C] = consF C (cons D ys)
    | [x1,x2,x3,x4] == [X,X,X,X] = consF X (cons L ys)
    | [x1,x2,x3,x4] == [I,I,I,I] = consF I (cons V ys)
abbrev (cons x y) = consF x y
```

So that we have `ana abbrev ∘ ana roman` which gives us the proper Roman number representation. Though it this can be inefficient since, computationally, we first make a large list of Roman letters and then abbreviate it. It, however, would be more efficient if we directly incorporated the abbreviation step whilst converting to Roman numerals.

To this end, we could try and derive a more convenient form. Using the *ana-fusion* law:

forall `f : A- > A'`, and `F`-coalgebras `(A, coalg)` and `(A', coalg')`:

$$\begin{aligned} &\texttt{(fmap f)} \circ \texttt{coalg == coalg'} \circ \texttt{f} \\ &\quad\Longrightarrow \texttt{ana coalg == (ana coalg')} \circ \texttt{f} \end{aligned}$$

In this case, we can apply the law with the following substitutions

$$\begin{aligned} &\texttt{(fmap (ana roman))} \circ \texttt{f == abbrev} \circ \texttt{ana roman} \\ &\quad\Longrightarrow \texttt{ana f == (ana abbrev)} \circ \texttt{(ana roman)} \end{aligned}$$

The antecedent `(fmap (ana roman)) ∘ f == abbrev ∘ ana roman` then becomes a specification for our new function `f`. We just need to solve this equation for `f`.

A direct insight we gain is that `f` is of type `Int -> RomanNumF Int`. Other than that `f` needs to already incorporate the abbreviations, as `fmap (ana roman)` only works on the `Int` part of the `RomanNumF Int` datatype. The already chosen Roman letter accompanying should already be the first part of the abbreviation. Now we still need the function to also properly chose the rest of the abbreviation, and the only way we can influence the next step is by changing the seed `Int` value.

We observe that adding a roman symbol in front of a roman numeral reduces its numerical value by that amount. If we then want to get the original amount we have to add the negated value to the next seed. This leaves us with the function `romanAbbrev` as below.

If we take `f` to be `romanAbbrev` as below, the left side of the implication holds (for all positive integers). The proof is easy but tedious. It is a induction proof on `x`, and requires 7 different case distinctions, but other than that is just a lot of definition unfolding.

```
romanAbbrev :: Int -> RomanNumF Int
romanAbbrev x
    | 900 <= x && x < 1000 = consF C (x+100)
    | 400 <= x && x < 500  = consF C (x+100)
    | 90  <= x && x < 100  = consF X (x+10)
    | 40  <= x && x < 50   = consF X (x+10)
    | 9   <= x && x < 10   = consF I (x+1)
    | 4   <= x && x < 5    = consF I (x+1)
    | otherwise = roman x
```

And so `ana romanAbbrev` is our final integer to roman function. Now `ana romanAbbrev` is both faster and uses less memory than its counterpart `ana abbrev` ∘ `ana roman`.

10 Monads and Effects

Pure lazy functional languages, such as Haskell offer the power of lazy evaluation and the simplicity of equational reasoning. Pure languages make changes in code easy, by making the types which operations depend on explicit. Though at times, this restructuring, even for a seemingly small change can become extensive, meanwhile using an impure feature might reduce this restructuring to a few lines. Impure languages offer tempting features such as state, exception handling or continuations – all powerful tools in the hands of a programmer. This offer of flexibility and expressive power is hard to pass away, a language not offering features like these, could be considered crippled compared to those that do.

If for example, we wanted to add error handling to a program made in a pure language, we would need to change the result type to include error values, and at every call we would have to check for and handle these errors appropriately. If instead, we had used an impure language with exception handling, there would have been no need for restructuring at all.

Without a special construct, cue the monad, these pure languages simply don't offer the flexibility needed for a pleasant coding-experience.

The Problem

This is not the only problem programmers deal with regularly. Amongst others, here is a short listing of similar problems taken from [3], where we usually give up on the purity of our functions:

1. Partiality
2. Nondeterminism
3. Side effects
4. Exceptions
5. IO

What is astonishing is that all these problems are somehow alike. This insight was first discovered by Eugenio Moggi [24].

The Solution

Pure functions take a value and return a value. What happens whilst executing the function shouldn't affect the outcome or do something odd/extra. We should be able to memoize the function. We find that during the execution of a pure function we can't just throw an exception or write to a log. As this would do something different than just taking a value and returning a value.

Somehow we want our effect to be represented in the return value of the function. We call these functions with added effects *computations* [24]. These computations are entirely different with each different effect. And you have to realize that, at this stage, these functions do nothing special. It's only when we insist on decomposition – being able to decompose a single computation into smaller computations – and to then compose them afterwards, that we need something called a monad.

You see, these computations are useless if we can't chain them together. Consider two functions, one that also has the added effect of printing "Hello" and another that has the added effect of printing "World!". We would appreciate it if our final program would be able to print "Hello World!". Though this is only possible if we could combine these output statements, and so we insist on composition.

I've said that all these semantics of all these varying effects can be captured by the structure of a monad! But what is a monad?

Yeah! What is a monad?

The concept of a monad comes from category theory. But don't bother completely understanding its definition right away. How we come to this definition is explained after introducing Kleisli Categories. We start with a definition as given in [38].

For all our purposes:

> A **monad** is a triple (`M`, `unitM`, `BindM`) consisting of a type constructor `M` and a pair of polymorphic functions:
>
> ```
> unitM :: a -> M a
> bindM :: M a -> (a -> M b) -> M b
> ```
>
> These functions must satisfy three laws, namely the laws of associativity, left identity and right identity, which are discussed later in this chapter.

The basic idea in converting a program to monadic form is this: a function of type `a -> b` is converted to one of type `a -> M b` (we embellish the function with `M`). Thus in a program where we would print to a log, functions have type `a -> (b, String)` rather than `a -> b`. The identity function has type `a -> a`. The corresponding function in monad form is `unitM`, which has type `a -> (a, String)`. It takes a value into its corresponding representation in the monad. In this example, the endofunctor was of course `M a = (a, String)`.

Kleisli Category

A Kleisli category is a category naturally associated to a monad. We will use this category as an aid to understanding the monad.

If we insist on being able to compose the computations, we find ourselves a category, a Kleisli category, where arrows between `A` and `B` are arrows from `A` to `M B` (Our computations!) in the original category **C**. Our insistence on being able to compose these arrows translates to the composition operator of this new category. In Haskell this composition takes a funny form, known as the fish operator:

```haskell
(>=>) :: (a -> M b) -> (b -> M c) -> (a -> M c)
```

If `f :: A` → `M B` and `g :: B` → `M C`, then in the Kleisli category:

$$\begin{tikzcd} A \arrow[r, "g"] \arrow[rr, bend right, "g>=>f"'] & B \arrow[r, "f"] & C \end{tikzcd}$$

Mind you that this fish operator is different for every Kleisli category. Meaning different for every computation.

If we want our computations to form a proper category. The fish operator will need to be associative and have an identity arrow of the form `A` → `M A`. Consequently, we can only form a Kleisli category for those computations that have an associative composition and an identity computation.

For our limited purposes, a Kleisli Category has, as objects, the objects of the underlying category **C** (in our case the types of the programming language). Morphisms from object `A` to object `B` are morphisms in **C** from `A` to an object derived from `B` using the particular embellishment (our computations, the endofunctor `M`).

Example 27 (Kleisli Category for the Writer monad)
How would this Kleisli category look for a chosen computation, say a computation that creates a log? Let us start by defining a type that could carry a logging string with it, the `Writer` type:

```haskell
type Writer a = (a, String)
```

Our morphisms are functions from an arbitrary type to some Writer type:

```
a -> Writer A
```

The String part is the text that is going to be logged.
We declare the composition:

```
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
m1 >=> m2 = \x ->
    let (y, s1) = m1 x
        (z, s2) = m2 y
    in (z, s1 ++ s2)
```

In the composition we execute the first computation. Then extract the first component of the resulting pair and pass it to the next computation. We concatenate the logs of both computations. Lastly, we combine the results into a value-log pair.

I will also define the identity morphisms for our Kleisli category:

```
unitWriter   :: a -> Writer a
unitWriter x = (x, "")
```

A log can be useful to track a lots of things, a prominent example in computer science is logging function calls, to get a trace, so that we can later debug our code.

Two morphisms existing in this category would be:

```
upCase :: String -> Writer String
upCase s = (map toUpper s, "upCase")

toWords :: String -> Writer [String]
toWords s = (words s, "toWords")
```

Finally, the composition of the two functions is accomplished with the help of the fish operator:

```
process :: String -> Writer [String]
process = upCase >=> toWords
```

This function takes a string, and capitalizes every letter, furthermore it splits everything in separate words. But what is special is that it logs what it has done. It passes the log implicitly using our fish operator, we don't have to deal with passing the log from function to function, that

> is done for us. Alternatively we could have used an impure global log, but this implementation retains it purity and thus allows equational reasoning and lazy evaluation.

So what did we just all do? We took our original model of types to a new extended category. One which shares the same objects, but where arrows are embellished and where their composition does something more than just chain output to input. Instead of having the composition laid out for us, we can now experiment with composition. It turns out that this additional choice of composition is precisely what we need to give simple denotational semantics to programs using side effects.

So what does a monad have to do with this Kleisli category?

To a Kleisli Category, the monad is the glue that makes it all work out and makes it a proper category. Say we have two arrows in the Kleisli Category from $\mathtt{A} \to \mathtt{B}$, and $\mathtt{B} \to \mathtt{C}$, then we can't just willy-nilly compose these, as they are arrows $\mathtt{A} \to \mathtt{M\,B}$ and $\mathtt{B} \to \mathtt{M\,C}$ in the original category. And the target of the first is different than the source of the latter.

$$\begin{tikzcd} \mathtt{A} \arrow[r] & \mathtt{M\,B} \arrow[r, phantom, "\neq"] & \mathtt{C} \arrow[r] & \mathtt{M\,C} \end{tikzcd}$$

What we can do is to apply $\mathtt{B} \to \mathtt{M\,C}$ to the $\mathtt{B}$ "within" $\mathtt{M\,B}$. To give us an arrow in C $\mathtt{A} \to \mathtt{M(M\,C)}$. We can do this because $\mathtt{M}$ is an endofunctor.

$$\begin{tikzcd} \mathtt{A} \arrow[r] & \mathtt{M\,B} \arrow[r] & \mathtt{M(M\,C)} \end{tikzcd}$$

Then we are almost there as we only have to somehow join those two $\mathtt{M}$'s together, and we get an arrow in our Kleisli category. Our glue, the monad, provides this joining mechanism. The monad also provides the identity arrow.

A **Kleisli triple** – or a monad in extension form – over a category **C** is a triple $(\mathtt{M}, \mathtt{unitM}, \mathtt{bindM})$, where

$$\begin{aligned} \mathtt{M} &:: \mathtt{Obj(C)} \to \mathtt{Obj(C)} \\ \mathtt{unitM}_{\mathtt{A}} &:: \mathtt{A} \to \mathtt{M\,A} \text{ for } \mathtt{A} \in \mathtt{Obj(C)} \\ \mathtt{bindM} &:: \mathtt{M\,A} \to (\mathtt{A} \to \mathtt{M\,B}) \to \mathtt{M\,B} \end{aligned}$$

bindM, in Haskell is given the infix notation >>=.

and the following operations hold:

```
unitM a >>= k                  =  k a -- left identity
m       >>= unitM              =  m   -- right identity
m       >>= (\x -> k x >>= h)  =  (m >>= k) >>= h --associative
```

These “monad laws” actually ensure that the corresponding Kleisli Category is actually a category. A Kleisli category is a category based on a monad. Using our new glue we can define our Kleisli category properly:

> Given a Kleisli triple (M, unitM, BindM) over C, the **Kleisli category** C_M is defined as follows:
>
> 1. the objects of C_M are those of C
> 2. the hom-set $C_M(A, B)$ of morphisms from A to B in C_M is C(A, M B)
> 3. the identity on A in C_M is $\mathtt{UnitM}_A :: A \to MA$
> 4. composition in a Kleisli category is given by the fish operator:
>
> ```
> (>=>)   :: (A -> M B) -> (B -> M C) -> A -> M C
> f >=> g = \a -> bindM (f a) g
> ```

It is natural to take $\mathtt{UnitM}_A$ as the identity on A in the category C_M since it maps x to x viewed as a computation.

Now the axioms for a Kleisli triple are equivalent to the axiom laws of a category, namely identity and associativity.

Fish Dissection

We take another look at the silly fish operator. We put on our programmer goggles, and take on a programmers viewpoint of this operator.

When implementing the fish operator of different monads you quickly realize that a lot of code is repeated and can be easily factored out. As a starting point, the Kleisli composition always returns a function that takes a single value, we could just as well start our fish with a lambda:

```
(>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
f >=> g = \a -> ...
```

And the only thing we can do with a is to pass it to our first function argument f.

```
f >=> g = \a -> let mb = f a
                in ...
```

So what do we have now? We still need to do something with the function g, which takes something of value b to mc and the only thing we have is `mb :: m b`. That should ring a bell. We only need our `bind` to turn this `m b` together with a function `g :: b -> m c` into a single value `m c`.

```
(>>=) :: m a -> (a -> m b) -> m b
```

Instead of defining the fish operator, we may instead define bind. That isn't so shocking, we already knew this, but it is interesting that we come to the same conclusion so easily, by just looking at the type of `(>=>)`.

Okay, but what does bind do? Can we dissect that too? If we take advantage of the fact that `m` is a type constructor (a functor) we can use `fmap` to apply `g` to the `b` "within" `m b` resulting in a value of type `m (m c)`. All we need now is a function that collapses or flattens the double application of `m`. Such a function is called `join`.

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

using `join`, we can rewrite bind as:

```
ma >>= f = join (fmap f ma)
```

This lead us to the third option for defining a monad:

```
class Functor m => Monad m where
    join   :: m (m a) -> m a
    return :: a -> m a
```

Here we have explicitly requested that `m` is a functor, we didn't have to do that in the previous two definitions of the monad. That's because any type constructor `m` that either support the fish or bind operator is automatically a functor. For instance, it's possible to define `fmap` in therms of `bind` and `return`.

```
fmap f ma = ma >>= \a -> return (f a)
```

Monads Categorically

This exact observation that the fish operator can be replaced by `join` can also be described categorically. Let's get to the maths, immediately it becomes confusing. Mathematicians use different notation than programmers. We will adopt this different notation, as it is more common. Instead of using `M` (for monad) mathematicians prefer to use the letter `T` for the endofunctor and the greek letters: μ for `join` and η for `return`. Both `join` and `return` are parametric polymorphic functions, so we know that these are natural transformations. This is also the take of the categorical definition.

Therefore, in category theory, a monad is defined as an endofunctor `T` equipped with a pair of natural transformations μ and η.

A **monad** – in monoid form – over $\mathtt{C}$ is a triple $(\mathtt{T}, \eta, \mu)$, where $\mathtt{T} : \mathtt{C} \to \mathtt{C}$ is an endofunctor, $\eta :: \mathtt{I} \to \mathtt{T}$ and $\mu :: \mathtt{T} \circ \mathtt{T} \to \mathtt{T}$ are natural transformations and the following diagrams commute:

$$\begin{tikzcd}
\mathtt{T}^3 \arrow[r, "\mathtt{T}\circ\mu"] \arrow[d, "\mu\circ\mathtt{T}"'] & \mathtt{T}^2 \arrow[d, "\mu"] \\
\mathtt{T}^2 \arrow[r, "\mu"'] & \mathtt{T}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathtt{I}\circ\mathtt{T} \arrow[r, "\eta\circ\mathtt{T}"] \arrow[dr, equal] & \mathtt{T}^2 \arrow[d, "\mu"] & \mathtt{T}\circ\mathtt{I} \arrow[l, "\mathtt{T}\circ\eta"'] \arrow[dl, equal] \\
& \mathtt{T} &
\end{tikzcd}$$

Remark 15 (Notation $\mathtt{T} \circ \mu$)
You might look at these diagrams and think "That doesn't make any sense, why would we compose a functor with a natural transformation?!" and you would be completely right – that doesn't make any sense. The notation $\mathtt{T} \circ \mu$ is actually an abbreviation for the horizontal composition $\mathtt{I}_\mathtt{T} \circ \mu$, where $\mathtt{I}_\mathtt{T}$ is the identity natural transformation. However, the notation is unambiguous, exactly because it doesn't make any sense: the only option it could be in this context is the identity natural transformation at component $\mathtt{T}$.

$$\begin{tikzcd}
\bullet \arrow[r, bend left=60, "\mathtt{T}^2"{name=A}] \arrow[r, bend right=60, "\mathtt{T}"'{name=B}] \arrow[Rightarrow, from=A, to=B, "\mu"] & \bullet \arrow[r, bend left=60, "\mathtt{T}"{name=C}] \arrow[r, bend right=60, "\mathtt{T}"'{name=D}] \arrow[Rightarrow, from=C, to=D, "\mathtt{I}_\mathtt{T}"] & \bullet \arrow[r, phantom, "\mapsto"] & \bullet \arrow[r, bend left=60, "\mathtt{T}^3"{name=E}] \arrow[r, bend right=60, "\mathtt{T}^2"'{name=F}] \arrow[Rightarrow, from=E, to=F, "\mathtt{I}_\mathtt{T}\circ\mu"] & \bullet
\end{tikzcd}$$

Now, these definitions specify the same and therefore are isomorphic, there is a bijection between Kleisli triples and monads:

$\Rightarrow$ Given a Kleisli triple $(\mathtt{T}, \eta, \mathtt{bindM})$ the correspond monad is $(\mathtt{T}, \eta, \mu)$, where $\mathtt{T}$ is the extension of the function $\mathtt{T}$ to the endofunctor $\mathtt{T}$ by taking `(T f) x = bindM x` $(\eta\circ$ `f)` for `f :: a -> b` and $\mu_\mathtt{A}$ `x == bindM x` $\mathtt{id}_\mathtt{TA}$.

$\Leftarrow$ Conversely, given a monad $(\mathtt{T}, \eta, \mu)$, the corresponding Kleisli triple is $(\mathtt{T}, \eta, \mathtt{bindM})$, where $\mathtt{T}$ is the restriction of the functor $\mathtt{T}$ to objects, and `bindM x f =` $(\mu_\mathtt{B} \circ$ `(T f)) x` for `f :: A -> T B`

Roughly speaking, the associativity law states that the two ways of reducing $\mathtt{T}^3$ down to $\mathtt{T}$ must give the same result. Two unit laws (left and right) state that when η is applied to `T` and then reduced by μ, we get back `T`.

Previously the requirement – that the composition of Kleisli arrows is associative and that $\eta_{\mathtt{a}}$ is the identity Kleisli arrow at `a` – translated to the monadic laws for μ and η. But this new definition makes them look more like monoid laws. In fact, μ – pronounced *mu* – is often called *mu*ltiplication, and η – pronounced *êta* sometimes *ita* – is called un*it*. So we have multiplication μ, unit η, associativity and unit laws. That indeed sounds quite like a monoid, but our definition of a monoid is too narrow to describe a monad as a monoid so let us generalize the notion of a monoid.

Monoidal Categories

We all know what a monoid is. It is a set with a binary operation and a special element called unit! Where the binary operation must be associative and unital with regard to that neutral element. Moreover, we had already made a categoric description of a monoid as a single-object category where elements are represented by endomorphisms $(\mathtt{m} \to \mathtt{m})$.

In a way we associated to every element a function from $(\mathtt{m} \to \mathtt{m})$. In Haskell this is encoded into the type of `mappend`.

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

This curried version of `mappend` can be interpreted as mapping every element of `m` to a function:

$$\mathtt{mappend} :: \mathtt{m} \to (\mathtt{m} \to \mathtt{m})$$

It's this interpretation that gives rise to the definition of a monoid as a single-object category. But because currying is built into Haskell, we could as well have started with the uncurried definition of multiplication:

$$\mathtt{mappend} :: (\mathtt{m}, \mathtt{m}) \to \mathtt{m}$$

And see how this shift in perspective leads to another arrow-theoretic definition of the monoid. Our new perspective on the `mappend` function suggests a different path to generalization: replacing the Cartesian product with the categorical product. We could start with a category where products are globally defined, pick an object `m` there, and define multiplication as a morphism:

$$\mu :: \mathtt{m} \times \mathtt{m} \to \mathtt{m}$$

Okay, that is one part. Now for η, the unit, we want to select a single element. However as category theorist we don't like talking about elements of an object (we are not allowed to!). Luckily, we can use our trick of using a morphism from the terminal object `i` to select our single element.

$$\eta :: \mathtt{i} \to \mathtt{m}$$

Unlike in the previous definition of a monoid as a single-object category, monoidal laws here are not automatically satisfied – we have to impose them. But to formulate them, we have to establish the monoidal structure of the underlying categorical product itself.

E.g. we need to formulate diagrams that specify the monoidal behaviour of the product. (associativity and identity)

We start with associative property. Associativity of the monoid states that to order in which we apply μ does not matter. We have two ways of reducing $\mathtt{M} \times \mathtt{M} \times \mathtt{M}$. Note that since the Cartesian product is a bifunctor we can lift the morphisms `id` and μ.

$$\begin{tikzcd}
\mathtt{M} \times (\mathtt{M} \times \mathtt{M}) \arrow[r, "\mathtt{id}\times\mu"] & \mathtt{M} \times \mathtt{M} \arrow[r, "\mu"] & \mathtt{M}
\end{tikzcd}$$

$$\begin{tikzcd}
(\mathtt{M} \times \mathtt{M}) \times \mathtt{M} \arrow[r, "\mu\times\mathtt{id}"] & \mathtt{M} \times \mathtt{M} \arrow[r, "\mu"] & \mathtt{M}
\end{tikzcd}$$

Ideally we would like to say: "These two path are the same!" and be done with it. But there is a slight problem: $\mathtt{M} \times (\mathtt{M} \times \mathtt{M})$ is not the same as $(\mathtt{M} \times \mathtt{M}) \times \mathtt{M}$. They, however, are isomorphic. There is a natural isomorphism called the *associator* α that converts between them. With the help of the associator we can create the following commuting diagram that captures the associative property:

$$\begin{tikzcd}
(\mathtt{M} \times \mathtt{M}) \times \mathtt{M} \arrow[rr, "\alpha"] \arrow[d, "\mu\times\mathtt{id}"] & & \mathtt{M} \times (\mathtt{M} \times \mathtt{M}) \arrow[d, "\mathtt{id}\times\mu"] \\
\mathtt{M} \times \mathtt{M} \arrow[dr, "\mu"'] & & \mathtt{M} \times \mathtt{M} \arrow[dl, "\mu"] \\
& \mathtt{M} &
\end{tikzcd}$$

Likewise, we can establish a diagram for the left and right identity properties. Where I is the neutral element, and λ and ρ are two natural isomorphisms called, respectively, the left and right *unitor*.

$$\begin{tikzcd}
\mathtt{I} \times \mathtt{M} \arrow[r, "\eta\times\mathtt{id}"] \arrow[dr, "\lambda"'] & \mathtt{M} \times \mathtt{M} \arrow[d, "\mu"] & \mathtt{M} \times \mathtt{I} \arrow[l, "\mathtt{id}\times\eta"'] \arrow[dl, "\rho"] \\
& \mathtt{M} &
\end{tikzcd}$$

With this new construction we can define a monoid on top of any category with products and a terminal object. As long as we can pick an object m and two morphisms μ and η that satisfy the diagrams for some α, λandρ, we have a monoid.

We could stop here and pretend this version is all good and fine. But it isn't; at least not for our purpose. If we want to give claim to the similarity between a monad and a monoid we need to be able to say that endofunctor composition acts monoidal. But endofunctor composition is not a categorical product! The problem is that our current concept of a monoid on a category is too specific. We use products, but don't call on all properties of the categorical product, indeed we only needed associativity of the product, not the fact that it has projections. Similarly, for the terminal object, we never used its terminal property. We only needed that it worked as a neutral element with regard to the product.

So instead of using the categorical product, we can use something that behaves like a product, but isn't actually one. This is called a *tensor product* – the definition of which is beyond the scope of this paper. But we only need to know that it is associative up to isomorphism and doesn't give us all the extra baggage of a product. In addition, it is also a bifunctor, because we need it to lift morphisms.

With this we can define a monoidal category.

> A **monoidal category** is a category **C** together with a tensor product $\otimes$ :: $C \times C \to C$ and a unit object i, together with three natural isomorphisms called, respectively, the associator and the left and right unitors:

$$\begin{aligned}\alpha_{abc} &:: (a \otimes b) \otimes c \to a \otimes (b \otimes c)\\ \lambda_a &:: i \otimes a \to a\\ \rho_a &:: a \otimes i \to a\end{aligned}$$

Then we can define a monoid in a more general setting of a monoidal category (instead of a category with products and a terminal object). It is just a straightforward generalization of our previous results.

$$\begin{tikzcd}
(M \otimes M) \otimes M \arrow[rr, "\alpha"] \arrow[d, "\mu\otimes id"] & & M \otimes (M \otimes M) \arrow[d, "id\otimes\mu"] \\
M \otimes M \arrow[dr, "\mu"'] & & M \otimes M \arrow[dl, "\mu"] \\
& M &
\end{tikzcd}$$

$$\begin{tikzcd}
I \otimes M \arrow[r, "\eta\otimes id"] \arrow[dr, "\lambda"'] & M \otimes M \arrow[d, "\mu"] & M \otimes I \arrow[l, "id\otimes\eta"'] \arrow[dl, "\rho"] \\
& M &
\end{tikzcd}$$

Monads as Monoids

Anyway, back to our our original goal. We wanted to show the similarity between a monoid and a monad, in fact a monad is a monoid.

If we wish to define a monad as a monoid using our above definition, we first need to establish that it lives within a monoidal category. So we need to find a good candidate for a tensor product in the category of endofunctors.

To be a tensor product we first have to establish that the candidate is a bifunctor. A bifunctor in the category of endofunctors would take two morphisms – here natural transformations – and give back a natural transformation from the tensor product of their sources to the tensor product of their targets.

$$\otimes :: (\mathtt{a} \to \mathtt{b}) \to (\mathtt{c} \to \mathtt{d}) \to (\mathtt{a} \otimes \mathtt{c} \to \mathtt{b} \otimes \mathtt{d})$$

If we were to check if endofunctor composition fits this pattern, we would find the following by replacing objects by endofunctors, arrows by natural transformations, and the tensor product by composition:

$$\circ :: (\mathtt{F} \to \mathtt{F}') \to (\mathtt{G} \to \mathtt{G}') \to (\mathtt{F} \circ \mathtt{G} \to \mathtt{F}' \circ \mathtt{G}')$$

Which you my recognize as the special case of horizontal composition. And as we know, horizontal composition of natural transformations respects identity and associativity, and so we conclude that endofunctor composition forms a bifunctor.

Secondly, we need to find a unit object with regard to that tensor product. Endofunctor composition has a clear identity, namely, the identity functor $\mathtt{Id}_{\mathtt{C}}$. The third condition that a tensor product is associative up to isomorphism is fulfilled by endofunctor composition. To boot, associativity and identity of endofunctor composition are both strict! Meaning that for α, λ and ρ we can use the identity natural transformation `id`.

We now have that category of endofunctors of **C**, together with functor composition as tensor product and the identity functor as unit forms a monoidal category. Then what is a monoid in this category? It's an object – that is an endofunctor `T`; and two morphisms – that is natural transformations:

$$\mu :: \mathtt{T} \circ \mathtt{T} \to \mathtt{T}$$

$$\eta :: \mathtt{I} \to \mathtt{T}$$

Not only that, α, λ and ρ are the identity natural transformations, consequently the monoid laws shrink into these diagrams:

$$\begin{tikzcd}
\mathtt{T}^3 \arrow[r, "\mathtt{T}\circ\mu"] \arrow[d, "\mu\circ\mathtt{T}"'] & \mathtt{T}^2 \arrow[d, "\mu"] \\
\mathtt{T}^2 \arrow[r, "\mu"'] & \mathtt{T}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathtt{I}\circ\mathtt{T} \arrow[r, "\eta\circ\mathtt{T}"] \arrow[dr, equal] & \mathtt{T}^2 \arrow[d, "\mu"] & \mathtt{T}\circ\mathtt{I} \arrow[l, "\mathtt{T}\circ\eta"'] \arrow[dl, equal] \\
& \mathtt{T} &
\end{tikzcd}$$

They are exactly the monad laws we've seen before.

> " All told, a monad in X is just a monoid in the category of endofunctors of X " (Saunders Mac Lane [17, p. 137])

Effects

Our new slogan, that Saunders Mac Lane so kindly provided, gives us a rather compact definition of a monad. Moreover, this definition allows for easy identification of monads. It's exactly those endofunctors that behave monoidal and in programming languages, our endofunctors are type constructors.

The only thing left on this topic, now that we know what a monad is and does, is ask ourselves why these monads are so important in functional programming. We've seen our one example using the `Writer` monad. Furthermore, in the introduction of this section, I gave a small list of similar problems. Let's go through the list and identify the embellishment that applies to each problem in turn.

We shall, however, not give directions on how to use these. Readers that wish for a tutorial could refer to *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell* [26].

Partiality

Whilst coding we occasionally deal with functions that might not terminate. In a lazy language like Haskell, such never-ending functions may, nevertheless, return a value and this value can be passed down to following functions. The value that is passed along represents that the function does not terminate and is called "bottom". This value is of course not usable, but on the condition that we don't explicitly try to use it, our program will continue to run. We can use a monad to model this partiality. Every function that might not terminate can be embellished: we extend the return type with another value, by lifting it into a new type that contains all values of the original plus the "bottom" value.

This monad of partiality can be represented using the built-in Haskell type

```
data Maybe a = just a | nothing
```

`Maybe` is a parameterised type. Elements built using the just data constructor represent successful computations, whilst nothing represent failure.

The monad operators can be implemented as

```
return a = just a

m >>= f = case f of
            just a  -> just (f a)
            nothing -> nothing
```

We could explicitly encode failure (so that we can have functions return `failure`) by setting.

```
failure = nothing
```

> **Remark 16** (Partiality in Haskell)
> In Haskell, every function may be non-terminating. Therefore all types in Haskell are assumed to be lifted. This is also the reason why we talk about **Hask** the category of lifted Haskell types and functions, rather than simple **Set**.

Nondeterminism

Nondeterministic functions are functions that if given the same input, may return many different results. Semantically this is no different than a function returning a list of all possible answers. This equivalence is especially striking in a lazily evaluated language, where if we just need a single value, we can take the head of the list. Since the language is lazy it would never evaluate the tail. If instead, we wanted a random value, we could randomly select one element from the list.

The list monad implements the idea of non-deterministic computations. If we execute two nondeterministic computations in sequence, the first could return any of its possible outcomes. The outcome would then be passed to the second function, that again – considering this input – will output one of its possible return values.

This process modelled using a list can be represented as the first function giving a list of all its possible values. The second function is then applied to every possible output of the first, returning for every input – again – a list. To get the final list of all possible return values, we have to collapse all lists back into a single one.

The list monad can be implemented in Haskell as such:

```
return a = [a]

m >>= f = concat (map f m)
```

Alternatively, we could define

```
join = concat
```

Side-effects (State)

You may think of a function that has read/write access to some state as a pure function that takes in a state as an additional argument and produces a value-state pair as a result: `(a,s) -> (b,s)`. At first this might not look like a Kleisli arrow, but after currying `a -> ( s -> (b,s))` we can recognize it as a computation.

To implement the monad of side-effect in Haskell we define a new type, `State s a`, to represent computations producing an `a`, with a side-effect on a state of type `s`.

```
newtype State s a = state (s -> (s,a))
```

The monad operations are then:

```
return a = state (\s -> (s,a))

State m >>= f = state (\s -> let (s', a)  = m s
                                 state m' = f a
                             in m' s')
```

The state can be manipulated using

```
get :: State s s
get = state (\s -> (s,s))

put :: s -> State s ()
put s = state (\_ -> (s, ()))
```

Example 28 (Increment state)
For example, a function to increment the state could be expressed using these functions as

```
increment :: State Int ()
increment = get >>= \s ->
              put (s+1)
```

or alternatively:

```
increment :: State Int ()
increment = do
            s <- get
            put (s+1)
```

Remark 17 (Read-only and write-only)
Our previous logging computation, also know as the `Writer` monad is actually a specific case of this `State` monad. Where the state is write-only. Likewise, there is also a read-only state, facilitated by the `Reader` monad.

Exceptions

A function in an imperative language that throws an exception, is just a partial function in disguise. For specific input values it doesn't give back a return value but an exception, which likely just states that the function has no valid output value for that input. The difference is that it makes a fancy distinction between the types of failures it has.

The simplest implementation of exceptions in terms of pure total functions uses the `Maybe` functor like the model for partial functions. If we want to also return some information about the cause of the failure, as real exceptions do, we can use the `Either` functor instead. A computation that goes right, outputs a right value.

Here is the Monad instance for `Either`:

```
instance Monad Either String where
    left e >>= k = left e
    right a >>= k = k a
    return a = right a
```

A simple example of a function that can throw an exception is:

```
divide :: Int -> Int -> Either String Int
divide x y
        | y == 0    = left "Cannot divide by 0"
        | otherwise = right (x `div` y)
```

The monadic composition for `Either` correctly propagates the error, in fact, it shuts down any further computations when an error is detected. That's exactly what we want when an error is detected.

IO

Input/output has been a rather embarrassing topic for *purely* functional languages. And so have foreign function calls been an awkward story. There simply is no way to guarantee that a function written in a different language does not have a side-effect. We can't in good conscience call such function without risking compromising Haskell's purely functional semantics.

Despite this, we can still model input/output in a functional way. We can treat the entire universe as an object in our program. Then we consider a program to be a function from the state of the universe before it is run, to the state of the universe afterwards. Mathematically this is entirely fine, though in the real world we can't pass the state of the entire universe to our functions. That is fine as long as we never explicitly interact with it, we can treat it as if there is a type that perfectly describes this. We call this type `Universe`.

Then our `IO` type becomes:

```
type IO a = State Universe a
```

Along with that we alter the primitives slightly

```
readFile  :: String -> IO String
writeFile :: String -> String -> IO ()
```

Uses

Category theory gives us a mathematical model for programming. Category theory gives us a way to describe computational effects as monads. One advantage of this that is possible to make formal proofs of correctness of software. This might not seem that important for consumer grade products, those work reasonable well already, but for critical software, such as in rockets meant for human spaceflight, knowing that the program is proved to be correct would put me at ease.

⌈11⌋ Conclusions

In this paper, we've tried to explore the various applications category theory has in computer science. We witnessed how the de-compositional nature of programming problems gives birth to a category, where types are objects and functions are morphisms. We've seen how from this application of category theory various concepts in computer science are analogue to basic categorical definitions. We've argued that category theory holds the tools to explain the fundamental structure behind programming. We've applied category theory to various programming problems to improve our understanding, and in specific cases, this allows for elegant solutions. Either improving usability, re-usability or productivity.

Category theory provides a bunch of new tools -— Free theorems, Recursion Schemes and Monads. Using these devices, we can structure programs in new and useful ways, and we've shown several examples of this.

Still, the approaches discussed aren't perfect. For example, the IO monad in Haskell has developed into a sin-bin and monads do not compose nicely.

This paper is also relevant to the present controversy over the use of category theory to programming. Some believe that category theory – as abstruse as it is –- is just an abstract framework, too far removed from practical purposes, that it serves no place in real programming. A big asset category theory has to offer are generalized specifications and seeing patterns in a system you would otherwise not directly see. This is, of course, a result of abstraction, a good abstraction keeps just the right amount to reveal the essence of something. I believe category theory removes just the right things to reveal human thinking when building theories and finding structure.

This paper provides further evidence, that even this abstract nonsense is too versatile not to use. It is a powerful language we have to describe structure. We should not obstruct access to such a tool.

Bibliography

[1] Structure, 2020. Accessed: 18 June 2020.

[2] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming: An introduction. In *3rd International Summer School on Advanced Functional Programming*, pages 28–115. Springer-Verlag, 1999.

[3] Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. *APPSEM 2000. LNCS*, 2395, 10 2000.

[4] Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall, Inc., USA, 1997.

[5] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[6] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. *ACM SIGPLAN Notices*, 41(1):206–217, 2006.

[7] E.W. Dijkstra. *Notes on structured programming*. EUT report. WSK, Dept. of Mathematics and Computing Science. Technische Hogeschool Eindhoven, 2nd ed. edition, 1970.

[8] Louis Dionne. Familiar template syntax for generic lambdas. Technical Report P0428R2, ISO/IEC JTC 1, Informationtechnology, Subcommittee SC 22, Programming Language C++, july 2017.

[9] Samuel Eilenberg and Saunders MacLane. Group extensions and homology. *Annals of Mathematics*, 43(4):757–831, 1942.

[10] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.

[11] Brian Goetz. Lambda expressions for the java™ programming language. Technical Report JSR-335, Oracle, february 2014.

[12] J. Hackett and G. Hutton. Programs for cheap! In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 115–126, 2015.

[13] Ralf Hinze and Daniel WH James. Reason isomorphically! In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, pages 85–96, 2010.

[14] Jaakko Järvi, John Freeman, and Lawrence Crowl. Lambda expressions and closures: Wording for monomorphic lambdas. Technical Report N2550=08-0060, ISO/IEC JTC 1, Informationtechnology, Subcommittee SC 22, Programming Language C++, february 2008.

[15] Daniel M. Kan. Adjoint functors. *Transactions of the American Mathematical Society*, 87(2):294–329, 1958.

[16] Joachim Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 103:151–161.

[17] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer-Verlag New York, 2nd ed. edition, 1998.

[18] F. William Lawvere. An elementary theory of the category of sets. *Proceedings of the National Academy of Sciences of the United States of America*, 52(6):1506–1511, 1964.

[19] Saunders Mac Lane. The pnas way back then. *Proceedings of the National Academy of Sciences*, 94(12):5983–5985, 1997.

[20] Grant Reynold Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, 1990. date_submitted:2008 Rights: University of Groningen.

[21] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[22] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, Berlin, Heidelberg, 1991. Springer-Verlag.

[23] Bartosz Milewski. *Basic Category Theory for Programmers*. 2018.

[24] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[25] D. L. Parnas, John E. Shore, and David Weiss. Abstract types defined as classes of variables. *SIGMOD Rec.*, 8(2):149–154, March 1976.

[26] Simon Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001.

[27] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, MA, USA, 1991.

[28] David Pitt. *Categories*, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986.

[29] Andrew M. Pitts. Categorical logic. Technical Report UCAM-CL-TR-367, University of Cambridge, Computer Laboratory, may 1995.

[30] B. Plotkin. Algebra, categories and databases. volume 2 of *Handbook of Algebra*, pages 79 – 148. North-Holland, 2000.

[31] Henri Poincare. Science and hypothesis. la science et l'hypothèse, 1905.

[32] John C. Reynolds. Types, abstraction and parametric polymorphism. In Mason R.E.A, editor, *Proceedings of IFIP Congress*, volume 83, pages 513–523, Amsterdam, North-Holland, 1983. Elsevier Science Publishers B.V.

[33] P.J. Scott. Some aspects of categories in computer science. volume 2 of *Handbook of Algebra*, pages 3 – 77. North-Holland, 2000.

[34] Daniel Seidel and Janis Voigtländer. Improvements for free. *Electronic Proceedings in Theoretical Computer Science*, 57:89–103, Jul 2011.

[35] Faisal Vali, Herb Sutter, and Dave Abrahams. Proposal for generic (polymorphic) lambda expressions. Technical Report N3559, ISO/IEC JTC 1, Informationtechnology, Subcommittee SC 22, Programming Language C++, march 2013.

[36] Janis Voigtländer. Free theorems simply, via dinaturality, 2019.

[37] Philip Wadler. Theorems for free! In *FPCA*, 1989.

[38] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

Index