

Programming with Categories (DRAFT)

Brendan Fong Bartosz Milewski David I. Spivak

(Last updated: October 6, 2020)

Contents

Preface	vii
Learning programming	ix
Installing Haskell	x
Haskell points	xi
Acknowledgments	xiv
 1 Categories, Types, and Functions	 1
1.1 Programming: the art of composition	1
1.2 Two fundamental ideas: sets and functions	3
1.2.1 What is a set?	3
1.2.2 Functions	4
1.2.3 Some intuitions about functions	6
1.3 Categories	9
1.3.1 Motivation: the category of sets	9
1.3.2 The definition of a category	12
1.3.3 Examples of categories	14
1.3.4 Thinking in a category: the Yoneda perspective	20
1.4 Categories and Haskell	23
1.4.1 The lambda calculus	23
1.4.2 Types	26
1.4.3 Haskell functions	28
1.4.4 Composing functions	30
1.4.5 Thinking categorically about Haskell	33
 2 Universal constructions and the algebra of types	 35
2.1 Constructing datatypes	35
2.2 Universal constructions	37
2.2.1 Terminal objects	37
2.2.2 Initial objects	39
2.2.3 Products	40

2.2.4	Coproducts	43
2.3	Type constructors	44
2.3.1	Type constructors	45
2.3.2	Unit and void	46
2.3.3	Tuple types	48
2.3.4	Sum types	53
2.4	Exponentials and function types	56
2.4.1	Interlude: Distributivity	56
2.4.2	Exponential objects	57
2.4.3	Function types, and currying in Haskell	60
3	Functors, natural transformations, and type polymorphism	65
3.1	Relationships, relationships, relationships	65
3.2	Functors	66
3.2.1	Definition	66
3.2.2	Examples of functors	67
3.2.3	Functors and shapes	71
3.2.4	The category of categories	74
3.3	Type classes	74
3.3.1	Polymorphism in Haskell	74
3.3.2	Defining type classes and instances	75
3.4	Functors in Haskell	79
3.4.1	The Functor type class	79
3.4.2	First examples of functors in Haskell	81
3.4.3	Bifunctors	83
3.4.4	A first glance at profunctors	86
3.5	Natural transformations	88
3.5.1	Definition	88
3.5.2	Natural transformations in Haskell	89
3.6	Bonus: Representable functors and the Yoneda embedding	92
4	Algebras and recursive data structures	97
4.1	The string before the knot	98
4.2	What you can do with recursive data types	100
4.3	Algebras	103
4.4	Initial algebras	105
4.4.1	Lambek's lemma	107
4.5	Recursive data structures	108
4.5.1	Returning to expression trees	110
4.5.2	The essence of recursion	112
4.5.3	Algebras, catamorphisms, and folds	113
4.6	Coalgebras, anamorphisms, and unfolds	114

4.6.1	The type of streams, as a terminal coalgebra	116
4.6.2	The stream of prime numbers	117
4.7	Fixed points in Haskell	120
4.7.1	Implementing initial algebras by universal property	121
4.7.2	Implementing terminal coalgebras by universal property	123
5	Monads	125
5.1	A teaser	125
5.2	Different ways of working with monads	127
5.2.1	Monads in terms of the “fish”	127
5.2.2	Monads in terms of join	127
5.2.3	Monads in terms of bind	130
5.2.4	Monads in terms of the do notation	131
5.2.5	Monads and effects	134
5.3	Examples of monads	134
5.3.1	The exceptions monads	134
5.3.2	The list monad and nondeterminism	135
5.3.3	The writer monads	136
5.3.4	The reader monads	137
5.3.5	The state monads	139
5.3.6	The continuation monads	140
5.3.7	The IO monad	141
6	Monoidal Categories	143
6.1	Lax Monoidal Functors	143
6.1.1	Monad as Applicative	143
6.2	Strength and enrichment	146
7	Profunctors	149
7.1	Profunctors revisited	149
7.2	Ends and Coends	153
7.3	Profunctors in Haskell	156
7.4	Category of profunctors	158
7.4.1	Category of profunctors in Haskell	160
7.5	Day convolution	161
7.5.1	Day convolution in Haskell	163
7.6	Optics	163
7.6.1	Motivation	163
7.6.2	Tensorial optics	164
7.6.3	Lens	164
7.6.4	Lens in Haskell	165
7.6.5	Prism	165

7.6.6	Prism in Haskell	166
7.7	Profunctor optics	166
7.7.1	Tambara modules	166
7.7.2	Profunctor optics in Haskell	167
Index		169

Preface

This book is about how to think using category theory. What does this mean? Category theory is a favorite tool of ours for structuring our thoughts, whether they be about pure math, science, programming, engineering, or society. It's a formal, rigorous toolkit designed to emphasise notions of relationship, composition, and connection. It's a mathematical tool, and so is about abstraction, stripping away details and noticing patterns; one power it gains from this is that it is very concise. It's great for compressing thoughts, and communicating them in short phrases that can be reliably decoded.

The flip side of this is that it can feel very unfamiliar at first – a strange, new mode of thought – and also that it often needs to be complemented by more concrete tools to fully realise its practicality. One also needs to practice a certain art, or taste, in knowing when an abstraction is useful for structuring thought, and when it should be broken, or approximated. Programmers know this well, of course: a programming language never fully meets its specification; there are always implementation details that the best must be familiar with.

In this book, we've chosen programming, in particular *functional programming*, as the vehicle through which we'll learn how to think using category theory. This choice is not arbitrary: category theory is older than digital computing itself, and has heavily influenced the design of modern functional programming languages such as Haskell. As we'll see, it's also influenced best practice for how to structure code using these languages.

Thinking is not just an abstract matter; the best thought has practical consequences. In order to teach you how to think using category theory, we believe that it's important to give a mechanism for implementation and feedback on how category theory is affecting your thought. So in this book, we'll complement the abstraction of category theory – lessons on precise definitions of important mathematical structures – with an introduction to programming in Haskell, and lessons on how category theoretic abstractions are approximated to perform practical programming tasks.

That said, this book is not intended to be a book *about* Haskell, and there are many other programming languages that you could use as a world in which to try out these categorical ideas. In fact, we hope you do! Thinking categorically is about using a set

of design patterns that emphasise composition, universal constructions, and algebraic structure. Used judiciously, we believe this style of thought can improve the clarity and correctness of code in any language, including those currently popular, and those that will be developed in the years to come.

Category theory is a vast toolbox, that is still under active construction. A vibrant community of mathematicians and computer scientists is working hard to find new perspectives, structures, and definitions that lead to compact, insightful ways to reason about the world. We won't teach all of it. In fact, we'll teach a very small core, some central ideas all over fifty years old.

The first we'll teach is the namesake: categories. Programming is about composition: it's about taking programs, some perhaps just single functions, and making them work in sync to construct a larger, usually more expressive, program. We call this way of making them work in sync *composition*. A category is a world in which things may be composed. In programming these things are called, well, programs, but in category theory we use the more abstract word *morphism*. Morphisms are often thought of as processes or relationships. Processes have an input and an output; similarly, relationships form between objects. Morphisms are similar: categories also have *objects*, and morphisms have an input object, called a *domain*, and an output object, called a *codomain*.

Since programming is about composition, and categories model the essence of composition, one interesting game to play is to think of – we'll say *model* – a programming language as a category. In this model, morphisms correspond to programs. What do the domain and codomain of a morphism correspond to? The objects of the category correspond to *types*, like `int` or `string`. In a category, we can only compose a morphism f with a morphism g if the codomain of f is the same as the domain of g . If we're modelling a programming language as a category, this suggests we want to only allow composition of programs f and g if the output type of f is the same as the input type of g . A program that consumes an `int` shouldn't be able to accept a `string`! Haskell is designed with this principle in mind, and checks for this sort of error at compile time, only allowing construction of programs that are well-typed.

Category theory is about relationships, and as part of this viewpoint, relationships between categories are very important. The basic sort of relationship is known as a *functor*. A functor between categories \mathcal{C} and \mathcal{D} must give an object of \mathcal{D} for every object of \mathcal{C} . These correspond to type constructors (which a bit of additional structure). Going deeper still, the basic sort of relationship between functors is known as a *natural transformation*. These too are useful for thinking about programming: they are used to model polymorphic functions.

As may be now clear, types play an a fundamental role in thinking about programming from a categorical viewpoint. We'll next talk about how category theory lends insight into methods for constructing new types: first algebraic datatypes, and then recursive datatypes.

In category theory, this becomes a question of how to construct new objects from given objects. In category theory we privilege the notion of relationship, and a deep, beautiful part of theory concerns how to construct new objects just by characterizing them as having a special place in the web of relationships that is a category. For example, in Haskell there is the unit type `()`, and this has the special property that every other type `a` has a unique function `a → ()`. We say that the unit type thus has a special *universal property*, and in fact if we didn't know the unit type existed, we could recover or construct it by giving this property. More complicated universal properties exist, and we can construct new types in this way. In fact, we'll see that it's nice if our programming language can be modelled using a special sort of category known as a *cartesian closed category*; if so, then our programming language has product types and function types.

Another way of constructing types is using the (perhaps confusingly named) notion of algebras and coalgebras for a functor. In particular, specifying a functor lets us talk about universal constructions known as initial algebras and final coalgebras. These allow us to construct recursive datatypes, and methods for accessing them. Examples include lists and trees.

Functional programming is very neat and easy to reason about. It's lovely to be able to talk simply about the program square: `int → int`, for example, that takes an integer, and returns its square. But what happens if we also want this program to wait for an input integer, or print the result, or keep a log, or modify some state variable? We call these *side effects* of the program. Functional programming teaches that we should be very careful about side effects, as they can happen away from the type system, and this can make programs less explicit and modular, and more difficult to reason about. Nonetheless, we can't be too dogmatic. We do want to print results! Monads are special functors that each describe a certain sort of side effect, and are a useful way of being careful about side effects.

Learning programming

Throughout this book you'll be learning to program in Haskell. One way of learning a programming language is to start with formal syntax and semantics (although very few programming languages have formal semantics). This is very different from how most people learn languages, be it natural or computer languages. A child learns the meaning of words and sentences not by analyzing their structure and looking up the definitions in Wikipedia. In fact, a child's understanding of language is very similar to how we work in category theory. We derive the meaning of objects from the network of interactions with other objects. This is easy in mathematics, where we can say that an object is defined by the totality of its interactions with possibly infinite set of other objects (we'll see that when we talk about the Yoneda lemma). In real life we don't have the time to explore the infinity, so we can only approximate this process. We do

this by living on credit.

This is how we are going to learn Haskell. We'll be using things, from day one, without fully understanding their meaning, thus constantly incurring a conceptual debt. Some of this debt will be paid later. Some of it will stay with us forever. The alternative would be to first learn about cartesian closed categories and domain theory, formally define simply typed lambda calculus, build its categorical model, and only then wave our hands a bit and claim that at its core Haskell approximates this model.

We emphasize that this book is *not* intended to be a complete introduction to Haskell. While we will end up introducing all that is essential for learning to express categorical ideas in Haskell, there many beautiful features of the language that we will not touch upon, such as lazy compilation or list comprehensions. For the reader interested in learning Haskell itself, there are many good resources out there; see Hutton for example.

Installing Haskell

Start by downloading and installing the [Haskell Platform](http://www.haskell.org/ghc/) on your computer. (On the Mac or Linux, you'll have to close and reopen the terminal after installation, to initialize the environment.) To start an interactive session from the command line, type `ghci` (GHC stands for the Glasgow Haskell Compiler or, more affectionately, the Glorious Haskell Compiler; and 'i' stands for interactive). You can start by evaluating simple expressions at the prompt, e.g.:

```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> 2 * (3 + 4)
14
Prelude> 1/2
0.5
Prelude> 2^64
18446744073709551616
Prelude> cos pi
-1.0
Prelude> mod 8 3
2
Prelude> :q
Leaving GHCi.
```

Prelude is the name of the standard Haskell library, which is automatically included in every Haskell program and in every `ghci` session.

As you can see, besides basic arithmetic operations, you have access to some standard functions. Notice that a function call does not require the arguments to be parenthesized, even for two-argument functions like `mod` (division modulo). This is

something you will have to get used to, but it will make a lot of sense later, when we talk about partial application and currying (that's the kind of conceptual debt we were talking about).

To exit the command loop, type `:q`

The interactive environment is great for trying things out, but real programs have to be compiled from files. We'll be mostly working with single-file programs. Use your favorite editor to create and edit Haskell source files, which are files with the extension `.hs`. To begin with, create a file `main.hs` and put this line of code in it:

```
main = putStrLn "Hello Haskell!"
```

Compile this file using the command:

```
ghc main
```

and then run the resulting executable. The details depend on the operating system. For instance, you might have to invoke the program by typing `./main` at the command prompt:

```
$ ./main
Hello Haskell!
```

If you name your file something else, it will be treated as a separate module, and will have to include a module declaration at the top. For instance, a file named `hello.hs` will have to start with:

```
module Hello where
main = putStrLn "Hello Haskell!"
```

(Module names must start with upper case.)

Haskell points

Throughout the book, elements of Haskell will be interspersed with concepts from category theory, so it makes sense to periodically summarize what we have learned in a more organized way. For instance, we have just learned that you can do arithmetic in Haskell.

Operators You have at your disposal the usual operators `+`, `-`, `*`, and `/`, as well as the power operator `^`. You can apply them to integers and floating-point numbers (except for the power, which only allows integer exponents).

Unary minus There is a little gotcha with the unary minus: when you write a negative literal, like `-8`, it is treated like the application of the function `negate` to a literal `8`. This is why you sometimes have to parenthesize it, as in:

```
mod (-8) 3
```

or

```
2 * (-3)
```

Without parentheses, the Haskell compiler will issue one of its infamously cryptic error messages.

Functions Here are some useful functions: `abs`, `floor`, `ceiling`, `round`; there is integral division `div` and `mod` satisfying the equality:

```
(div x y) * y + (mod x y) == x
```

We have square root, `sqrt`, the usual trigonometric functions, and the constant `pi`, natural exponential `exp` and logarithm `log`.

Let syntax When using the interactive environment, `ghci`, variables are defined using the `let` syntax, e.g.:

```
let x = sin (pi / 4)
let llama = cos (pi / 4)
x^2 + llama^2
```

Expressions are immediately evaluated and the results are printed (if they are printable). Variable names must start with a lowercase letter. Conveniently, following mathematical usage, variable names may contain apostrophes, as in

```
let x' = 2 * x
```

Multi-line modes in GHCi To input multiple lines of Haskell into the interactive environment at once, wrap the code in `:{` and `:}`, e.g.:

```
:{
let abs x | x >= 0 = x
          | otherwise = -x
:}
```

Modules When working source files, the module definition has to be included at the top (except if the file is called `main.hs`, in which case the module statement can be omitted). For example, if the file is called `test.hs`, it must start with something like:

```
module Test where
```

The module name `Test` can be replaced with anything that starts an uppercase letter, and may contain periods, as in `Data.List`.

If you want to compile and execute a Haskell file, it must contain the definition of `main`.

Input/Output In general, input and output are performed in `main`. Unlike in imperative languages, I/O is special in Haskell and it requires the use of the `IO` monad. We will eventually talk about monads, but for now, we'll just follow a few simple rules. To begin with, we can print any string in `main` using `putStrLn`. We can also print things that can be displayed, like numbers or lists, using `print`. For instance:

```
main = print (2^31 - 1)
```

Remember to enclose composite expressions in parentheses when passing them to `print`.

If you want to do multiple I/O operations in `main`, you have to put them in the `do` block:

```
main = do
    putStrLn "This is a prime number"
    print (2^31 - 1)
```

Statements in a block must be all indented by the same amount. Haskell is whitespace sensitive.

Comments To make your code more readable to humans, you might want to include comments. An end-of-line comment starts with a double hyphen and extends till the end of line

```
id :: a -> a -- identity function
```

A multi-line comment starts with `{-` (curly brace, dash) and ends with `-}`.

Language pragmas Language pragmas have to be listed at the very top of a source file (before imports). Formally, these are just comments, but they are used by the compiler to influence the parsing.

```
{-# language ExplicitForAll #-}
```

Alternatively, pragmas can be set in GHCi using the command `:set`

```
Prelude> :set -XTypeApplications
Prelude> :t id
id :: a -> a
Prelude> :t id @Int
id @Int :: Int -> Int
```

Further resources Here are some useful online resources:

- <https://hoogle.haskell.org>. Hoogle lets you look up Haskell function definitions, either by name or by type signature.
- <https://www.cs.dartmouth.edu/cbk/classes/8/handouts/CheatSheet.pdf>, Haskell cheat sheet

Acknowledgments

The authors would like to thank David A. Dalrymple, Eliana Lorch, Nelson Niu, Paolo Perrone, and Gioele Zardini for many helpful comments.

Work by Spivak and Fong was sponsored in part by AFOSR grants and Topos Institute.

Categories, Types, and Functions

1.1 Programming: the art of composition

What is programming? In some sense, programming is just about giving instructions to a computer, a strange, very literal beast with whom communication takes some art. But this alone does not explain why more and more people are programming, and why so many (perhaps including you, dear reader) are interested in learning to program better. Programming is about using the immense power of a computer to solve problems. Programming, and computers, allow us to solve big problems, such as forecasting the weather, controlling a lunar landing, or instantaneously sending a photo to your mom on the other side of the planet.

How do we write programs to solve these big problems? We decompose the big problems into smaller ones. And if they are still too big, we decompose them again, and again, until we are left writing the very simple functions that come at the base of a programming language, such as concatenating two lists (or even modifying a register). Then, by solving these small problems and composing the solutions, we arrive at a solution to the larger problem.

To take a very simple example, suppose we wanted to take a sentence, and count how many words it contains. This capability is usually not provided to us in the base library of a programming language, such as Haskell's **Prelude**. Luckily, we can perform the task by composition.

Our first ingredient will be the function `words`, which takes a sentence (encoded as a string) and turns it into a comma-separated list of words. For example, here's what happens if we call it on the sentence "Hello world":

```
Prelude> words "Hello world"  
["Hello","world"]
```

Our second ingredient is the function `length`, which counts the number of entries in a list. For example, we might run:

```
Prelude> length ["I", "like", "cats"]
```

```
3
```

Composition in Haskell is denoted by a period “.” between two functions. We can define new functions by composing existing functions:

```
Prelude> let countwords = length . words
```

This produces a solution to our problem.

```
Prelude> countwords "Yay for composition!"
```

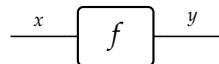
```
3
```

Given that composition is such a fundamental part of programming, and problem solving in general, it would be nice to have a science devoted to understanding the essence of composition. In fact, we have one: it’s known as category theory.

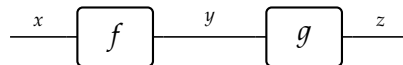
As Bartosz once wrote elsewhere, a category is an embarrassingly simple concept. A category is a bunch of objects, and some arrows that go between them. We assume nothing about what these objects or arrows are; all we have are names for them. We might call our objects very abstract names, like x , y , and z , or more evocative ones, like 42, True, or **String**. Our arrows have a source and a target. For example, we might have an arrow called f , with source x and target y . This could be depicted as an arrow:

$$x \xrightarrow{f} y$$

Or as a box called f , that accepts an ‘ x ’ and outputs a ‘ y ’:



What is important is that in a category we can compose. Given an arrow $f: x \rightarrow y$ and an arrow $g: y \rightarrow z$, we may compose them to get an arrow with source x and target z . We denote this arrow $g \circ f: x \rightarrow z$. We might also draw it as piping together two boxes:



This should remind you of our word counting example above, stripped down to its bare essence.

A category is a network of relationships, or slightly more precisely, a bunch of objects, some arrows that go between them, and a formula for composing arrows. A programming language generally has a bunch of types and some programs that go between them (i.e. take input of one type, and turn it into output of another). The guiding principle of this book, is that if you think of your (ideal) programming language like a category, good programs will result.

So let’s go forward, and learn about categories. To begin, it will be helpful to mention a few things about another fundamental notion: sets.

1.2 Two fundamental ideas: sets and functions

1.2.1 What is a set?

A set, in this book, is simply a bag of dots.

$$X = \begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline \bullet & \bullet & \bullet \\ \hline \end{array} \quad Y = \begin{array}{|c|c|c|c|} \hline a & \text{foo} & \heartsuit & 7 \\ \hline \bullet & \bullet & \bullet & \bullet \\ \hline \end{array} \quad Z = \bigcirc$$

This set X has three *elements*, the dots. We could write it in text form as $X = \{0, 1, 2\}$; when we write $1 \in X$ it means “1 is an element of X ”. The set Z has no elements; it’s called the *empty set*. The number of elements of a set X is called its *cardinality*, is denoted by $|X|$. Note that cardinalities of infinite sets may involve very large numbers indeed.

Example 1.1. Here are some sets you’ll see repeated throughout the book.

Name	Symbol	Elements between braces
The natural numbers	\mathbb{N}	$\{0, 1, 2, 3, \dots, 42^{2048} + 17, \dots\}$
The n th ordinal	\underline{n}	$\{1, \dots, n\}$
The empty set	\emptyset	$\{\}$
The integers	\mathbb{Z}	$\{\dots, -59, -58, \dots, -1, 0, 1, 2, \dots\}$
The booleans	\mathbb{B}	$\{\text{true}, \text{false}\}$

Exercise 1.2.

1. What is the cardinality $|\mathbb{B}|$ of the booleans?
2. What is the cardinality $|\underline{n}|$ of the n th ordinal?
3. Write $\underline{1}$ explicitly as elements between braces.
4. Is there a difference between $\underline{0}$ and \emptyset ?

◇

Definition 1.3. Given a set X , a *subset* of it is another set Y such that every element of Y is an element of X . We write $Y \subseteq X$, a kind of curved version of a less-than-or-equal-to symbol.

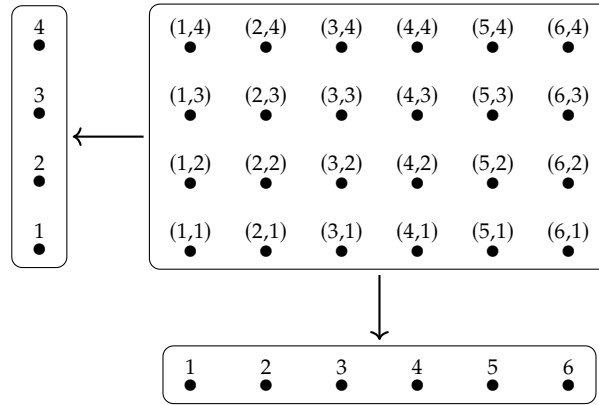
Exercise 1.4.

1. Suppose that a set X has finitely many elements and Y is a subset. Is it true that the cardinality of Y is necessarily less-than-or-equal-to the cardinality of X ? That is, does $Y \subseteq X$ imply $|Y| \leq |X|$?
2. Suppose now that Y and X are arbitrary sets, but that $|Y| \leq |X|$. Does this imply $Y \subseteq X$? If so, explain why; if not, give a counterexample.

◇

Definition 1.5. Given a set X and a set Y , their (*cartesian*) *product* is the set $X \times Y$ that has pairs (x, y) as elements, where $x \in X$ and $y \in Y$.

One should picture the product $X \times Y$ as a grid of dots. Here is a picture of $\underline{6} \times \underline{4}$, and how it relates to $\underline{6}$ and $\underline{4}$:



The name *product* is nice because the cardinality of the product is the product of the cardinalities: $|X \times Y| = |X| \times |Y|$. For example $|\underline{6} \times \underline{4}| = 24$, the product of 6 and 4.

Exercise 1.6. We said that the cardinality of the product $X \times Y$ is the product of $|X|$ and $|Y|$. Does that work even when X is empty? Explain why or why not. \diamond

One can take the product of any two sets, even infinite sets, e.g. $\mathbb{N} \times \mathbb{Z}$ or $\mathbb{N} \times \underline{4}$.

Exercise 1.7.

1. Name three elements of $\mathbb{N} \times \underline{4}$.
2. Name three subsets of $\mathbb{N} \times \underline{4}$.

\diamond

1.2.2 Functions

A mathematical function can be thought of as a machine that turns input values into output values. It's *total* and *deterministic*, meaning that every input results in at least one and at most one—i.e. exactly one—output. If you put in 5 today, you'll get a unique answer, and you'll get exactly the same answer if you put in 5 tomorrow.

Definition 1.8. Let X and Y be sets. A (*mathematical*) *function* f from X to Y , denoted $f: X \rightarrow Y$, is a subset of $f \subseteq X \times Y$ with the following properties.

- (a) For any $x \in X$ there is at least one $y \in Y$ for which $(x, y) \in f$.
- (b) For any $x \in X$ there is at most one $y \in Y$ for which $(x, y) \in f$.

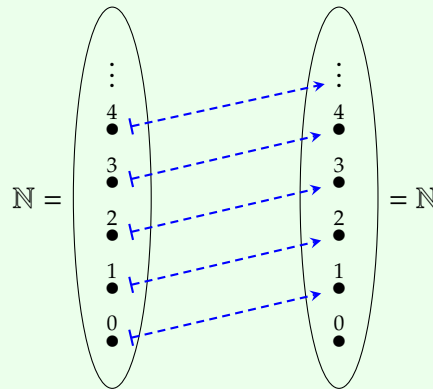
If f satisfies the first property we say it is *total*, and if it satisfies the second property we say it is *deterministic*. If f is a function (satisfying both), then we write $f(x)$ or $f x$ to denote the unique y such that $(x, y) \in f$.

We call X the *domain* of f , and Y the *codomain*.

Remark 1.9. The word function is also used for certain terms in the Haskell programming language. We'll talk about both mathematical functions and Haskell functions, and usually refer to both simply as 'functions'. The context should make it clear which one we mean.

This is a rather abstract definition; perhaps some examples will help. One way of denoting a function $f: X \rightarrow Y$ is by drawing "maps-to" arrows $x \mapsto y$ when $y = f(x)$. Since f is total and deterministic, every $x \in X$ gets exactly one arrow emanating from it, but no such rule for y 's.

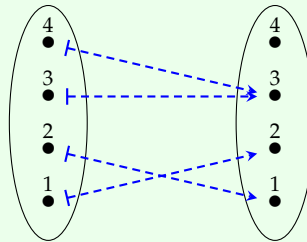
Example 1.10. The *successor* function $s: \mathbb{N} \rightarrow \mathbb{N}$ consists of all pairs $(n, n + 1)$, where $n \in \mathbb{N}$. In other words, s sends $n \mapsto n + 1$. For example $s(52) = 53$. Here's a picture:



Every natural number n input is sent to exactly one output, namely $s(n) = n + 1$.

Example 1.11. Here's a picture of the function $f: \underline{4} \rightarrow \underline{4}$ where

$$f = \{(1, 2), (2, 1), (3, 3), (4, 3)\} :$$



Note that in the domain every element has a unique arrow emanating from it, while in the codomain 3 receives two arrows, while 4 doesn't receive any.

Example 1.12. The function $\text{isEven}: \mathbb{N} \rightarrow \mathbb{B}$ sends a number n to **true** if n is even, and n to **false** if n is odd. We might write this using case notation, as follows:

$$\text{isEven}(n) = \begin{cases} \text{true} & n \text{ is even;} \\ \text{false} & n \text{ is odd.} \end{cases}$$

Exercise 1.13.

1. Suppose someone says “ $n \mapsto n-1$ is also a function $\mathbb{N} \rightarrow \mathbb{N}$. Are they right?”
2. Suppose someone says “ $n \mapsto 42$ is also a function $\mathbb{N} \rightarrow \mathbb{N}$. Are they right?”
3. Draw arrows from elements in $\underline{3}$ to elements in $\underline{4}$ in a way that’s not total.
4. Draw arrows from elements in $\underline{3}$ to elements in $\underline{4}$ in a way that’s not deterministic.

◇

Exercise 1.14.

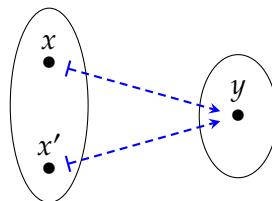
1. How many functions $\underline{3} \rightarrow \underline{2}$ are there? Write them all.
2. How many functions $\underline{1} \rightarrow \underline{7}$ are there? Write them all.
3. How many functions $\underline{3} \rightarrow \underline{3}$ are there?
4. How many functions $\underline{0} \rightarrow \underline{7}$ are there?
5. How many functions $\underline{0} \rightarrow \underline{0}$ are there?
6. How many functions $\underline{7} \rightarrow \underline{0}$ are there?

◇

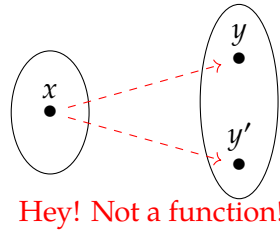
In general, for any $a, b \in \mathbb{N}$ there are b^a functions $\underline{a} \rightarrow \underline{b}$. You can check your answers above using this formula. One case which may be confusing is when $a = b = 0$. In this case, some mathematicians, such as a calculus teacher, might prefer to say “the expression 0^0 is undefined”, but we’re not in calculus class. It is true that if a and b are real numbers, there is no smooth way to define 0^0 , but for natural numbers, the formula “count the number of functions $a \rightarrow b$ ” works so well, that here we define $0^0 = 1$.

1.2.3 Some intuitions about functions

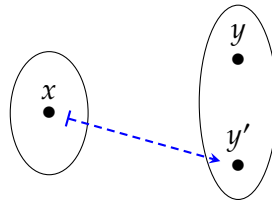
A lot of intuitions about functions translate into category theory. A function is allowed to collapse multiple elements from the domain into one element of the codomain:



On the other hand, a function is forbidden from splitting a domain element into multiple codomain elements.



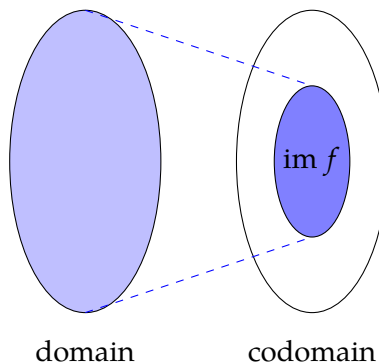
A second source of asymmetry: functions are defined for all elements in the domain, but not every element of the codomain needs to be hit. For example:



The subset of elements in the codomain that are mapped to by the source is called the *image* of a function:

$$\text{im } f = \{y \in Y \mid \exists x \in X. f(x) = y\}$$

The symbol \exists is shorthand for *there exists*, and so we read this statement: “The image of f is the set of y ’s in Y such that there exists an x in X where $f(x) = y$.” One might picture this as follows:



The directionality of functions is reflected in the notation we are using: we represent functions as arrows going from source to target, from *domain* to *codomain*. This directionality makes them interesting.

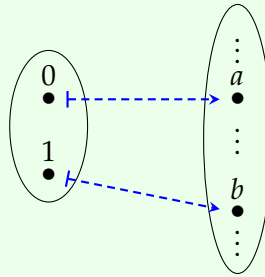
You may think of a function that maps many things to one as discarding some information. The function $\mathbb{N} \rightarrow \mathbb{B}$ that takes a natural number and returns `true` if it’s even and `false` otherwise doesn’t care about the precise value of a number, it only cares about it being even or odd. It *abstracts* some important piece of information by discarding the details it considers inessential.

You may think of a function that doesn't cover the whole codomain as *modeling* its domain in a larger environment. It's creating a model of its domain in a larger context, especially when it additionally collapses it by discarding some details. A helpful intuition is to think of the domain as defining a shape that is projected into a bigger set and forms a pattern there. We'll see later that, compared to category theory, set theory offers a very limited choice of bare-bones shapes.

Abstraction and modeling are the two major tools that help us understand the world.

Example 1.15. A *singleton set*—that is, a set with one element, such as $\underline{1}$ or $\{*\}$ —is the simplest non-trivial shape. A function from a singleton picks a single element from the target set. There are as many distinct functions from a singleton to a non-empty set as there are elements in that set. In fact we may identify elements of a set with functions from the singleton set. We'll see this idea used in category theory to define *global elements*.

Example 1.16. A two-element set can be used to pick pairs of elements in the target set. It embodies the idea of a pair. For example, functions from $\underline{2} \rightarrow X$ are much the same as pairs $(a, b) \in X \times X$.



As an example, consider a function from a two-element set to a set of musical notes. You may say that it embodies the idea of a musical interval.

Exercise 1.17. Let N be the set of musical notes, or perhaps the keys on a standard piano. Person A says “a musical interval is a subset $I \subseteq N$ such that I has two elements. Person B says “no, a musical interval is a function $i: \underline{2} \rightarrow N$, from a two element set to N . They prepare to fight bitterly, but a peacemaker comes by and says “you’re both saying the same thing!” Are they? \diamond

Exercise 1.18. How would you describe functions from an empty set to another set A . What do they model? (This is more of a Zen meditation than an exercise. There are no right or wrong answers.) \diamond

Remark 1.19. You might have noticed that our definition of a set—“a bag of dots”—is rather informal, and we’ve focussed on introducing sets by discussing what you can

do with them, like form pairs and construct functions, rather than formally saying “what they really are”. This is not to avoid the topic, but to establish a starting point. The theory of sets is a rich subject, rich enough to be considered a *foundation* for mathematics, but it’s also rather technical, and the details will distract us from the main, categorical story here. Instead, we’ve just chosen to begin with a working knowledge of sets and functions, and basic constructions like the cartesian product, because these are necessary to define categories.

There is another incentive to studying set theory, that is that sets themselves form a category, which is a very fertile source of examples and intuitions. On the one hand, we wouldn’t like you to think of morphisms in an arbitrary category as being some kind of functions. On the other hand, we can define and deeply understand many properties of sets and functions that can be generalized to categories. In fact, we can think of sets as primordial, ‘discrete’ categories.

What an empty set is everybody knows, and there’s nothing wrong in imagining that an initial object in a category is “sort of” like an empty set. At the very least, it might help in quickly rejecting some wrong ideas that we might form about initial objects. We can quickly test them on empty sets. Programmers know a lot about testing, so this idea that the category of sets is a great testing platform should sound really attractive.

1.3 Categories

In this section we’ll define categories, and give a library of useful examples. We begin by motivating the definition by discussing the ur-example: the category of sets and functions.

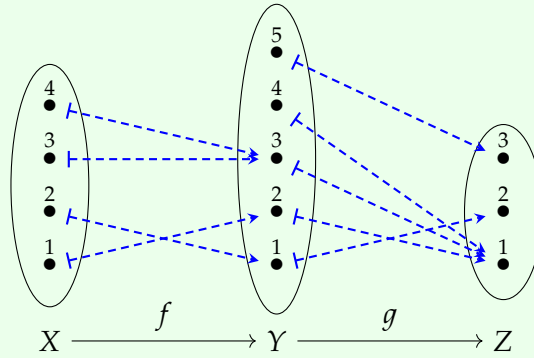
1.3.1 Motivation: the category of sets

The *identity function* on a set X is the function $\text{id}_X: X \rightarrow X$ given by $\text{id}_X(x) = x$. Given an input x , it outputs that same x . It does nothing. This might seem like a very boring thing, but it’s like 0: adding it does nothing, but that makes it quite central. For example, 0 is what defines the relationship between 6 and -6: they add to 0.

Just like 0 as a number really becomes useful when you know how to combine numbers using $+$, the identity function really becomes useful when you know how to combine functions using “composition”.

Definition 1.20. Let $f: X \rightarrow Y$ and $g: Y \rightarrow Z$ be functions. Then their *composite*, denoted either $g \circ f$ or $f \circ g$, is the function $X \rightarrow Z$ sending each $x \in X$ to $g(f(x)) \in Z$.

Example 1.21. A great way to visualize function composition is by path following.

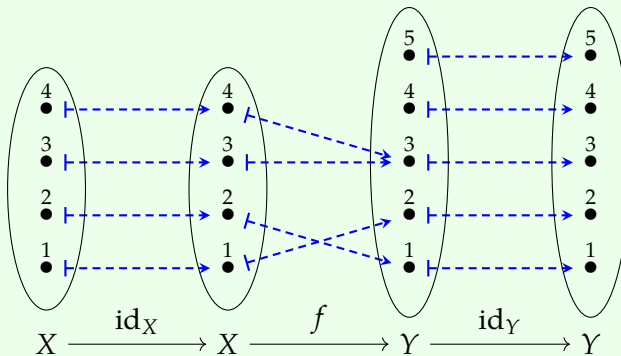


Following the paths from X to Z , we see that $g(f(3)) = 1$ and $g(f(2)) = 2$.

Exercise 1.22. Recall the successor function $s: \mathbb{N} \rightarrow \mathbb{N}$ from Example 1.10. What is $(s \circ s)(4)$? How about $(s \circ s)(23)$? Give a general description of $s \circ s$. \diamond

Remark 1.23. There are two competing notational conventions for composing functions and, as we'll see later, for composing morphisms in any category. Since we usually draw arrows in the left-to-right order, as in $f: X \rightarrow Y$, or $X \xrightarrow{f} Y$, it's natural to write composition in *diagrammatic order*, writing $f \circ g$ for $X \xrightarrow{g} Y \xrightarrow{f} Z$. On the other hand, we've already discussed the standard 'function application' notation fx or $f(x)$. If we were to apply f to x and then g to the result, then we'd write $g(fx)$ or $g(f(x))$. For this reason, it's natural to write composition in *application order*, writing $g \circ f$ so that $(g \circ f)x = g(fx)$. Since application order is the one preferred in Haskell, we'll usually use that. We pronounce $g \circ f$ as *g after f*.

Example 1.24. Suppose that $f: X \rightarrow Y$ is a function. Then if we compose it with either (or both!) of the identity functions, $\text{id}_X: X \rightarrow X$ or $\text{id}_Y: Y \rightarrow Y$, the result is again f .



To make this precise, we need to say what it means for two functions to be equal.

Definition 1.25. Two functions $f: X \rightarrow Y$ and $g: X \rightarrow Y$ are equal if $f(x) = g(x)$ for every $x \in X$.

The *unit laws* then say that for any $f: X \rightarrow Y$, we have $f \circ \text{id}_X = f$ and $\text{id}_Y \circ f = f$. This gives the slogan:

Composing with the identity doesn't do anything.

Another important property of function composition is that you can compose multiple functions at once, not just two. That is, if you have a string of n functions $X_0 \xrightarrow{f_1} X_1 \xrightarrow{f_2} \cdots \xrightarrow{f_n} X_n$, you can collapse it into one function by composing two-at-a-time in many different ways. This is denoted mathematically using parentheses. For example we can compose this string of functions $V \xrightarrow{h} W \xrightarrow{g} X \xrightarrow{f} Y \xrightarrow{e} Z$ as any of the five ways represented in the pentagon below:

$$\begin{array}{c}
 e \circ (f \circ (g \circ h)) \\
 \bullet \\
 \swarrow \quad \searrow \\
 e \circ ((f \circ g) \circ h) \quad (e \circ f) \circ (g \circ h) \\
 \bullet \qquad \bullet \\
 \swarrow \quad \searrow \quad \swarrow \quad \searrow \\
 (e \circ (f \circ g)) \circ h \quad ((e \circ f) \circ g) \circ h \\
 \bullet \qquad \bullet
 \end{array}
 \tag{1.26}$$

It turns out that all these different ways to collapse four functions into a single function give the same answer. You could write it simply $e \circ f \circ g \circ h$ and forget the parentheses all together.

A better word than “collapse” is *associate*: we’re associating the functions in different ways. The *associative law* says that $(f \circ g) \circ h = f \circ (g \circ h)$. The slogan is:

When composing functions, how you parenthesize doesn't matter: you'll get the same answer no matter what.

Exercise 1.27. Let $a: \underline{4} \rightarrow \mathbb{N}$ send a number $n \in \underline{4}$ to $5 \times n$, and recall the successor function $s: \mathbb{N} \rightarrow \mathbb{N}$ of Example 1.10, and the function $\text{isEven}: \mathbb{N} \rightarrow \mathbb{B}$ of Example 1.12.

Show that $(\text{isEven} \circ s) \circ a = \text{isEven} \circ (s \circ a)$. \diamond

Exercise 1.28. Consider the pentagon (sometimes called the *associahedron*) in Eq. (1.26).

1. Show that each of the five dotted edges corresponds to an instance of the associative law in action.

2. Are there any other places where we could do an instance of the associative law that isn't drawn as a dotted edge?

◇

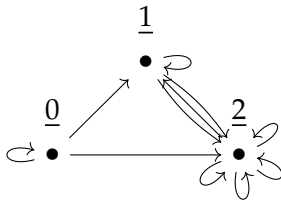
To summarise, we have discussed sets, functions between sets, and composition of functions. Composition of functions obeys three laws: the left and right unit laws, and the associative law. This is all the data needed for a category.

1.3.2 The definition of a category

Here's a slogan:

A category is a network of composable relationships.

The prototypical category is **Set**, the category of sets.¹ The objects of study in **Set** are, well, sets. The relationships of study in **Set** are the functions. These form a vast network of arrows pointing from one set to another.



(and if you toss in 3, you'll need to add $3^0 + 3^1 + 3^2 + 3^3 + 2^3 + 1^3 + 0^3 = 49$ more arrows with it! See <https://oeis.org/A231344>) (1.29)

But **Set** not just any old network of relationships: it's organized in the sense that we know how to compose the functions. Just like the mother of my mother is my grandmother, if a function relates X and Y , and another function relates Y and Z , we can compose them to get a relationship between X and Z . This imposes a tight constraint: if pretty much any function was somehow left out of the network in **Set**, we would find that certain functions wouldn't have a composite in the network.

Let's see the precise definition.

Definition 1.30. A category \mathcal{C} consists of four constituents:

- (i) a set $\text{Ob}(\mathcal{C})$, elements of which are called *objects of \mathcal{C}* ;
- (ii) for every pair of objects $c, d \in \text{Ob}(\mathcal{C})$ a set $\mathcal{C}(c, d)$, elements of which are called *morphisms from c to d* and often denoted $f: c \rightarrow d$;
- (iii) for every object c , a specified morphism $\text{id}_c \in \mathcal{C}(c, c)$ called the *identity morphism for c* ; and

¹More properly, **Set** is the category of *small sets*, meaning sets all of whose elements come from some huge pre-chosen 'universe' \mathcal{U} . The reason we have to qualify this is that paradoxes result when you try to make sense of the idea of 'the set of all sets'. We won't get into the set theoretical aspects regarding the idea of universes, but instead just state that the objects of **Set** are sets up to a certain size (which may be very, very large indeed, just not paradoxically large).

In the end, to learn how to think categorically, you don't need to worry about all these "size issues"; just focus on the category theory for now. If you do later want to get deep into the technical set-theoretic issues, see [**].

- (iv) for every three objects b, c, d and morphisms $f: b \rightarrow c$ and $g: c \rightarrow d$, a specified morphism $(g \circ f): b \rightarrow d$ called the *composite of g after f* (sometimes denoted $f \circ g$).

These constituents are subject to three constraints:

Left unital: for any $f: c \rightarrow d$, the equation $\text{id}_c \circ f = f$ holds;

Right unital: for any $f: c \rightarrow d$, the equation $f \circ \text{id}_d = f$ holds;

Associative: for any $f_1: c_1 \rightarrow c_2$, $f_2: c_2 \rightarrow c_3$, and $f_3: c_3 \rightarrow c_4$, the following equation holds

$$(f_3 \circ f_2) \circ f_1 = f_3 \circ (f_2 \circ f_1).$$

If $f: c \rightarrow d$ is a morphism, we again call c the *domain* and d the *codomain* of f .

So that's the key definition of this book: a category is some objects, some morphisms that relate them, and a rule for composing these morphisms!

Note that we can only compose morphisms f and g when the codomain of f is equal to the domain of g . If we think again about the picture Eq. (1.29), composition allows us to take a sequence of arrows head-to-tail in the picture, say from $\underline{0}$ to $\underline{2}$ to $\underline{1}$, and compose or 'summarise' it as a single arrow from $\underline{0}$ to $\underline{1}$. Moreover, associativity says we can take any sequence, following as many arrows as we like from head-to-tail, and uniquely collapse it down to a single arrow capturing the whole sequence. The identity morphism says that for any object, say $\underline{0}$, we can simply choose an empty sequence, 'do nothing', and that still can be represented as a morphism in the category: the identity morphism.

You should get familiar with the notation $\mathcal{C}(a, b)$ for the set of morphisms between two objects. It contains the name of the category, here \mathcal{C} , followed by the names of two objects between parentheses. For example, $\mathbf{Set}(A, B)$ is the set of functions from A to B .

Example 1.31 (The category of sets). The category of sets, denoted \mathbf{Set} has all the sets^a as its objects. Given two sets $A, B \in \text{Ob}(\mathbf{Set})$, the set $\mathbf{Set}(A, B)$ has as its elements all functions $f: A \rightarrow B$. (There's sets everywhere: objects are sets, and for every two objects we have another set $\mathbf{Set}(A, B)$.) The identity morphism $\text{id}_A: A \rightarrow A$ is just the identity function, and composition is given by composition of functions. We saw that this data obeys the unit and associative laws in Section 1.3.1.

^aAgain, up to some large size.

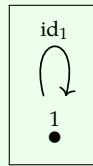
Exercise 1.32. Let's return to a remark we made earlier about the category \mathbf{Set} . Is there any single morphism you can take out of it, such that—without taking away any more morphisms—the remaining collection of objects and morphisms is still a category? \diamond

1.3.3 Examples of categories

Learning many different examples of categories is useful for having a collection of tools to deepen your understanding of the categorical concepts we'll encounter in this book. It will help separate out the essential features of the concepts from those that are incidental to one or two examples. Think of them like a testing suite for your categorical mental models!

Even though the notion of a category was inspired by examples like **Set**, there are tons of categories that don't really resemble it at all! Some of the most important categories are the little ones. It's like how the numbers 0, 1, and 2 are more often used than the number 9^9 .

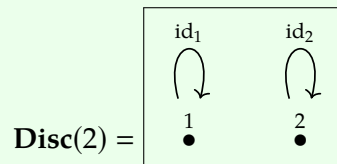
Example 1.33. There is a category **1** with one object, which we call 1, and with only one morphism, which we call id_1 .



We call the unique morphism id_1 as it must be the identity morphism on 1; there's no other morphism to choose. What is the composition rule? Well, the only morphism in the category is $\text{id}_1: 1 \rightarrow 1$, so we only need to say what the composite $\text{id}_1 \circ \text{id}_1$ is. We define it to be id_1 ; again, since there are no other morphisms, it's the only choice we can make!

Exercise 1.34. Consider Example 1.33 in light of the formal definition of a category, Definition 1.30. What is the set $\text{Ob}(\mathbf{1})$? What is the set $\mathbf{1}(1, 1)$? Why does **1** satisfy the unit and associative laws? \diamond

Example 1.35 (Discrete categories). There is also a category with two objects 1 and 2 and two morphisms $\text{id}_1: 1 \rightarrow 1$ and $\text{id}_2: 2 \rightarrow 2$.



In fact, for every set S , there's an associated category **Disc(S)**, called the *discrete category* on S . The objects of **Disc(S)** are the elements of S , i.e. $\text{Ob}(\mathbf{Disc(S)}) = S$, and the

morphisms are just the identities:

$$\mathbf{Disc}(\mathbf{S})(s, s') = \begin{cases} \{\text{id}_s\} & \text{if } s = s' \\ \emptyset & \text{if } s \neq s' \end{cases}$$

In particular, there is the *empty category*, which has no objects at all, given by $\mathbf{0} = \mathbf{Disc}(\mathbf{0})$.

So far we haven't gained any advantage over sets. But in a category we not only have objects; we may have arrows between them. This is when interesting structures arise. For instance, we can add a morphism $ar: 1 \rightarrow 2$ to the two-object category.

Example 1.36. This tiny category is sometimes called the *walking arrow category* **2**.

$$\mathbf{2} := \boxed{\text{id}_1 \hookrightarrow \bullet \xrightarrow{f} \bullet \rightrightarrows \text{id}_2}$$

Exercise 1.37. Define a composition rule and identity morphisms for **2** such that the resulting data satisfies the three the laws of a category. Could you have made any other choice? \diamond

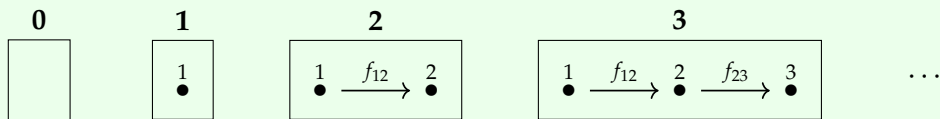
Since an identity morphism id_a automatically has to be there for each object a , and since a composite morphism $g \circ f$ automatically has to be there for every pair of morphisms $f: a \rightarrow b$ and $g: b \rightarrow c$, we often leave these out of our pictures.

Example 1.38 (Not drawing all morphisms). In the picture $\boxed{\bullet \rightarrow \bullet \rightarrow \bullet}$, only two arrows are drawn, but there are implicitly six morphisms: three identities, the two drawn arrows, and their composite.

So with this new convention, we redraw the walking arrow category from *Example 1.36* as

$$\mathbf{2} := \boxed{\begin{matrix} 1 \\ \bullet \end{matrix} \xrightarrow{f} \begin{matrix} 2 \\ \bullet \end{matrix}}$$

Example 1.39 (Ordinal categories). There is a progression of *ordinal categories* that look like this:

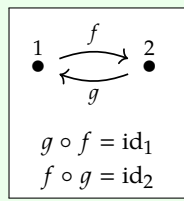


Exercise 1.40. Being sure to take into account Example 1.38,

1. How many morphisms are there in $\mathbf{3}$?
2. How many morphisms are there in \mathbf{n} ?
3. Are $\mathbf{0}$ and $\mathbf{Disc}(0)$ the same?
4. Are $\mathbf{1}$ and $\mathbf{Disc}(1)$ the same?

◇

Example 1.41 (The walking isomorphism). In the following category, which we will call \mathbf{I} , there are two objects and four morphisms:

 <p> $g \circ f = \text{id}_1$ $f \circ g = \text{id}_2$ </p>	below \circ right	id_1	id_2	f	g
	id_1	id_1	Hey!	Hey!	g
	id_2	Hey!	id_2	f	Hey!
	f	f	Hey!	Hey!	id_2
	g	Hey!	g	id_1	Hey!

To the left we see the drawing, with equations that tell us how morphisms compose. To the right, we see a table of all the composites in the category. Whenever two morphisms are not composable—the output object of one doesn’t match the input type of the other—the table yells at you. In Haskell, the error message is something like “couldn’t match types”.

Exercise 1.42. Suppose that someone tells you that their category \mathcal{C} has two objects c, d and two non-identity morphisms, $f: c \rightarrow d$ and $g: d \rightarrow c$, but no other morphisms. Does f have to be the inverse of g , i.e. is it forced by the category axioms that $g \circ f = \text{id}_c$ and $f \circ g = \text{id}_d$?

◇

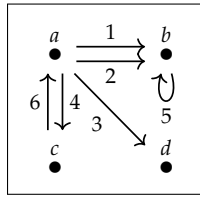
Free categories

One simple way to construct a category is to freely construct one from a graph.

Definition 1.43. A *graph* consists of two sets V, E and two functions $\text{src}: E \rightarrow V$ and $\text{tgt}: E \rightarrow V$. The elements of V are called vertices, the elements of E are called edges, and for each edge $e \in E$, the vertex $\text{src}(e)$ is called its source and the vertex $\text{tgt}(e)$ is called its target.

One can depict a graph by drawing a dot on for each element $v \in V$ and an arrow

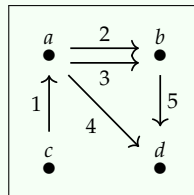
for each element $e \in E$, making the arrow point from the source of e to the target of e :



Edge	src	tgt	Vertex
1	a	b	a
2	a	b	b
3	b	b	c
4	a	c	d
5	b	b	
6	c	a	

For any graph G as above, there is an associated category, called the *free category* on G , denoted $\mathbf{Free}(G)$. The objects of $\mathbf{Free}(G)$ are the vertices of G . The morphisms of $\mathbf{Free}(G)$ are the paths in G , i.e. the head-to-tail sequences of edges in G . For each vertex, the trivial path—no edges—counts as a morphism, namely the identity. You can compose paths (just stick them head-to-tail), and this composition is associative. We say that this category is the ‘free’ category because composition obeys no equations, or constraints: every distinct path is a distinct morphism.

Exercise 1.44. Consider the following graph G :



How many objects are there in $\mathbf{Free}(G)$? Name them. How many morphisms are there in $\mathbf{Free}(G)$? Name them too. \diamond

Exercise 1.45. Is the ordinal category **3** (see Example 1.39) the free category on a graph? If so, on which graph; if not, why not? \diamond

Monoids

When getting to know a new mathematical structure, it’s often helpful to think about simplifying assumptions that can help give a feel for just one aspect of the structure. For example, an interesting assumption you can make about a category \mathbf{C} is that it has just one object. It doesn’t quite matter what this object is called, so let’s just call it $*$. In this case, there is just one homset too: $\mathbf{C}(*, *)$. In other words, all morphisms in \mathbf{C} are morphisms $* \rightarrow *$.

The result is called a *monoid*. These are incredibly useful constructs in programming too (for example, we’ll see how to use them to fold over recursive data structures), so let’s formally introduce the name and some terminology.

Definition 1.46. A monoid (M, e, \diamond) consists of

- (i) a set M , called the *carrier*;
- (ii) an element $e \in M$, called the *unit*; and
- (iii) a function $\diamond: M \times M \rightarrow M$, called the *operation*.

These are subject to two conditions:

- (a) (unitality) for any $m \in M$, we have $e \diamond m = m$ and $m \diamond e = m$;
- (b) (associativity) for any $l, m, n \in M$, we have $(l \diamond m) \diamond n = l \diamond (m \diamond n)$.

Exercise 1.47. Why is a monoid the same as a category with exactly one object? Suppose I have a category \mathbf{C} with a single object $*$, what is the carrier set of the resulting monoid? What is the unit and the operation? \diamond

Example 1.48. A very familiar example is $(\mathbb{N}, 0, +)$, the *additive monoid of natural numbers*. The carrier is \mathbb{N} , the unit is 0, and the operation is $+$. In this monoid, you can “add numbers in sequence”, e.g. $5 + 6 + 2 + 2$.

Exercise 1.49. Remember that we write \mathbb{B} for the set containing `true` and `false`; we call this set the ‘booleans’. To make a set into the carrier of a monoid, we need to specify an operation and a unit. Two well-known choices for an operation are ‘AND’ and ‘OR’. For each of these two operations, what is the unit? \diamond

Exercise 1.50. Monoids are everywhere! Look for a monoid in your life, and write down its carrier, unit, and operation. Why is it unital and associative? \diamond

Preorders

Monoids are categories that are tiny in terms of objects—just one! Similarly, we can discuss categories which are tiny (or *thin*) in terms of morphisms.

A *preorder* is a category such that, for every two objects a, b , there is *at most* one morphism $a \rightarrow b$. That is, there either is or is not a morphism from a to b , but there are never two morphisms a to b . If there is a morphism $a \rightarrow b$, we write $a \leq b$; if there is not a morphism $a \rightarrow b$, we don’t.

Like monoids, preorders are important enough to warrant their own terminology, separate from their existence as thin categories.

Definition 1.51. A preorder (P, \leq) consists of:

- (i) a set P ; and
- (ii) a subset \leq of $P \times P$, called the *order*.

Given $(p, q) \in P \times P$, we write $p \leq q$ when the pair is in the subset \leq . These are subject

to two conditions:

- (a) (reflexivity) for any $p \in P$, we have $p \leq p$;
- (b) (transitivity) for any $p, q, r \in P$ such that $p \leq q$ and $q \leq r$, we have $p \leq r$.

Example 1.52. If you can think of a collection of objects you would call ordered, then it's likely to be a preorder. For example, the natural numbers \mathbb{N} , integers \mathbb{Z} , and real numbers \mathbb{R} all form preorders with their usual \leq order.

Example 1.53. There is a preorder \mathcal{P} whose objects are the positive integers $\text{Ob}(\mathcal{P}) = \mathbb{N}_{\geq 1}$ and where

$$\mathcal{P}(a, b) := \{x \in \mathbb{N} \mid x * a = b\}$$

This is a preorder because either $\mathcal{P}(a, b)$ is empty (if b is not divisible by a) or contains exactly one element.

Exercise 1.54. Consider the preorder \mathcal{P} of Example 1.53. We know that we can think of it as a category with at most one morphism between any two objects.

1. What is the identity on 12?
2. Show that if $x: a \rightarrow b$ and $y: b \rightarrow c$ are morphisms, then there is a morphism $y \circ x$ to serve as their composite.
3. Would it have worked just as well to take \mathcal{P} to have all of \mathbb{N} as objects, rather than just the positive integers? \diamond

Building new categories from old

When you have a category, or two or three, you can use them to build other categories! Here are three common ways we'll use.

Example 1.55 (Opposite category). For any category \mathcal{C} , there is a category \mathcal{C}^{op} defined by “turning all the arrows around”. That is, the two categories have the same objects, and all the arrows simply point the other way:

$$\text{Ob}(\mathcal{C}^{\text{op}}) := \text{Ob}(\mathcal{C}) \quad \text{and} \quad \mathcal{C}^{\text{op}}(a, b) := \mathcal{C}(b, a).$$

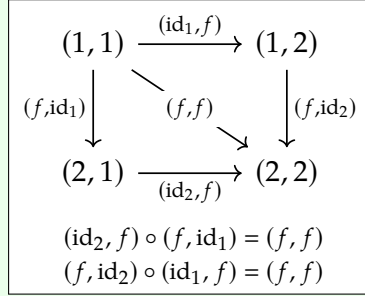
The opposite category is so much like the original category that it can be tricky at first to spot the difference. But the difference is important: category theory is all about the arrows, so which direction they point is fundamental!

Example 1.56 (Product category). Suppose \mathcal{C} and \mathcal{D} are categories. We can form a new

category $\mathcal{C} \times \mathcal{D}$ as follows:

$$\text{Ob}(\mathcal{C} \times \mathcal{D}) := \text{Ob}(\mathcal{C}) \times \text{Ob}(\mathcal{D}) \quad \text{and} \quad (\mathcal{C} \times \mathcal{D})((c, d), (c', d')) := \mathcal{C}(c, c') \times \mathcal{D}(d, d').$$

For example, the category $\mathbf{2} \times \mathbf{2}$ looks like this:



Example 1.57 (Full subcategories). Let \mathcal{C} be any category, and suppose you want “only some of the objects, but all the morphisms between them”. That is, you start with a subset of the objects, say $D \subseteq \text{Ob}(\mathcal{C})$, and you want the biggest subcategory of \mathcal{C} containing just those objects; this is called the *full subcategory of \mathcal{C} spanned by D* and we denote it $\mathcal{C}_{\text{Ob}=D}$. It’s defined by

$$\text{Ob}(\mathcal{C}_{\text{Ob}=D}) := D \quad \text{and} \quad \mathcal{C}_{\text{Ob}=D}(d_1, d_2) = \mathcal{C}(d_1, d_2).$$

For example, the category of finite sets is the full subcategory of **Set** spanned by the finite sets.

Exercise 1.58. Does the picture shown in Eq. (1.29) look like the full subcategory of **Set** spanned by $\{\underline{0}, \underline{1}, \underline{2}\} \subseteq \text{Ob}(\mathbf{Set})$? Why or why not? \diamond

1.3.4 Thinking in a category: the Yoneda perspective

A category is a web of relationships. A cornerstone of the philosophy of category theory, stemming from a central result known as the Yoneda lemma, is that relationships are all that’s needed: an object is no more, nor less, than how it relates to others.

One first peek at this is through the notion of a generalized element. Let a be an object in a category \mathcal{C} . Given any object c in \mathcal{C} , we can ask what a looks like from the point of view of c . (We choose c for ‘seer’.) The answer to this is encoded by the homset $\mathcal{C}(c, a)$, which is the set of all morphisms from c to a .

Recall from Example 1.15 that elements of a set X are in one-to-one correspondence with functions $1 \rightarrow X$. Inspired by this, we often abuse our language to say that ‘an element of X ’ is a function $x: 1 \rightarrow X$. More precisely, we could say that x is ‘an element of shape 1’. We then generalize this notion to arrive at the following definition.

Definition 1.59. Let c be an object in a category \mathcal{C} . Given an object a , a *generalized element of a of shape c* is a morphism $e: c \rightarrow a$.

The reader might object: this is just another word for morphism! Nonetheless, we will find it useful to speak and think in these terms, and believe this is enough to justify making such a definition.

This allows us to put slightly more nuance in our mantra that a category is about relationships. Note that, almost by definition, a set is determined by its (generalized) elements (of shape 1). Not all categories have an object that can play the all-seeing role of 1. It is true, however, more democratically: an object in a category is determined by its generalized elements of all shapes.

A nuance is that, as always, we should take into account the role of morphisms. Suppose we have a morphism $f: a \rightarrow b$. We then may obtain sets of generalized elements $\mathcal{C}(c, a)$ and $\mathcal{C}(c, b)$. But more than this, we also obtain a *function*

$$\begin{aligned} f \circ -: \mathcal{C}(c, a) &\longrightarrow \mathcal{C}(c, b); \\ x &\longmapsto f \circ x. \end{aligned}$$

This function describes how f transforms generalized elements of a into generalized elements of b .

To say more precisely what we mean requires the notion of functor, which we shall get to in the next section. We will return to the Yoneda viewpoint in the chapters to come, with a few more tools under our belt. One application of this perspective accessible so far, however, relates to the notion of sameness.

Isomorphisms: when are two objects the same?

Let's think about the category of sets for a moment. Suppose we have the set $\underline{2} = \{0, 1\}$ and the set $B = \{\text{apple}, \text{pear}\}$. Are they the same set? Of course not! One contains numbers, and the other (names of) fruits! And yet, they have something in common: they both have the same number of elements.

How do we express this in categorical terms? In a category, we don't have the ability to count the number of elements in an object – indeed, objects need not even have elements! We're only allowed to talk of objects, morphisms, identities, and composition. But this is enough to express a critically (and categorically) important notion of sameness: isomorphism.

Morphisms in **Set** are functions, and we can define a function $f: \underline{2} \rightarrow B$ that sends 0 to apple and 1 to pear. In the reverse direction, we can define a function $g: B \rightarrow \underline{2}$ sending apple to 0 and pear to 1. These two functions have the special property that, in either order, they compose to the identity: $g \circ f = \text{id}_{\underline{2}}$, $f \circ g = \text{id}_B$.

This means that we can map from $\underline{2}$ to B , and then B back to $\underline{2}$, and vice versa, without losing any information.

Definition 1.60. Let a and b be objects in a category \mathcal{C} . We say that a morphism $f: a \rightarrow b$ is an *isomorphism* if there exists $g: b \rightarrow a$ such that $g \circ f = \text{id}_a$ and $f \circ g = \text{id}_b$. We will call g the *inverse* of f , and sometimes use the notation f^{-1} .

If an isomorphism $f: a \rightarrow b$ exists, we say that the objects a and b are *isomorphic*.

We will spend a lot of time talking about isomorphisms in the category of sets, so they have a special name.

Definition 1.61. A *bijection* is a function that is an isomorphism in the category **Set**. If there is a bijection between sets X and Y , we say the elements of X and Y are in *one-to-one correspondence*.

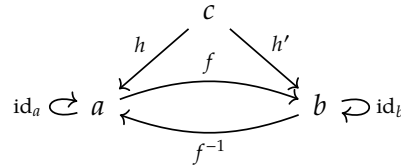
Isomorphic objects are considered indistinguishable within category theory. One way to understand why is to remember that category theory is about relationships, and that isomorphic objects relate in the same way to other objects. Let's talk about this in terms of generalized elements.

Fix a shape c . Recall that a morphism $f: a \rightarrow b$ induces a function from the generalized elements of a to those of b . If f is an isomorphism, then this function is a bijection.

Proposition 1.62. Let $f: a \rightarrow b$ be an isomorphism. Then for all objects c , we have a bijection

$$\mathcal{C}(c, a) \xrightleftharpoons[f^{-1} \circ -]{f \circ -} \mathcal{C}(c, b)$$

To see this, consider the following diagram.



Given a generalized element $h: c \rightarrow a$ of a , $f \circ -$ sends it to $h' = f \circ h$. Conversely, $f^{-1} \circ -$ sends h' to $f^{-1} \circ h' = f^{-1} \circ f \circ h = h$, since f and f^{-1} are inverses.

Exercise 1.63.

1. What is the composite $f \circ -$ followed by $f^{-1} \circ -$?
2. Prove Proposition 1.62.

◇

In other words, Proposition 1.62 says that for each shape c , if a and b are isomorphic, their generalized elements are in one-to-one correspondence. In general, morphisms

that are not isomorphisms may lose information: they need not induce a bijection on generalized elements. Using the intuition of Section 1.2.3, if f is not an isomorphism, then the induce function $f \circ -$ it may collapse two generalized elements of a , or may not cover the entire set of generalized elements of b .

1.4 Categories and Haskell

So far it's been all math and philosophy; let's get to some programming. In the introduction to this chapter, we spoke about how composition is at the core of programming, and saw how we can take two Haskell functions, such as `length` and `words`, and compose them using the `.` syntax to get a new function.

We've also claimed that categories are a mathematical tool for thinking about composition. So perhaps, then, it might be useful to think about Haskell categorically. This indeed is the main lesson of this book: thinking about Haskell categorically allows us to find powerful ways of expressing ourselves using Haskell. But there are questions to be answered. First, what is a Haskell function? And second, if Haskell functions are to be thought of as the morphisms of a category, what are the objects?

1.4.1 The lambda calculus

A Haskell function is based on the notion of mathematical function. Like a mathematical function, it captures the notion of an input–output machine. But while the notion of mathematical function focusses on the semantics of this machine, being simply a description of all input–output pairs, the notion of Haskell function is motivated by the syntax, or how to describe such machines. So a Haskell function is a certain sort of term that can be constructed in the Haskell programming language, that can be thought of as an input–output machine. This brings us to the lambda calculus.

Haskell's syntax is based on the lambda calculus. The lambda calculus comes to us from Alonzo Church's work in mathematical logic in the 1930s. It is a neat, compact language for writing down mathematical functions using two primitive notions: lambda abstraction and function application.

We begin with some variable symbols, like x , y , z , and so on. Sentences in the language of the lambda calculus are called *lambda terms*, and these variables are considered lambda terms themselves.

Given a lambda term A and a variable x , lambda abstraction creates a new lambda term $\lambda x.A$.² This can be thought of as declaring that any instance of x in the lambda term A should be thought of as a variable; in other words, the new lambda term can be thought of as, in some sense, a 'function' where x is the input.

Given two lambda terms A and B , function application creates a new lambda term AB . If we are thinking of A as a function of some variable x , then we think of this as

² λ is the symbol for the Greek letter lamdba.

replacing all occurrences of x in A with B . We also include parentheses in the language, to make the order of construction of a term explicit.

For example, here are some lambda terms that you might see:

$$x \quad xyz \quad \lambda x.x \quad \lambda x.xy \quad \lambda x.(\lambda y.x) \quad (\lambda x.xx)(zy) \quad (\lambda x.yx)(\lambda z.z)$$

The rules of the lambda calculus say that we should consider two lambda terms the same if we can turn one into another by this idea of function application. Take for example the lambda term $(\lambda x.xx)(zy)$. We treat x as a variable, and zy as the value to substitute for this variable. So we say that this term is the same (more technically, ‘evaluates’ or ‘reduces’ to) the term $(zy)(zy)$. Similarly, we have examples:

$$(\lambda x.x)y = y \quad (\lambda x.xy)(zz) = (zz)y \quad (\lambda x.(xy)x)(\lambda z.z) = ((\lambda z.z)y)(\lambda z.z) = y(\lambda z.z).$$

Notice that the lambda term $\lambda x.x$ behaves like an identity function: given some x , it just returns x .

We also consider two lambda terms the same if we can obtain one from the other just by changing the names of the variables; for example $\lambda y.y$ is considered the same as $\lambda x.x$. A technical point is that special care has to be taken to avoid variable name conflicts when substituting. For instance, if A is $\lambda y.(\lambda x.xy)$ and B contains x , in order to avoid unintentionally using the same name for two distinct variables, we have to rename the first x before evaluating AB . This is okay because $\lambda y.(\lambda x.xy)$ is the same as $\lambda y.(\lambda z.zy)$ and so on.³

Exercise 1.64 (Church Booleans). While simple, the lambda calculus is a very expressive language. This makes it a good choice for building a language like Haskell. One frequently useful construction in a programming language is conditional logic: statements such as ‘if t then A else B ’. Here t is some proposition: a statement that evaluates to true or false. Here’s one way conditionals can be modelled in the lambda calculus.

Define the following lambda terms:

$$\text{True} = \lambda x.(\lambda y.x)$$

$$\text{False} = \lambda x.(\lambda y.y)$$

$$\text{Cond} = \lambda x.x$$

To express ‘if t then A else B ’, we write $\text{Cond}((tA)B)$, where t evaluates to True or False.

1. Show that $\text{Cond}((\text{True}A)B) = A$.
2. What is $\text{Cond}((\text{False}A)B)$?
3. Let $\text{And} = \lambda p.(\lambda q.(pq)p)$. What is $(\text{And True})\text{False}$?

³This renaming is intuitively straightforward, but producing unique names for variables when implementing interpreters is a non-trivial problem.

◇

Exercise 1.65 (The Y combinator). The Y combinator is an iconic lambda term whose reduction does not terminate; if we keep trying to evaluate it by substitution, it only grows. It is defined as follows:

$$Y = \lambda f. ((\lambda x. f(xx))(\lambda x. f(xx)))$$

Show that $Yg = g(Yg)$, and hence $Yg = g(g(Yg)) = g(g(\dots(g(Yg))))$ too.

The Y combinator can be used to express recursion in the lambda calculus.

◇

A more formal, and thorough, introduction to the beautiful topic of the lambda calculus can be found in many textbooks on logic and programming languages, including [***]. We'll not go into further detail here. What matters for us, is that Haskell lets us almost directly employ this same notation. For example, to write the identity function in Haskell, we may write:

```
\x -> x
```

The Greek letter lambda is replaced, in ASCII, with a backslash `\` presumably because it looks like the back part of a lambda λ , and the dot is replaced with an arrow.

Importantly, Haskell allows us to give names to terms – “abstracting” them – and so we would define `id` to be the identity function as follows:

```
id = \x -> x
```

This is usually not necessary. The identity function is part of the standard Haskell library called **Prelude**, which is by default loaded into any program. If we allow ourselves to use some other functions from **Prelude**, we can also define functions such as:

```
square = \x -> x^2
implies = \x -> \y -> not y || x
```

Exercise 1.66. Fire up GHCi (see ??), type in these definitions of `square` and `implies`. Then play around a bit. What happens when you type in `square 7`? `square 190517`? `square x`? What about `implies True False`?

Also notice that `id` is already defined. Try for example `id 4`.

◇

Haskell Note 1.67 (Pattern matching). There is an alternative, and by far more common, syntax for function definition, using Haskell’s pattern matching features. In this syntax, we might write the above three functions as:

```
id x = x
square x = x^2
implies y x = not y || x
```

But keep in mind that this is just “syntactic sugar”: the compiler converts the pattern matched syntax down to the more basic lambda syntax.

1.4.2 Types

The lambda calculus is very expressive. In fact it’s Turing complete, which means that any program that can be run on a Turing machine can be also expressed in lambda calculus. A Turing machine is an idealized computer that has access to an infinite tape from which it can read, and to which it can output data. And just like a Turing machine is totally impractical from the programmer’s point of view, so is the lambda calculus. For instance, it’s true that you can encode boolean logic and natural numbers using lambda expressions—this is called Church encoding—but working with this encoding is impossibly tedious and error prone, not to mention inefficient. More significantly, because any lambda term may be ‘applied’ to any other lambda term using function application, it’s okay to apply a number to a function, even if the result is meaningless. Imagine debugging programs written in lambda calculus!⁴

To fix this, even though Haskell is based on the lambda calculus, not every lambda term can be translated to Haskell. For example, consider the lambda term $\lambda x.xx$. This function takes another term A , and applies A to itself: A is fed to itself as its argument! This behavior is hard enough to reason about that it’s not allowed in Haskell. The following code doesn’t compile:

```
ouroboros = \x -> x x
```

So how do we know when we’re allowed to translate a lambda term? Types.

More properly, we should say Haskell’s syntax is based on the *simply-typed lambda calculus*. Every Haskell term is required to have a type. These types ensure Haskell terms and functions can’t be composed willy-nilly: the target type of one must match the source type of the next.

If a term x has type A , we write:

⁴There is also the issue of the Kleene-Rosser paradox (or its simplified version called Curry’s paradox), which shows that untyped lambda calculus is inconsistent; but a little inconsistency never stopped a programming language from being widely accepted.


```
x :: A
```

In fact, in GHCi, you can ask what a term's type is, using the command `:t` or `:type`. For example:

```
Prelude>:t "hi"  
"hi" :: [Char]
```

This tells you that the string `"hi"` is a list of characters.

Exercise 1.68. Fire up GHCi. What are the types of the following terms?

1. `True`
2. `42`
3. `"cat"`
4. `\x -> x x`

◇

Exercise 1.69. Recall the Y combinator from Exercise 1.65. Try to assign the Y combinator a type. What goes wrong?

◇

Types are what they sound like: they describe the type of data a term is meant to represent. In Haskell we have some built-in types, like `Integer`, whose terms are arbitrary precision integers, and `Int`, which is its fixed size companion (with present implementations, usually a 64-bit integer; that is, an integer from -2^{63} to $2^{63} - 1$). In practice, a fixed-size representation is preferred for efficiency reasons, unless you are worried about overflow (e.g., when calculating large factorials). A built-in type we have just seen is `Char`; its terms are unicode characters. A lot of common types, rather than being built-in, are defined in the `Prelude`. Such types include strings `String` and booleans, `Bool`. Much of this course is about how to construct new types from existing types, and categorical ideas like universal constructions help immensely.

Since every Haskell term is required to have a type, sometimes it's your responsibility to tell the Haskell compiler what type your term is. This is done by simply by declaring it, using the same syntax as above. So to tell Haskell compiler you're constructing a term `x` of type `A`, we write:

```
x :: A
```

Note that it's not necessary to declare the type of absolutely every line of code: the Haskell compiler has a powerful type inference system built in, and will be able to work with a bare minimum of type declarations. Having said that, specifying types explicitly will help you catch errors in your code. Experienced Haskell programmers

often start by defining types first and then specifying the implementations, in what is called type-driven development.

Haskell Note 1.70. In Haskell, the names of concrete types start with an upper case letter, e.g. `Int`. The names of type variables, or type parameters, like `a` in the example below, start with lowercase letters.

We defined the identity function `id` without giving a type signature. In fact our definition works for any arbitrary type. We call such a function a *polymorphic* function. We'll talk about polymorphism in more detail in Section 3.5.2. For now, it's enough to know that a polymorphic function is defined for all types.

It's possible, although not necessary, to use the universal quantifier `forall` in the definition of *polymorphic functions*:

```
id :: forall a. a -> a
```

This latter syntax, however, requires the use of the language pragma `ExplicitForAll` (we'll explain this later).^a

^aNote that the period after the `forall` clause is just a separator and has nothing to do with function composition, which we'll discuss shortly.

Remark 1.71 (Types vs sets). Types are very similar to sets, and it frequently will be useful to think of the type declaration `x :: A` as analogous to the set theoretic statement $x \in A$; i.e. that x is an element of the set A . There are some key differences though, which the category theoretic perspective helps us keep straight. The main difference is that, in the world of sets, sets have elements. So given a set A , it's a fair question to ask what its elements are, or whether a particular x is an element of A . Types and terms are the other way around: terms have types.

More formally, $x \in A$ is a predicate, a yes or no question that you can prove or disprove. So when somebody hands you an x , you may ask, is this an element of A ? Conversely, `x :: A` is a judgment, an assertion—it *defines* x as being of the type A , and it doesn't require a proof.

Exercise 1.72. Ponder the differences and similarities between the set of integers \mathbb{Z} , and the Haskell types `Integer` and `Int`. ◇

1.4.3 Haskell functions

In category theory, we write $f: a \rightarrow b$ to mean f is a morphism from a to b . In other words, we are implicitly working in some category \mathcal{C} , and f is an element of the homset:

$$f \in \mathcal{C}(a, b)$$

For example, using this notation in the category **Set**, we write $f: A \rightarrow B$ for a function from a set A to a set B .

In Haskell, we can also declare some terms as Haskell functions, using an almost identical syntax. Given two Haskell types **A** and **B**, there is a **A -> B** whose terms are (Haskell) functions accepting an **A** and producing a **B**. So for example, since the logical operation ‘not’ accepts a boolean and returns a boolean, we have

```
>:t not
not :: Bool -> Bool
```

We call the type **A -> B** the *type of functions from A to B*.

When defining a mathematical function, it’s necessary to first specify the domain and codomain. For example, we might want to define a function f that squares an integer, so we write $f: \mathbb{Z} \rightarrow \mathbb{Z}$. We might then define the function itself: $f(x) = x^2$. Similarly, when defining a Haskell function, it is customary to first declare its type; this is often referred to as its *type signature*. So, for example, to define the function **square**, we might first give the type signature

```
square :: Integer -> Integer
```

A type signature must then be accompanied by an *implementation*. We have to tell the computer how to evaluate a function. We have to write some code that will be executed when the function is called; for example

```
square x = x^2
```

Since the practice of programming in Haskell is all about writing Haskell functions, we’ll spend many of the chapters to come discussing techniques for defining functions. These techniques will be informed by categorical thinking.

Remark 1.73 (Haskell functions vs mathematical functions). As we have already noted, just as types are not the same sets, functions in Haskell are not the same as mathematical functions between sets. But we can make some remarks about their similarities and differences.

One way to compare Haskell functions and mathematical functions is to suppose that the terms of the domain and codomain types form sets, and then to write down the set of input–output pairs for the Haskell function. Let’s call this the *denotation* of the Haskell function. In many cases we care about, the denotation does indeed define a mathematical function, and this denotation often captures the key idea behind the Haskell function. For example, the Haskell type **Bool** can be represented by the set **{True, False}**, which is isomorphic to \mathbb{B} , and the denotation of the Haskell function **not** is then the usual ‘not’ mathematical function.

An important part of the philosophy of Haskell is that denotations of Haskell functions should, as far as possible, be deterministic: the same input value will always produce the same output value. In programming language jargon, we say that Haskell is a *pure, functional* language. But, in a programming language, *producing* a value means executing an algorithm. As long as the algorithm takes reasonable amount of time for every possible input value, it defines a function. But there are recursive algorithms that never terminate, and they don't have a simple set-theoretic denotation.

Exercise 1.74. Consider the mathematical function $g: \mathbb{Z} \rightarrow \mathbb{Z}$, $g(x) = x + 1$.

1. Implement g as a Haskell function `g' :: Integer -> Integer`.
2. Implement g as a Haskell function `g'' :: Int -> Int`.
3. What is `g'' (2^63 - 2)`?
4. What is `g'' (2^63 - 1)`?
5. What are the denotations of `g'` and `g''`? How do they compare to g ?

◇

1.4.4 Composing functions

We now have encountered Haskell functions. What do we do with functions? Compose them! As function composition is such a basic operation, in Haskell we give it almost⁵ the simplest name, a dot:

.

Function composition in Haskell is written in *application order*. This means that the composite of a function `f :: a -> b` and a function `g :: b -> c` is written

```
g . f :: a -> c
```

This is pronounced ‘*g* after *f*’.

Haskell Note 1.75. Note that we’ve written composition as an *infix* operator. This is an operator of two arguments that is written between the two arguments. Another example is addition: we write `4 + 5` to add two numbers.

To use an infix operator as an *outfix* operator—that is, a binary operator that is written before its arguments—we place it in parentheses. So `4 + 5` may also be written `(+) 4 5`.

Composition is itself a Haskell function. What is its type signature? One way to think about it is that it takes a pair of functions, and returns a new function. The Haskell

⁵Function application is the most common operation in Haskell, so its syntax is reduced to the absolute minimum. Just like in the lambda calculus, in most cases a space between two symbols is enough.

definition takes advantage of *currying*. We'll talk much more about this in Chapter **, but in this case, it's a trick that allows us to consider a function of two arguments as a function of the first argument that returns a function of the second argument. This trick gives the type signature:

```
(.) :: (b -> c) -> ((a -> b) -> (a -> c))
```

The function type symbol `->` by default associates to the right, so this is equal to the type

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Note that the rest of the parentheses are essential.

Here's the implementation. Given two functions `g :: b -> c` and `f :: a -> b`, we produce a third function defined as a lambda. The only thing we can do with the functions is to apply them to arguments, and that's what we do:

```
(.) = \g -> \f -> \x -> g (f x)
```

Using pattern matching, we can also write this as

```
(.) g f = \x -> g (f x)
```

The result of calling `f` with the argument `x` is being passed to `g`. Notice the parentheses around `f x`. Without them, `g f x` would be parsed as: `g` is a function of two arguments being called with `f` and `x`. This is because function application is left associative. All this requires some getting used to, but the compiler will flag most inconsistencies.

Composition is just a function with a funny name, `(.)`. You can form a function name using symbols, rather than more traditional strings of letters and digits, by putting parentheses around them. You can then use these symbols, without parentheses, in infix notation. So, in particular, the above can be rewritten:

```
g . f = \x -> g (f x)
```

or, using the syntactic sugar for function definition, simply as:

```
(g . f) x = g (f x)
```

Composition, just like identity before, is a fully polymorphic function, as witnessed by lowercase type arguments. It could be written as:

```
(.) :: forall a b c. (b -> c) -> (a -> b) -> a -> c
(g . f) x = g (f x)
```

Defining new morphisms by composing existing ones is used in the *point-free* style of programming. Here's an example, similar to the one in the introduction.

Example 1.76. Let's call the function `words` with a string `"Hello world!"`. Here's how it's done:

```
Prelude> words "Hello world!"
["Hello","world!"]
```

The result is a comma-separated list of words with spaces removed.

Now let's call the function `concat` to concatenate the entries of the list:

```
Prelude> concat ["Hello","world!"]
"Helloworld!"
```

We can compose the two functions using the composition operator, which is just a period `"."` between the two functions. Let's apply it to the original string:

```
Prelude> (concat . words) "Hello world!"
"Helloworld!"
```

Haskell Note 1.77 (Binding and parentheses). Notice the use of parentheses. Without them, the line:

```
length . words "Hello world!"
```

would result in an error, because function application binds stronger than the composition operator. In effect, this would be equivalent to:

```
length . (words "Hello world!")
```

The compiler would complain with a somewhat cryptic error message. This is because the composition operator expects a function as a second argument, and what we are passing it is a list of strings.

Exercise 1.78. Consider the mathematical function $f: \mathbb{Z} \rightarrow \mathbb{Z}$ that sends an integer to its square, $f(x) := x^2$, and the function $g: \mathbb{Z} \rightarrow \mathbb{Z}$ sends an integer to its successor, $g(x) := x + 1$.

1. Write analogous Haskell functions to f and g , implementing them both as functions **Int** \rightarrow **Int**.
2. Let $h := f \circ g$. What is $h(2)$? Compute it both by hand and using your Haskell functions.
3. Let $i := g \circ f$. What is $i(2)$? Again, check that your hand and Haskell agree.
4. Let $j := f \circ g \circ f$. What is $j(2)$? ◇

1.4.5 Thinking categorically about Haskell

A category, recall, has objects, morphisms, identities, and composition. In more detail, the morphisms go between objects, every object has an identity morphism, and if we have two morphisms, one of whose domain is the other's codomain, we can compose them.

Haskell has types, functions, identities, and composition. The functions go between types, every type has an identity function, and if the codomain of a function is the domain type of another, then we can compose them.

This leads to an analogy: Haskell is like a category. This analogy is very powerful, and this book is devoted to exploring the consequences. What is especially remarkable is that this is only an analogy. Haskell is a programming language, a tool for expressing computations, that is supported a community of developers and practitioners. Haskell is not one thing, but is a specification and many implementations, and both are under constant change. Yet this analogy is useful no matter which version of Haskell one is working with.

Even if Haskell is not a mathematical object, it is tempting to try to make this analogy precise, and people often speak of a fictitious category **Hask**, of Haskell types and functions. We won't delve deeply into the difficulties of making such a construction, but we can briefly point out some difficulties. These mostly center around the question of when two Haskell functions are the same. On one end of the spectrum, we might say a Haskell function is exactly the code used to write it. But then `id . f` and `f` have different code, and so our category doesn't obey the unit laws. Closer to what we mean when we talk about a Haskell function, one might say two Haskell functions are equal if they have the same denotation. But the concept of denotation itself is not well defined, and making it so poses difficulties.

In the end, perhaps a useful comparison is the Haskell type **Int**. This is usually (but not always) presently implemented as 64-bit integers, and hence has values -2^{63} to $2^{63} - 1$. This is decidedly not the set of integers: for example it does not contain the number 2^{63} , and if we try to write it, we get -2^{63} . But it is enough like the integers—e.g. there's a 0, and a 1, and certain laws like the unit law for addition and multiplication hold—that it's often useful to think about it *as though* **Int** were the set of integers. Moreover, **Int** is often more practical to use than the arbitrary precision integer type **Integer**, because people have found that we get much better performance when we truncate.

In Exercise 1.78 we used the type `Int` to help us express ideas and compute results about the integers, and used the integers to help us think about what our Haskell programs do. As we go forward, we'll similarly learn to use Haskell to help us express categorical ideas, and category theory to help us think about what our Haskell programs do.

Universal constructions and the algebra of types

2.1 Constructing datatypes

We have seen that types play a fundamental role in thinking about programming from a categorical point of view. Haskell, like many typed programming languages, starts with a collection of base types, such as a type **Char** of characters, **Int** of integers, or **Bool** of truth values. But programming is about using some simple building blocks to construct rich and expressive behavior, and often these types are not enough. To make a language more expressive, we introduce ways of making new types from old, introducing operations on types such as taking the product, the sum, or the exponential. These operations define an ‘algebra’ of types.

For example, let’s say we want to construct a type representing a standard deck of French playing cards. Each card has a rank (which is either a number from 2 to 10 or one of jack, queen, king, or ace) as well as a suit (diamonds \diamond , clubs \clubsuit , hearts \heartsuit , or spades \spadesuit). For simplicity, let’s say we’re given types **Rank** and **Suit**. The type **Card** of cards should simply have values that consist of a rank *and* a suit, such as the two of diamonds ($2\diamond$), or the ace of spades ($A\spadesuit$). A type constructed from other types in the way **Card** was constructed from **Rank** and **Suit** is known as a *product type*.

In mathematical notation, we would use the same multiplication symbol that we use for arithmetic: $\text{Card} = \text{Rank} \times \text{Suit}$. But in Haskell, we pun on the way terms of this type are represented—namely as pairs (r, s) —and thus denote the product type by $(\text{Rank}, \text{Suit})$.

Product types are useful whenever we wish to construct a type whose values consist of a value of one type *and* a value of another. Other examples include the type $(\text{String}, \text{Int})$ of name and age pairs, the type (Int, Int) representing locations on a two-dimensional grid, or the type $(\text{Int}, \text{Int}, \text{Int})$ representing points on a three-dimensional grid.

An additional, and critical, part of a product type are its data accessors. Given a card, we should be able to extract both its rank and suit; that is, we should have two functions

```
getRank :: Card -> Rank
getSuit :: Card -> Suit
```

Another common way of constructing new types is a sum type; these are types that can take values from either one type *or* another. For example, if you're setting up a two-factor authentication scheme, you might give the user a chance to select a phone number or an email. Using the plus sign from arithmetic to represent a sum type, we might then write **TwoFA = PhoneNumber + EmailAddress**.

Sum types also come with associated functions. For example, when designing a login page, depending on user input we might call one of the two functions

```
phone2FA :: PhoneNumber -> TwoFA
email2FA :: EmailAddress -> TwoFA
```

Finally, given two types **A** and **B** we can look at the type **A -> B** of all functions from the first to the second. This would generally be called the *function type* in a programming context, and the *exponential type* in category theory context; see Exercise 1.14 for why. Passing functions as data is essential in functional programming and very useful in general.

So we have three ways of constructing new types from old, and they may seem quite different. But in fact they all have something very deep in common, namely they are all characterized by *universal properties*. This is the main purpose of this chapter.

As we keep repeating, a category describes a world of objects and their relationships. Just as with people, certain objects are made special, or characterized by how they relate to others. For example, in Haskell the unit type **()** has the special property that for every other type **A**, there is a unique function to it, namely **\a -> ()**. Moreover, up to isomorphism, the unit type is the only type with this property. We say that the unit type is defined by its *universal property*.

Defining objects by their universal properties is a common theme throughout mathematics and programming. Next in this chapter we'll get to some examples of universal constructions that are particularly important: terminal objects, products, initial objects, and coproducts. These are all very well modelled in Haskell, and we'll discuss the different ways in which they can be implemented and used.

2.2 Universal constructions

2.2.1 Terminal objects

One of the simplest examples of a universal object in a category is a terminal object.

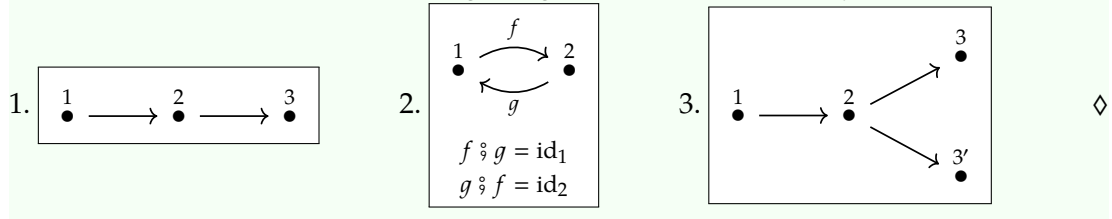
Definition 2.1. Let \mathcal{C} be a category. An object 1 in \mathcal{C} is called a terminal object if for every object a there is a unique morphism $!_a: a \rightarrow 1$.

In what sense is such an object 1 universal? Well, it's universal in the sense that it's defined by a special property with respect to *all* other objects: *for all* objects a in \mathcal{C} , there is a *unique* morphism to it.

People sometimes refer to the maps of the form $!$ as “bang”, maybe because of how “!” is sometimes used in English to indicate something immediate or extreme. The map to a terminal object is both immediate and extreme: there is exactly one way “bang!” to go from c to the terminal object.

Exercise 2.2. What does the universality of 1 tell us about morphisms from 1 to 1 ? \diamond

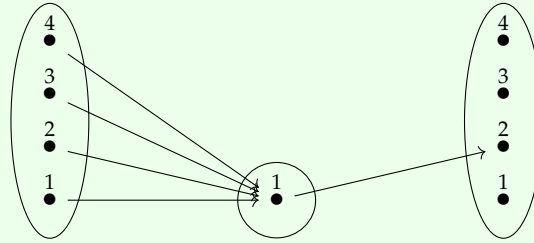
Exercise 2.3. Which of the following categories has a terminal object?



Example 2.4 (Many terminal objects). Any set containing exactly one element is a terminal object in **Set**. For example, the sets $\{1\}$, $\{57\}$, and $\{*\}$ are all terminal objects. Why? Suppose we have another set X . Then what are the functions, for example, from X to $\{57\}$? For each element $x \in X$, we need give an element of $\{57\}$. But there is only one element of $\{57\}$, the element we call 57. So the only way to define a function $f: X \rightarrow \{57\}$ is to define $f(x) = 57$ for all $x \in X$.

Example 2.5 (Constant morphisms). Suppose that \mathcal{C} has a terminal object 1 , and for any object c , let $!_c: c \rightarrow 1$ denote the unique morphism. We say a morphism $f: c \rightarrow d$ is *constant* if there exists a morphism $g: 1 \rightarrow d$ such that $f = g \circ !_c$. It's easy to see how it

works in **Set**:



Said another way, a morphism is constant if it “factors through” the terminal object.

Exercise 2.6.

1. Name a poset with a terminal element.
2. Name a poset without a terminal element.

◇

Example 2.7. In a poset, a terminal object is a greatest element. This is because in a poset we interpret a morphism from a to b as the relation $a \leq b$. Suppose we have a terminal object 1. Then, by definition, for any object a , we have a morphism from a to 1 and hence $a \leq 1$. Thus 1 is a greatest element. For example, in the poset of subsets of $\{x, y, z\}$, ordered by inclusion, the total subset $\{x, y, z\}$ itself is a terminal object (and in fact the only one).

We have already seen that **Set** has many terminal objects. It’s also possible for a category not to have any terminal objects. For example, the poset \mathbb{N} of natural numbers ordered by the usual \leq ordering has no greatest element, and hence no terminal object.

Exercise 2.8. Does the category of partial functions have a terminal object? If so, what is it? ◇

Note that we’ve sometimes said “the” terminal object, but we give a definition for “a” terminal object. It so happens that any universal object is unique up to unique isomorphism, and so we consider this strong enough to simply say “the”. If you and I give two different terminal objects, we know at least there will be a unique morphism that describes how to transform one into the other, and vice versa, without losing or gaining any information.

Exercise 2.9 (All terminal objects are isomorphic). Let x and y be terminal objects in a category. Show that they are isomorphic. In fact, show that there is a unique isomorphism between them. (Hint: why is there a morphism $x \rightarrow y$? How about $y \rightarrow x$? Can you name a morphism $x \rightarrow x$? How many other morphisms are there $x \rightarrow x$?) ◇

Definition 2.10. Let \mathcal{C} be a category with a terminal object 1 . A *global element* of an object a is a morphism $1 \rightarrow a$.

Exercise 2.11. For each of the following statements about an object c , decide if it is equivalent to the statement “ c is terminal”. If so, provide a proof, if not, find a category in which they are not.

1. The object c has a unique global element.
2. There exists another object d for which there is a unique generalized element of shape d in c .
3. For all objects d , there is a unique generalized element of each shape d in c

◇

Exercise 2.12. Recall the category **Cat** from Definition 3.25. Show that the category **1** is the terminal category in **Cat**.

◇

2.2.2 Initial objects

A terminal object lies at the ‘end’ of a category: every object points to it. One might also look for objects at the ‘beginning’ of a category. These are called *initial objects*.

Definition 2.13. Let \mathcal{C} be a category. An object 0 in \mathcal{C} is called an *initial object* if for every object a there is a unique morphism $!_a: 0 \rightarrow a$.

Example 2.14. In a poset, an initial object is a least element.

Exercise 2.15. In **Set**, the initial object is the empty set, \emptyset . Why? (Since the empty set has zero elements, we often write 0 for an initial object in a category.)

◇

Exercise 2.16. Someone tells you an initial object in \mathcal{C} is just a terminal object in \mathcal{C}^{op} . What do they mean? Are they correct?

◇

Exercise 2.17. Show that a trivial category **0** with no objects or morphisms is initial in **Cat**. (First, convince yourself that this is really a category.)

◇

The initial object is defined by its *mapping out* property. As a shape, you may think of it as “a shape with no shape” or the emptiness. The definition tells you that you can see this shape in every other object and in itself.

We'll find this mapping out property of initial objects very useful in the next chapter, where we show how to construct each recursive type as an initial object in a certain category, and use its mapping out property to define recursive functions.

Exercise 2.18. Show that if there is a morphism from the terminal object to the initial object (in other words, if the initial object has a global element) then the two objects are isomorphic (such an object is then called the *zero object*.) \diamond

2.2.3 Products

So far we've considered terminal objects, which are defined using a mapping-in property; and initial objects, which have a mapping-out property. Objects with a mapping-in property are called *limits*: a terminal object is a kind of limit. Things with a mapping-out property are called *colimits*: an initial object is a kind of colimit. In this section we'll discuss products, which are a kind of limit; you'll soon hear us talking about a mapping-in property.

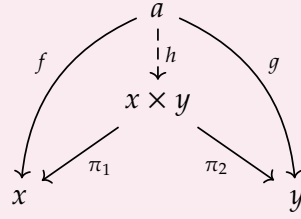
Given two sets X and Y , we can always construct their cartesian product $X \times Y$. This is the set whose elements are pairs (x, y) , where $x \in X$ and $y \in Y$. This is a useful object to construct; to return to the playing cards example of the introduction to this chapter, the set of playing cards is simply the cartesian product of the set of ranks and the set of suits.

The construction of the cartesian product refers to elements, and thus cannot be performed in an arbitrary category. Indeed, in an arbitrary category \mathcal{C} with a terminal object 1 , even if we replaced elements with global elements, the set of pairs of global elements is still a set, not an object of \mathcal{C} .

The categorical way to generalize the cartesian product of sets is to think (surprise!) in terms of relationships. Note that the cartesian product of sets has a special mapping-in property: if we want to define a function $f: A \rightarrow X \times Y$ from some set A to a product, it is enough to define a function $f_X: A \rightarrow X$ and $f_Y: A \rightarrow Y$. Then we can let $f(a)$ be the pair $(f_X(a), f_Y(a)) \in X \times Y$. That is, a morphism $A \rightarrow X \times Y$ in **Set** is the same as a pair of morphisms $A \rightarrow X$ and $A \rightarrow Y$. So products are about pairs, but they're about pairs of *morphisms*, rather than elements.

Definition 2.19. Let x and y be objects in a category \mathcal{C} . A *product* of x and y consists of three things: an object, denoted $x \times y$ and two morphisms $\pi_1: x \times y \rightarrow x$ and $\pi_2: x \times y \rightarrow y$, with the following *universal property*: For any other such three things, i.e. for any object a and morphisms $f: a \rightarrow x$ and $g: a \rightarrow y$, there is a unique morphism

$h: a \rightarrow x \times y$ such that the following diagram commutes:



Often we just refer to $x \times y$ as the product of x and y . We call the morphisms π_1 and π_2 *projection maps*. We will frequently denote h by $h = \langle f, g \rangle$.

Example 2.20. A product in a poset is a greatest lower bound. For example, consider the poset of natural numbers ordered by division. Then the product of 12 and 27 in this poset is 3.

Exercise 2.21.

1. In the poset (\mathbb{N}, \leq) , where $5 \leq 6$, what is simplest way to think about the product of m and n ?
2. Write down a poset for which there are two elements that don't have a product. \diamond

Example 2.22. The product of two sets X and Y in **Set** is exactly the cartesian product $X \times Y$, with the projection functions $\pi_1: X \times Y \rightarrow X$ and $\pi_2: X \times Y \rightarrow Y$ defined respectively by $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$.

To prove this, given functions $f: A \rightarrow X$ and $g: A \rightarrow Y$, we must show there is a unique function $h: A \rightarrow X \times Y$ such that $\pi_1 \circ h = f$ and $\pi_2 \circ h = g$. To see there exists a h with this property, define $h(a) = (f(a), g(a))$. Note that $\pi_1(h(a)) = \pi_1(f(a), g(a)) = f(a)$, and similarly for g , so this h does obey the required laws.

To see that this h is the unique morphism with this property, recall that $\pi_1(x, y) = x$, and $\pi_2(x, y) = y$. Fix a , and let x and y be such that $h(a) = (x, y)$. Then $x = \pi_1(x, y) = \pi_1(h(a)) = f(a)$, and similarly $y = g(a)$. So $h(a) = (f(a), g(a))$.

Example 2.23. Suppose some category \mathcal{C} has a terminal object 1 . Then for any object x , $1 \times x$ is isomorphic to x .

The isomorphism is witnessed by morphisms $(!_x, \text{id}_x): x \rightarrow 1 \times x$, and $\pi_2: 1 \times x \rightarrow x$. The universal property of the product shows that both $(!_x, \text{id}_x) \circ \pi_2 = \text{id}_{1 \times x}$ and $\pi_2 \circ (!_x, \text{id}_x) = \text{id}_x$.

Similarly, $x \times 1 \cong x$. We thus say that the terminal object is a unit for the product.

Exercise 2.24. Let's work out an explicit example in the category **Set**. Choose sets X, Y, Z and functions $f: X \rightarrow Y$ and $g: X \rightarrow Z$. What is $h = (f, g)$? Do you think this is a good notation? \diamond

Note that, just like terminal objects, products are defined by a mapping-in property: the product $x \times y$ receives a unique map from every object that has maps to x and y . We could say that $x \times y$ is a “one-stop shop for morphisms to x and to y ”. If you want a morphism to x and to y , you just need a morphism to $x \times y$.

An object with maps to x and y is an example of what is known as a *cone*, and in fact products can be understood as terminal objects in a certain category of cones. A corollary of this is that since products in \mathcal{C} are an example of terminal objects in some related category, they are also unique up to unique isomorphism by Exercise 2.9.

Why is this useful for thinking about programming; what is the computational content of this idea? The problem it solves is one of defining a function $h: a \rightarrow x \times y$. The product can be used to *decompose* this problem into two simpler problems: one function $a \rightarrow x$ and another $a \rightarrow y$. Indeed, the universal property of the product implies that we have a one-to-one correspondence (i.e. an isomorphism of sets)

$$\mathcal{C}(a, x \times y) \cong \mathcal{C}(a, x) \times \mathcal{C}(a, y). \quad (2.25)$$

This means that in order to specify any morphism in the set on the left (i.e. one morphism $a \rightarrow x \times y$), it's enough to name its corresponding element in the set on the right (i.e. a pair consisting of a morphism $a \rightarrow x$ and a morphism $a \rightarrow y$). This sort of decomposition is a common theme in using universal constructions for structuring programs.

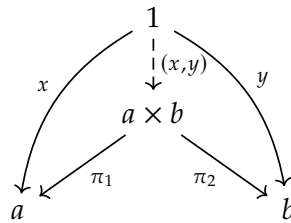
Another viewpoint on products is via global and generalised elements.

Proposition 2.26. The set of global elements of a product is the cartesian product of the sets of global elements of the factors.

The proof is simply to choose $a = 1$ in Eq. (2.25):

$$\mathcal{C}(1, x \times y) \cong \mathcal{C}(1, x) \times \mathcal{C}(1, y).$$

The left hand side is the set of global elements of $x \times y$, while the right hand side is the cartesian product of the set of global elements of x and that of y .



Exercise 2.27. Given a fixed shape c , show that a c -shaped (generalized) element of $a \times b$ is the same as a pair consisting of a c -shaped element of a and a c -shaped element of b . \diamond

Exercise 2.28 (Products of morphisms). Suppose we have morphisms $f: a \rightarrow b$ and $g: c \rightarrow d$, such that products $a \times c$ and $b \times d$ exist. Then we can construct a morphism $f \times g: a \times c \rightarrow b \times d$, known as the product of f and g .

This morphism defined as follows: $f \times g := (f \circ \pi_1, g \circ \pi_2)$. Note that, as usual, the map into the product $b \times d$ is defined by pairing a map into b with a map into d .

Show that in the category **Set**, the function $f \times g$ sends a pair $(x, y) \in a \times c$ to the pair $(f(x), g(y))$. \diamond

Exercise 2.29 (n -ary products). Given three objects x, y, z , one can define their ternary (three-fold) product $P_{x,y,z}$ to be a one-stop shop for maps to all three. That is, $P_{x,y,z}$ comes with morphisms $\pi_1: P_{x,y,z} \rightarrow x$ and $\pi_2: P_{x,y,z} \rightarrow y$ and $\pi_3: (x, y, z) \rightarrow z$, and for any other P' equipped with morphisms $p_1: P' \rightarrow x$, $p_2: P' \rightarrow y$, and $p_3: P' \rightarrow z$, there is a unique morphism $(p_1, p_2, p_3): P' \rightarrow P_{x,y,z}$ such that $p_i = \pi_i \circ (p_1, p_2, p_3)$ for each $i \in \{1, 2, 3\}$.

1. Suppose that \mathcal{C} has a terminal object. Show that if it has ternary products then it also has (binary, i.e. ordinary) products.
2. Show that if \mathcal{C} has binary products then it also has ternary products. That is, show that $P_{x,y,z}$ is isomorphic to $(x \times y) \times z$. \diamond

If a category \mathcal{C} has a terminal object and (binary) products, then it has n -ary products for all n : the terminal object is like the 0-ary product. We would say that \mathcal{C} has *all finite products*.

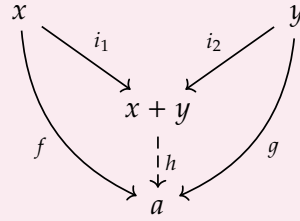
Definition 2.30. We say that a category is a *cartesian category* if it has all finite products.

2.2.4 Coproducts

We may dualize the definition of product to obtain the definition of coproduct. In contrast to the mapping-in property that defines products and terminal objects, coproducts (like initial objects) are defined by a mapping-out property.

Definition 2.31. Let x and y be objects in a category \mathcal{C} . A *coproduct* of x and y is an object, denoted $x + y$, together with morphisms $i_1: x \rightarrow x + y$ and $i_2: y \rightarrow x + y$, such that for any object a and morphisms $f: x \rightarrow a$ and $g: y \rightarrow a$, there is a unique

morphism $h: x + y \rightarrow a$ such that the following diagram commutes:



We call the morphisms i_1 and i_2 *inclusion maps*. We will frequently denote h by $h = [f, g]$.

Remark 2.32. Often we just refer to just the object $x + y$ as the coproduct of x and y , even though the coproduct technically also includes the inclusion maps i_1, i_2 .

Remark 2.33. Recall Proposition 2.26: a map from an object a into the product $x \times y$ is the same as a map $a \rightarrow x$ and a map $a \rightarrow y$. Dually, a map out of the coproduct $x + y$ to a is the same as a map $x \rightarrow a$ and a map $y \rightarrow a$.

This again has consequences for writing code. Suppose we want to define a function $h: a + b \rightarrow c$. The coproduct *decomposes* the task into two simpler problems of defining two functions $a \rightarrow c$ and $b \rightarrow c$.

Example 2.34. The coproduct of two sets X and Y in **Set** is exactly the disjoint union $X \sqcup Y$, which contains a unique element for every x in X and a unique element for every y in Y . Since it is the coproduct, we also write this set $X + Y$. The inclusion map $\iota_1: X \rightarrow X \sqcup Y$ simply sends x in X to its corresponding element of $X \sqcup Y$, and similarly for $\iota_2: Y \rightarrow X \sqcup Y$.

Exercise 2.35. What is the coproduct of two elements in a poset? ◇

Example 2.36. Just as a terminal object is a unit for products, an initial object 0 is a unit for coproducts. That is, for any object x in a category with coproducts $0 + x \cong x + 0 \cong x$.

Definition 2.37. We say that a category is *cocartesian* if it has an initial object and every pair of objects has a coproduct. We say that a category is *bicartesian* if it is both cartesian and cocartesian.

2.3 Type constructors

Let's bring this thinking to Haskell. We've learned about four universal constructions: terminal objects, initial objects, products, and coproducts. How can these help us think

about programming? In fact, since these four universal constructions are so central to the construction of useful types, corresponding constructions are built-in to Haskell and the Prelude. To describe these, we'll need to talk a bit about constructing types more generally.

2.3.1 Type constructors

In Haskell, we define types using the syntax

```
data TypeConstructor = DataConstructors
```

For example, the **Prelude** contains the following definition of the type **Bool**:

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

Type variables may also be used; these must begin with a lower case letter. For example, in the following type definition, **a** is a type variable.

```
data WithString a = MakeWithString a String
```

In these examples, **Bool** and **WithString** are called *type constructors*, since they construct new types. The type variable **a** means that a type must be given to the **WithString** constructor in order to construct a type; for example, given the type **Int**, we get a type **WithString Int**, which is a type whose terms are integers with strings.

Type constructors define new types. To produce terms (or *data*) of these types, we must use their *data constructors*. The type **Bool** has two data constructors, **True** and **False**. Thus these are the two terms of type **Bool**. A term of type **WithString a** must be constructed using the constructor **MakeWithString**, together with a term of type **a** and a **string**. Thus this data constructor is a function

```
MakeWithString :: a -> String -> WithString a
```

Normally, function names start with a lowercase letter, but data constructors are an exception: they are functions whose names start with an uppercase letter. Since each type has its own data constructors, the compiler can use these data constructors (like **MakeWithString**) as keywords that indicate the type of the term to follow: "whenever I see **MakeWithString** x y, I check what type **x** has, say **Int**, check that **y** is a **String**, and then and I'll know that **MakeWithString** x y has type **WithString Int**."

Here's how the constructor may be used to construct a value of the type **WithString Int**:

```
charles :: WithString Int
charles = MakeWithString 135 "bananas"
```

The first line declares the type of the name¹ `charles` as the type obtained by applying the type constructor `WithString` to `Int`. The second line declares `charles` to have a specific value, obtained by passing `135 "bananas"` to the data constructor `MakeWithString`.

Once a value of the type `WithString` `a` is constructed, it is immutable. This again is part of what makes Haskell a purely functional language. Because of immutability, every value “remembers” the way it was created: what data constructor was used and what value(s) were passed to it. This is why a value of type `WithString` `a` can always be deconstructed. This deconstruction is called *pattern matching*. Here’s how it’s done:

```
extractString :: WithString a -> String
extractString (MakeWithString x y) = y
```

The pattern that is matched here is `MakeWithString x y`. It names the data constructor `MakeWithString` and the values passed to it `x` and `y`. When applied to the above example,

```
extractString charles
```

will produce the string `"bananas"`.

Exercise 2.38. Why do we write the pattern `MakeWithString x y` in parentheses? What happens if we omit them and write the expression `extractString MakeWithString x y`?

◇

2.3.2 Unit and void

The unit type If we think of Haskell as a category, certain types will have certain ‘universal properties’, which can play an important role in structuring our programs.

A singleton type is a type with a unique, unparametrized data constructor. For example, we could define the type

```
data Terminal = UniqueValue
```

Just any set with one element is a terminal object in the category **Set**, this type behaves like a terminal object in Haskell: for every other type, there is only one way to define a (total) function to **Terminal**.

It’s useful to have a canonical choice of terminal object, so the Haskell supplies a special singleton type, known as the *unit type*. The unit type is already defined in any implementation of Haskell, but if we were to define it, we would write:

¹Informally, we and others might say this declares the type of the *variable* `charles`. Strictly speaking there are no *variables* in Haskell, since the name implies variability or mutation. A declaration like this *binds* a name to an expression. Traditionally, though, we call such a name a variable.

```
data () = ()
```

This definition contains a pun, of a sort common in Haskell type definitions. The symbol `()` (empty tuple) is being used in two, distinct ways: first as a type constructor (ie. as the name of a type), and second as a data constructor (ie. as the name of a term of that type). In fact, `()` is the only term of the type `()`, so this abuse of notation is not so bad, assuming you can tell terms from types (the Haskell compiler can).

Note also that the name `()` is special syntax: ordinary, user-defined types cannot contain parentheses.

We'll see that the unit type is very useful when defining functions with *side-effects*, a way to escape the purely functional aspects of Haskell.

Exercise 2.39. For all types `a`, define a function

```
bang :: a -> ()
```

Explain why the function you defined does the same thing as any other (total) function you could possibly have defined. ◇

The void type The idea of an initial object inspires the empty Haskell type. While again we might give it any name we like, the library `Data.Void` gives a standard implementation with the name `Void`. The syntax for defining the type void is simple.

```
data Void
```

Note that it has no data constructors, because it's empty! We can't construct a term of type `Void`.

There is a correspondence, called the Curry-Howard correspondence, saying that each type T can be interpreted as a logical statement. The statement is true if the type is *inhabited*; in other words, there is a way to produce a term of that type. Since `Void` has no data constructor, there is no way to create a term of type `Void`—the logical statement it represents is false.

Now in logic, the Latin phrase *ex falso quodlibet* ("from falsehood, anything follows") refers to the principle that if you can prove a contradiction, or *false*, anything follows. An initial object thus behaves a bit like a false statement, in the sense that any other object receives a morphism from it: given a proof of `Void`, you can prove anything. For fun, we'll refer the unique morphism from `Void` to any type `a` as `exFalso`

```
exFalso :: Void -> a
exFalso x = undefined
```

The funny thing here is that whenever the program tries to evaluate `undefined`, it will terminate immediately with an error message. We are safe, though, because there is no way `exFalse` will be executed. For that, one would have to provide it with an argument: a term of type `Void`, and no such term exists!

The library `Data.Void` calls this function `absurd :: Void -> a`.

Remark 2.40 (Function equality). You might be wondering why we consider `absurd` to be the only function from `Void` to, say, `Int`. For example, what about this function:

```
vTo42 :: Void -> Int
vTo42 x = 42
```

Is this function different from `absurd`, as instantiated for `Int`? Function equality is tricky. What we use here is called *extensional* equality: two functions are equal if, for every argument, they produce the same result. Or, conversely, to prove that two functions are different, you have to provide a value for the argument on which these two functions differ. Obviously, if the argument type is `Void`, you can't do that.

2.3.3 Tuple types

The next universal construction we introduced was the product. Let's talk about how to implement these in Haskell. This will be the first step in creating a library of useful functions that we will use throughout this book.

Given two types `a` and `b`, we want to construct a type that behaves like their product in our idealized Haskell category. To construct a new type, we use a type constructor:

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

The type constructor says that we have a new type, `Pair a b`, while the value constructor says that to construct a value of the type `Pair a b`, we specify a value of type `a` and one of type `b`.

For example, let's take `a = Int` and `b = Bool`.

```
p :: Pair Int Bool
p = MkPair 5 True
```

Recall from Definition 2.19 that a product consists of three things: in addition to the product object, we have two projection maps. Just as in `Set`, in our Haskell version of a product these maps extract the components of a pair. We can define these as follows.

```
proj1 :: Pair a b -> a
proj1 (MkPair a b) = a
```

```
proj2 :: Pair a b -> b
proj2 (MkPair a b) = b
```

Note that we've used pattern matching to define these functions. Because we know that a term of type **Pair** *a b* must be constructed using the constructor **MkPair**, we may define a function out of **Pair** *a b* not by referring to the argument of the function as some monolithic variable *x*, but by referring to it as the construction **MkPair** *a b*, and hence giving us a name for both the first (*a*) and second (*b*) components of the pair.

Also note the punning between the type and term levels: we see the variable *a* in the type **Pair** *a b* as well as the expression **MkPair** *a b*. In fact, these are two different *as*, but closely related: we use this notation because the expression *a* is a term of type *a*. This may be confusing at first, but becomes very efficient to read. And of course, since the language (and compiler) separates the type level from the term level, there is no ambiguity in meaning.

Exercise 2.41. Write a program that defines the value *x* = ("well done!", **True**) of type **Pair String Bool**, and then projects out the first component of the pair. ◇

Haskell Note 2.42 (Built-in pair type). Just as for the void type, product types are so useful that they're implemented in the Haskell base with a special syntax, that mimics the pair notation traditionally used for the cartesian product of sets. One might think of it as defined using the following code:

```
data (a,b) = (a,b)    --doesn't actually compile, but it's the idea
```

Here we have again taken our type–data level puns to the extreme, and given the type constructor and data constructor the same name *(,)*. The type constructor *(a,b)* should be thought of as analogous to **Pair** *a b*, while the data constructor *(a,b)* is analogous to **MkPair** *a b*.

The built-in pair type comes with projection maps

```
fst :: (a,b) -> a
fst (x,y) = x

snd :: (a,b) -> b
snd (x,y) = y
```

From now on when discussing pairs we'll default to the above hard-coded syntax *(a,b)* rather than **Pair** *a b*.

Haskell Note 2.43 (Tuple types). More generally, this tupling syntax can be used with any number of entries, implementing the idea of n -ary products. For example, we can use the type `(Int, Int, String)`. The default projection maps `fst` and `snd` are only defined for the pair types `(a, b)`, however.

Haskell Note 2.44 (`data`, `type`, `newtype`). In addition to the keyword `data` for defining new types, Haskell provides the keywords `type` and `newtype` for renaming types.

The keyword `type` allows creation of type synonyms: two names for the same type, that are treated as identical by the compiler. For example, if we define

```
type AnimalName = String
```

then `"elephant" :: String` and `"elephant" :: AnimalName` both typecheck, and any function that accepts a `String` will accept an `AnimalName`.

The keyword `newtype` allows the creation of a type isomorphic to an existing type: one must specify a single data constructor. For example, we might define

```
newtype AnimalName = MakeAnimalName String
```

This is similar in use to `data` with a single data constructor, but ensures that the runtime representation of an `AnimalName` is identical to that of a `String`, and hence has consequences for the efficiency of code.

Example 2.45 (Cards). In a standard 52 card deck of French playing cards, each card has a rank and a suit. This is a pair type! We might define a type `Card` in Haskell as follows:

```
type Card = (Rank, Suit)
```

```
newtype Rank = R Int
```

```
rank :: Rank -> Int
rank (R n) = n
```

`Suit` will be defined in Example 2.53, after we have discussed sum types.

Using the universal property of products The universal property of the product says that, given a type `c`, a pair of functions `c -> a` and `c -> b` is the same as a single

function `c -> (a, b)`. In other words, the types `(c -> a, c -> b)` and `(c -> (a, b))` are isomorphic.

This isomorphism is very useful to us as programmers: it says we may construct a function into a pair just by solving the simpler problems of constructing functions into the factors! It's helpful to explicitly have available the isomorphism between the two types. In one direction, we have the function `tuple`.

```
tuple :: (c -> a, c -> b) -> (c -> (a, b))
tuple (f, g) = \c -> (f c, g c)
```

We may also implement the function `tuple` using pattern matching as follows:

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

In the other direction, we can define the function `untuple`.

```
untuple :: (c -> (a, b)) -> (c -> a, c -> b)
untuple h = (\c -> fst (h c), \c -> snd (h c))
```

Exercise 2.46. Show that `tuple` and `untuple` are inverses, and hence the types `(c -> a, c -> b)` and `c -> (a, b)` are isomorphic. \diamond

Haskell Note 2.47. The standard library `Control.Arrow` already includes the function `tuple`, defined as an infix operator `&&&`. This operator is defined with the type signature

```
(&&&) :: (c->a) -> (c->b) -> (c -> (a, b))
```

Note that this type signature is different from the one we used for `tuple`; we say this is the *curried* version of the function. We'll return to this topic later in this chapter.

Record syntax Product types are ubiquitous in programming, so Haskell provides a lot of syntactic sugar to make them easier to use. The simplest product, a pair, is relatively easy to deal with. It's easy to construct, and easy to access the two fields, either by pattern matching, or through the two projections, `proj1` and `proj2`. But as you keep adding components, or *fields*, it gets harder to keep track of their purpose, since they are only distinguished by their position in the tuple. Record syntax for product types lets you assign names to their components. These names are sometimes called *accessors* or *selectors*.

For example, instead of defining the **Pair** datatype and then defining functions **proj1** and **proj2** as above, we can use the more concise syntax:

```
data Pair a b = MkPair { proj1 :: a, proj2 :: b }
```

More typically, since the accessors are thought of as names for fields as well as functions for accessing them, one chooses names accordingly:

```
data Pair a b = MkPair { fst :: a, snd :: b }
```

Record syntax also allows you to "modify" individual fields. In a functional language, this means creating a new version of a data structure with particular fields given new values. For instance, to increment the first component of a pair, we could define a function

```
incrFst :: Pair Int String -> Pair Int String
incrFst p = p { fst = fst p + 1 }
```

Here we are defining **incrFst p**, which has the same components as **p** except for the **fst** field, which is set to **fst p + 1** (the value of **fst p** plus one).

Example 2.48. Here's a more elaborate example of record syntax that we will later use to implement Solitaire solver in Haskell. The game state is a record with three fields

```
data Game = Game { founds  :: Foundations
                  , cells   :: Cells
                  , tableau :: Tableau }
```

The fields can be accessed through their selectors, as in

```
game :: Game    --suppose game is already given
cs = cells game --then we can get its cells this way
```

We can also implement three *setters*,^a which use the record update syntax

```
putFounds game fs = game { founds = fs }
putCells  game cs = game { cells  = cs }
putTableau game ts = game { tableau = ts }
```

As usual for product types, you can construct a record either using the record syntax with named fields, or by providing values for all fields in correct order

```
newGame :: [Card] -> Game      -- we'll implement this in an appendix
newGame deck = Game newFoundations
                  newCells
                  (newTableau deck) -- newtableau takes an argument
```

^aOne might also do this “setting” and “getting” using the Haskell lens library.

2.3.4 Sum types

To model finite coproducts in Haskell, there are sum types.

One may construct generalized elements of a coproduct $a + b$ by either constructing a generalized element of a , then composing with i_1 , OR by constructing a generalized element of b , then composing with i_2 . In fact, in **Set**, all global elements of the coproduct may be constructed in this way. Because of this, the built in syntax in Haskell for implementing coproducts makes use of the vertical line $|$, which in computer science is traditionally associated with OR.

```
data Coproduct a b = Incl1 a | Incl2 b
```

Types constructed using $|$ are known as *sum types*. A more traditional name for **Coproduct** in Haskell is simply **Either**:

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

Here **Left** and **Right** correspond to the two injections i_1 and i_2 .

```
Left :: a -> Either a b
Right :: b -> Either a b
```

An instance of **Either** a b may be created using either data constructor. For example:

```
x :: Either Int Bool
x = Left 42
y :: Either Int Bool
y = Right True
```

Thus, in analogy with the coproduct in **Set**, the terms of type **Either Int Bool** are just the terms of type **Int** together with the terms of type **Bool**.

Recall from Remark 2.33 that maps out of a coproduct $a + b$ the same as a pair of maps, one out of a , and one out of b . We define functions out of a sum type by pattern matching. Here are two possible syntaxes:

```

h :: Either a b -> c
h eab = case eab of
    Left  a -> foo a
    Right b -> bar a

```

```

h :: Either a b -> c
h (Left a)  = foo a
h (Right b) = bar b

```

Exercise 2.49. What are the types of the functions `foo` and `bar` above? ◇

Example 2.50 (Maybe). Note that a constructor in a sum type need not have any type variables. For example, the `Maybe` type constructor, defined in the `Prelude`, has the following definition:

```

data Maybe a = Nothing | Just a

```

Here `Nothing` is a constructor for a singleton type, with the unique term `Nothing`. Thinking of this as a terminal object 1, the type `Maybe a` models the coproduct $1 + a$.

These data constructors have the type signatures

```

Nothing :: Maybe a
Just    :: a -> Maybe a

```

Thinking of `Nothing` as a map from a singleton type, for example `()`, to `Maybe a`, these data constructors correspond to the two inclusions into the coproduct $1 + a$.

In effect, `Maybe a` adds a single, new term, `Nothing`, to the type `a`. This is useful for type safe definitions of operations that are not totally defined. For example, the integer division function `div` does not always return an integer: it returns an exception if we try to divide by zero. However, we can make this function total by sending division by zero to `Nothing`:

```

safeDiv :: Int -> Int -> Maybe Int
safeDiv m n =
    if n == 0
    then Nothing
    else Just (div m n)

```

Note that the return type is now **Maybe Int** instead of **Int**.

Example 2.51. The set \mathbb{B} of Boolean values, has two elements, **true** and **false**. As a set, it is isomorphic to any set with two values, so one way to think of it is simply as the coproduct $1 + 1$.

Similarly, the type **Bool** has two values, **True** and **False**, and one way to think of it is as the coproduct of two singleton types, one with data constructor **True**, and the other with data constructor **False**. The **Prelude** indeed defines **Bool** in this way:

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

Exercise 2.52. Implement **Bool2**, a type isomorphic to **Bool**, using **Either** and the unit type **()**. Make sure you define **true** and **false**. (Why must these begin with lower case letters?) ◇

Example 2.53. As promised in Example 2.45, we can now implement the type **Suit** from our running cards example. This is simply a sum type with four data constructors:

```
data Suit = Club | Diamond | Heart | Spade
```

Using the universal property of the coproduct In analogy with the function **tuple** for products, there is a convenient function in Haskell that encapsulates the universal property of the coproduct (Definition 2.31):

```
either :: (a->c) -> (b->c) -> (Either a b -> c)
either f g =
  \e -> case e of
    Left a -> f a
    Right b -> g b
```

The **either** function has an inverse

```
unEither :: (Either a b -> c) -> (a->c, b->c)
unEither h = (h . Left, h . Right)
```

The above style is called *point free* because it doesn't use variables; it's equivalent to the following

```
unEither h = (\a -> h (Left a), \b -> h (Right b))
```

2.4 Exponentials and function types

So far we've seen universal properties in terms of mapping in and mapping out properties. An important computational consequence, is that these properties describe bijections between certain homsets. For example, the product object $x \times y$ obeys the bijection

$$\mathcal{C}(a, x \times y) \cong \mathcal{C}(a, x) \times \mathcal{C}(a, y),$$

while the coproduct object $x + y$ obeys the bijection

$$\mathcal{C}(x + y, a) \cong \mathcal{C}(x, a) \times \mathcal{C}(y, a).$$

Indeed, it's possible to define the product and coproduct as the objects for which these bijections exist in a certain so-called 'natural' sense.

When a category has products, another important sort of universal object is an *exponential object*. The exponential object from x to y is written y^x . It obeys the bijection

$$\mathcal{C}(x \times a, y) \cong \mathcal{C}(a, y^x).$$

In this section we'll introduce the exponential object, and show how a critically important sort of types in Haskell, *function types*, is characterized by the properties of the exponential.

2.4.1 Interlude: Distributivity

Before we move on from products and coproducts, it will be fun to point out something about how they relate. The following is something from high school, but this time we are doing arithmetic on types.

Recall from Definition 2.37 that a bicartesian category is a category with finite products and coproducts. The category **Set** is an example, and we can think of Haskell as an example too. For any objects a, b, c in any bicartesian category, there is a morphism as follows:

$$\delta: (a \times c) + (b \times c) \rightarrow (a + b) \times c. \quad (2.54)$$

The existence of such an morphism follows from the universal properties of the product and coproduct. Remember that coproducts really understand maps out of them and products really understand maps into them, and that's exactly what we need: a map out of a coproduct and into a product.

Exercise 2.55. Categorical thinking translates into code. Do you see how to construct the morphism δ ? To check, see if you can implement the corresponding function in Haskell.

1. Implement functions of the type `a -> Either a b` and `b -> Either a + b`
2. Implement functions of the type `(a, c) -> (Either a b, c)` and `(b, c) -> (Either a b, c)`.

3. Implement `dist :: Either (a, c) (b, c) -> (Either a b, c)`. ◇

In high school we learned about the equation $(a \times c) + (b \times c) = (a + b) \times c$; we say that multiplication *distributes* over addition. The corresponding categorical fact is that in the category **Set**, δ is not just a morphism, but an isomorphism. Similarly, in Haskell, we can construct an inverse as follows

```
undist :: (Either a b, c) -> Either (a, c) (b, c)
undist (Left a, c) = Left (a, c)
undist (Right b, c) = Right (b, c)
```

This is not true, however, in an arbitrary bicartesian category. It requires that our category have one more sort of universal construction: exponential objects.

2.4.2 Exponential objects

Recall our central metaphor: Haskell types are objects in a category, and Haskell functions are morphisms. We've also seen, however, higher order functions, such as the composition operator, " \circ ". Higher order functions can take a function as an input, or give a function as an output. How do we explain these? What are the domain and codomain types of a higher order function?

To save the situation, we need an object in our category that represents all morphisms from one given object to another. This works perfectly well in the category **Set** of sets. Functions between two sets A, B form a set, the homset $\mathbf{Set}(A, B)$, and that set just already is an object in **Set** again. This object is denoted B^A , for reasons explained by Exercise 1.14.

In an arbitrary category \mathcal{C} , morphisms between two objects also form a set, the homset $\mathcal{C}(A, B)$, but a set is not generally an object in \mathcal{C} . If \mathcal{C} were to be similar to **Set** in the sense that the morphisms $A \rightarrow B$ pool up into a single object of \mathcal{C} , an 'internal version' of the homset, what property would we want this internal homset to have?

The answer to this question is far from obvious, but here's one critical property of function sets. For any element $f \in B^A$ of the function set from A to B , and for any element $a \in A$, there is an element $f(a) \in B$: we can evaluate f at a . Thus we have a function

$$\text{eval}_{A,B}: B^A \times A \rightarrow B$$

Moreover, for any other set X and function $e: X \times A \rightarrow B$, there is a unique function $e': X \rightarrow B^A$ such that the following diagram commutes:

$$\begin{array}{ccc} X \times A & \xrightarrow{e' \times \text{id}_A} & B^A \times A \\ & \searrow e & \downarrow \text{eval}_{A,B} \\ & & B \end{array} \quad (2.56)$$

In other words, this diagram states that $e(x, a) = \text{eval}_{A,B}(e'(x), a)$ for all $x \in X$ and $a \in A$.

This is what universal constructions are all about. We have to make sure that any other object X that pretends to be an (A, B) -function object (by providing an evaluation-like morphism $f: a \times b \rightarrow c$) is already captured by B^A .

Definition 2.57. Let $A, B \in \mathcal{C}$ be objects in a cartesian category. An object B^A , equipped with a morphism $\text{eval}_{A,B}: B^A \times A \rightarrow B$, is called an *exponential* or *function object* for morphisms A to B if it has the following universal property:

- for any object X and morphism $e: X \times A \rightarrow B$, there exists a unique map $e': X \rightarrow B^A$ such that the diagram (2.56) commutes.

Remark 2.58. This is quite an abstract definition! The most important consequence is that an exponential object B^A induces the following bijection between the following hom-sets, for any object X :

$$\mathcal{C}(X \times A, B) \cong \mathcal{C}(X, B^A). \quad (2.59)$$

The forward map, which we denote

$$\text{curry}: \mathcal{C}(X \times A, B) \rightarrow \mathcal{C}(X, B^A),$$

is simply the function that takes e to e' , as described by the universal property. Its inverse,

$$\text{uncurry}: \mathcal{C}(X, B^A) \rightarrow \mathcal{C}(X \times A, B)$$

sends $f: X \rightarrow B^A$ to $\text{eval}_{A,B} \circ (f \times \text{id}_A): X \times A \rightarrow B$.

Eq. (2.59) tells you almost everything you need to know about exponential objects. In fact, it's almost an alternative definition of 'exponential object'; one only needs to add that these bijections should be 'natural' in X . We'll explain what this means in the next chapter.

Definition 2.60. A category with a terminal object, and both a product and an exponential defined for every pair of objects is called *cartesian closed*.

Example 2.61. As discussed, the category **Set** is a cartesian closed category, with the exponential object B^A the set of functions from A to B .

The function curry acts as follows. First, it accepts a function $f: X \times A \rightarrow B$, and returns a function $\text{curry}(f): X \rightarrow B^A$. Thus, evaluating $\text{curry}(f)$ at x gives a function $\text{curry}(f)(x): A \rightarrow B$. This function is the function that sends $a \in A$ to $f(x, a)$. In other words, it takes a function of two variables x and a , and returns a function of one variable, x , that itself returns a function of one variable, a .

Remark 2.62. This process of taking a function of two variables and turning it into a higher order function of one variable is known as *currying*. This process, and hence the map ‘curry’, is named after the logician Haskell Curry. As you might have guessed, the programming language Haskell is also named after Haskell Curry. Exponential objects are a very important part of Haskell!

Cartesian closed categories are important in modeling programming languages and the lambda calculus in particular.

Exercise 2.63. Try to define the exponential object by replacing a product with a coproduct. What goes wrong? Try implementing it in Haskell. \diamond

Remark 2.64. We’ve said that the universal property of the exponential object B^A in a cartesian closed category \mathcal{C} makes it act like an internal version of the set of morphisms $A \rightarrow B$. But how do you get the morphisms out of it? Then answer is that you use the terminal object $1 \in \mathcal{C}$.

Consider the sequence of isomorphisms

$$\mathcal{C}(A, B) \cong \mathcal{C}(1 \times A, B) \cong \mathcal{C}(1, B^A) \quad (2.65)$$

The first isomorphism comes from the fact that the terminal object is a unit for the product: $1 \times A \cong A$ (see Example 2.23). This means that given a morphism $A \rightarrow B$, we may turn it into a morphism from $1 \times A \rightarrow B$ by precomposing with π_2 , and conversely by precomposing with its inverse $(!_A, \text{id}_A)$. This gives the first isomorphism in Eq. (2.65). The second isomorphism is given by the universal property of the exponential object (Eq. (2.59)). This gives us the following slogan.

The maps $A \rightarrow B$ are the global elements of the exponential object B^A .

Exercise 2.66 (Evaluation and pairing). The equation Eq. (2.59) holds for all A, B , and X . A standard categorical trick is to plug in certain special values; the results often have special properties.

1. If we plug in $X = B^A$, we get

$$\mathcal{C}(B^A \times A, B) \cong \mathcal{C}(B^A, B^A).$$

Now there’s a special element of the right-hand side, namely id . Applying the function $\text{uncurry}: \mathcal{C}(B^A, B^A) \rightarrow \mathcal{C}(B^A \times A, B)$ to id , we find that

$$\text{eval} = \text{uncurry}(\text{id}): B^A \times A \rightarrow B.$$

Explain why this is true.

2. Similarly, if we plug in $B = X \times A$, we get

$$\mathcal{C}(X \times A, X \times A) \cong \mathcal{C}(X, (X \times A)^A).$$

Now there's a special element of the left-hand side, namely id . This lets us make the definition

$$\text{pair} := \text{curry}(\text{id}): X \rightarrow (X \times A)^A.$$

Show that in the category **Set**, this function pair maps $x \in X$ to the function $\text{pair}(x): A \rightarrow X \times A$ defined by sending $a \in A$ to (x, a) .

◇

2.4.3 Function types, and currying in Haskell

When working in Haskell, the role of the exponential object for functions a to b is played by the *function type* $a \rightarrow b$. We've already seen this type signature. Understanding exponential objects means that we can use their insights to make powerful use of function types.

The properties of exponential objects revolve around the functions `curry`, `uncurry`, `eval`, and `pair`. Analogues of `curry` and `uncurry` are defined in **Prelude**, implemented like so:

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

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

As evident from these implementations, the essence of currying is a bijection between morphisms that take a product as an argument and morphisms that produce a function as output:

A function of a pair of arguments is equivalent to a function returning a function.

Haskell defaults to curried functions. That is, instead of using multi-arity functions, the standard style in Haskell is to write single argument functions that return functions. For example, recall the type signature of `(.)` from Section 1.4.4:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

This is a function that accepts a $b \rightarrow c$ and returns a function $(a \rightarrow b) \rightarrow a \rightarrow c$. (Recall that by Haskell's parenthesization conventions, the type $a \rightarrow b \rightarrow c$ is equal to the parenthesized type $a \rightarrow (b \rightarrow c)$.)

An elegant consequence of defaulting to curried functions is that parentheses are not needed to pass multiple arguments to a function. For example, consider a function `f'` of type signature

```
f' :: (a,b) -> c
```

To evaluate this at some pair `(a,b)`, we must write `f' (a,b)`. On the other hand, suppose we use the curried version

```
f :: a -> b -> c
f = curry f'
```

Then to evaluate `f' (a,b)`, we may simply write `f a b`. Since function application binds strongest, this is read as `(f a) b`—ie. apply the function `f a :: b -> c` to `b`. This gives the desired result.

The curried-by-default convention is also the reason why we can *partially apply* functions: for example, simply write `f a` for a function `f a :: b -> c`.

Haskell Note 2.67. In Haskell, the partial application of a binary operator is called a *section*. For example, consider the binary operator `(+)`, which takes two numbers and adds them.

The partial application `(+) 2` of `(+)` to `2` produces a function of one argument that adds `2` to the argument. For infix operations such as `+`, we may also write this section using the syntactic sugar `(2+)`.

Exercise 2.68. Fire up `ghci` and play around with the operator `+`.

- What is the type signature of `(+)`?
- What is the type signature of `uncurry (+)`?
- What is the type signature of `(+) 2`?

You'll notice the symbol `Num a =>` in these type signatures. This says that, in the following type expression, the type `a` should be understood as a type of numbers. We'll learn in more detail what this means when we discuss type classes in the next chapter. ◇

Exercise 2.69. From our work in Exercise 2.66, we can immediately define analogues of `eval` and `pair`:

```
Prelude> eval = uncurry id  --Haskell knows you mean id of type b -> c
Prelude> pair = curry id    --Haskell knows you mean id of type (b, c)
Prelude> :t eval
(b -> c, b) -> c
```

```
Prelude> :t pair
b -> c -> (b, c)
```

These two functions are extremely versatile. For example, we saw in Remark 2.58 that to uncurry $h: a \rightarrow c^b$, we simply compose with evaluation:

$$a \times b \xrightarrow{h \times \text{id}_b} c^b \times b \xrightarrow{\text{eval}} c. \quad (2.70)$$

1. Write a function `uncurry'` in Haskell that behaves the same as the `Prelude` function `uncurry`, but only using `eval`, following Eq. (2.70).
2. Similarly, write `curry'` using `pair`. ◇

Haskell Note 2.71. We usually see the curried version of the evaluator as an infix operator

```
($) :: (a -> b) -> a -> b
```

While this might simply seem like the identity function, because of way precedence of function application is parsed, this operator is often useful for making code a bit more readable. It is used between a function name and an expression that evaluates to its arguments. For instance, instead of writing `f (a + b)`, we can avoid one level of parentheses and write

```
f $ a + b
```

Example 2.72. Recall our discussion of the distributivity of products and coproducts in Section 2.4.1. Now that we've discussed exponential objects and function types in Haskell, we can use these tools to shed light on our inverse to the canonical map δ from Eq. (2.54).

We wish to construct a morphism

$$(a + b) \times c \rightarrow (a \times c) + (b \times c).$$

In any cartesian closed category, we can get a morphism of this type by giving a morphism

$$a + b \rightarrow ((a \times c) + (b \times c))^c$$

But this is a map out of a coproduct, and coproducts 'understand' maps out. This perspective leads to the following code:

```
helper :: Either a b -> (c -> Either (a, c) (b, c))
helper (Left a) = \c -> Left (a, c)
```

```

helper (Right b) = \c -> Right (b, c)
undist = uncurry helper

```

Exercise 2.73. For fun, we can also write the function `undist` in a point-free style:

```

undist :: (Either a b, c) -> Either (a, c) (b, c)
undist = uncurry (either (curry Left) (curry Right))

```

1. Explain what is going on here and why it does the same thing as the implementation in Example 2.72.
2. Explain in your own words why you need a bicartesian category in order to express the distributivity morphism from Eq. (2.54).
3. Explain in your own words why you need a bicartesian *closed* category in order for that map to be an isomorphism.

Note that these kinds of point-free implementations are notoriously difficult to obtain and, in general, are difficult to analyze, so they are avoided in practice. \diamond

Exercise 2.74. When we get to the continuations monad in ??, we'll need functions that involve a bunch of currying and evaluation. As a special case, implement functions of the following types.

1. `r :: a -> ((a -> Int) -> Int)`
2. `j :: (((a -> Int) -> Int) -> Int) -> Int -> ((a -> Int) -> Int).`

Hint: use `r` on the type `a -> Int`. \diamond

Functors, natural transformations, and type polymorphism

3.1 Relationships, relationships, relationships

Categories are about arrows from one object to another and how to compose them; category theory thus emphasizes the viewpoint that arrows—the ways objects relate to each other—are important. So one might ask: if relationships are so important, how do we describe how categories relate to each other?

The principle notion of relationship between two categories is called a *functor*. A functor from a category \mathcal{C} to a category \mathcal{D} consists of function from the objects of \mathcal{C} to the objects of \mathcal{D} , and a function from that arrows of \mathcal{C} to the arrows of \mathcal{D} , that obeys certain properties. These properties describe what it means to preserve the basic categorical structure.

So we have categories and functors. But, you might further dare to ask: if relationships are so important, how do we describe how *functors* relate to each other? In fact, in a beautiful example of categorical thinking, this is what led to the discovery of categories themselves. That is, mathematicians discovered categories not on their own, and not even by thinking about the way they relate to each other, but by thinking about how their *relationships* relate to each other.

Relationships between functors are known as *natural transformations*. In this chapter, we'll formally introduce functors and natural transformations, together with a bunch of useful examples to help you think about them.

This chapter is largely one that lays the mathematical groundwork for the coming chapters: we can't talk about algebras, monads, monoidal categories, or profunctors without first talking about functors, and for many of these topics natural transformations are needed too. But beyond looking at these concepts mathematically and exploring how they can be expressed in Haskell, we'll sneak some Haskell lessons in too. Functors and natural transformations will provide a good playbox to explore type

polymorphism in Haskell, in various guises.

A value is polymorphic when it can be given multiple types. For example, the definition

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

writes the identity function once and for all; we need not write a different identity function for **Int** and for **String**. Since it works for all types, as expressed by the type variable, or parameter, **a**, we say that **id** is *parametrically polymorphic*.

A second type of polymorphism is *ad hoc polymorphism*. This refers to a name that is given multiple different implementations, which are deployed depending on the type of data passed to it. For example, the operation (+) can be used to add two values which are both of type **Int**, **Integer**, **Float**, amongst other types. We express this ability using the notion of a *type class*. The operation (+) has type signature

```
(+) :: Num a => a -> a -> a
```

The expression **Num a** declares that in the following type expression, the type variable **a** must be of type class **Num**. As we'll see, type classes are a useful device for organizing the data of algebraic structures within Haskell. In particular, we'll meet the **Functor** type class, which will help organize the construction of functors implemented in Haskell. We'll also see that the notion of a functor in Haskell is not exactly the same as the mathematical definition, hence examining the ways that categorical thinking informs practice.

3.2 Functors

The basic premise of category theory is that you should be able to learn everything you need to know about X's by looking at mappings between X's. Look at the interface rather than the implementation.

But in some sense, it seems like we've been inadvertently breaking this rule when talking about categories themselves as the X's. We keep talking about objects and morphisms—the internal “implementation details” of a category—rather than some sort of mappings between categories, like the above paragraph says we should. In this section we introduce an appropriate notion of mapping between categories, called functors.

3.2.1 Definition

Here's the slogan:

A functor from one category to another is a structure-preserving map of objects and morphisms.

A functor $F: \mathcal{C} \rightarrow \mathcal{D}$ takes each object a in \mathcal{C} to an object that we denote $F a$ in \mathcal{D} . Similarly, it takes each morphism $f: a \rightarrow b$ in \mathcal{C} to a morphism in \mathcal{D} denoted by $F f$

$$F f: F a \rightarrow F b$$

What does it mean that a functor preserves the structure of a category? The structure of a category is defined by identity morphisms and composition, so a functor must map identity morphisms to identity morphisms

$$F \text{id}_a = \text{id}_{F a}$$

and composition to composition

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

This gives the following definition.

Definition 3.1. A functor $F: \mathcal{C} \rightarrow \mathcal{D}$ consists of two constituents:

- (i) a function $F: \text{Ob } \mathcal{C} \rightarrow \text{Ob } \mathcal{D}$, and
- (ii) for every a, b in $\text{Ob } \mathcal{C}$, a function $F_{a,b}: \mathcal{C}(a, b) \rightarrow \mathcal{D}(F a, F b)$.

These constituents are subject to two constraints. A functor

- (a) **Preserves identity:** for any $a \in \text{Ob } \mathcal{C}$, the equation $F \text{id}_a = \text{id}_{F a}$ holds.
- (b) **Preserves composition:** for any $f: a \rightarrow b$, $g: b \rightarrow c$, the equation $F g \circ F f = F(g \circ f)$ holds.

There are a few special words for various commonly used types of functor.

Definition 3.2. Let \mathcal{C} and \mathcal{D} be categories.

- A *contravariant functor* $\mathcal{C} \rightarrow \mathcal{D}$ is a functor $\mathcal{C}^{\text{op}} \rightarrow \mathcal{D}$.
- An *endofunctor* on \mathcal{C} is a functor $F: \mathcal{C} \rightarrow \mathcal{C}$.
- A *bifunctor* on \mathcal{C} is a functor $\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{D}$.
- A *profunctor* on \mathcal{C} is a functor $\mathcal{C} \times \mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}$.

Exercise 3.3. Just to play around with the idea and warm up a bit, write down all functors between the two-object categories **2**, **Disc(2)**, and **I** (as defined in Examples 1.35, 1.36, and 1.41). How many did you find? \diamond

3.2.2 Examples of functors

In Section 1.3.3 we gave a wide variety of examples of categories, including discrete categories, monoids, preorders, and the key example **Set**. The goal was to demonstrate

the flexibility of the concept, and provide a library of tools to test your understanding against. In Haskell, we'll mostly be interested in functors that behave like endofunctors on **Set**. But we'll again begin with a wide variety of examples, all of which are very useful!

Monoid homomorphisms

Recall that a category with one object is called a monoid (Definition 1.46). A functor between monoids is called a *monoid homomorphism*. Let's explore this idea.

Let \mathcal{M} and \mathcal{N} be categories, both with a single object called $*$. This means that they are monoids. Let's use the notation (M, e, \diamond) and (N, o, \bullet) for the monoids \mathcal{M} and \mathcal{N} respectively. Recall that $M = \mathcal{M}(*, *)$, $e = \text{id}_*$, and \diamond is composition in \mathcal{M} , and similarly for \mathcal{N} .

A functor $F: \mathcal{M} \rightarrow \mathcal{N}$ consists of:

- (i) A function $F: \text{Ob } \mathcal{M} = \{*\} \rightarrow \text{Ob } \mathcal{N} = \{*\}$. Note there is only one such function – it sends $*$ to $*$ – and so we can ignore this part of the definition.
- (ii) For every pair of objects (a, b) of \mathcal{M} , a function $F_{a,b}: \mathcal{M}(a, b) \rightarrow \mathcal{N}(Fa, Fb)$. Since there is only one object $*$ of \mathcal{M} , this means we only need one function: $F_{*,*}: \mathcal{M}(*, *) \rightarrow \mathcal{N}(*, *)$.

So the data of a functor is simply a function, let's just call it f , of type $f: M \rightarrow N$.

This function must obey:

- (a) Preserves identities: $f(e) = o$
- (b) Preserves composition: for all $m, m' \in M$, we have $f(m \diamond m') = f(m) \bullet f(m')$.

Monoid homomorphisms are very common.

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

Exercise 3.5. Is the function $\text{is_even}: \mathbb{Z} \rightarrow \mathbb{B}$ that maps even numbers to **true** and odd numbers to **false** a monoid homomorphism, from \mathbb{Z}_\times to \mathbb{B}_{AND} ? If so, prove it. If not, can you find other monoid operations on the sets \mathbb{Z} and \mathbb{B} that make is_even a monoid homomorphism? \diamond

Exercise 3.6. Consider the monoids $\mathbb{R}_+ = (\mathbb{R}, 0, +)$ and $\mathbb{R}_\times = (\mathbb{R}, 1, \times)$.

1. Is the function $x \mapsto 3x$ a monoid homomorphism $\mathbb{R}_+ \rightarrow \mathbb{R}_\times$? Prove it or explain why not.
2. Is the function $x \mapsto e^x$ a monoid homomorphism $\mathbb{R}_+ \rightarrow \mathbb{R}_\times$? Prove it or explain

why not.

3. Can you think of another monoid homomorphism $\mathbb{R}_+ \rightarrow \mathbb{R}_\times$?

◇

Monotone maps

Recall that a category with at most one morphism between any two objects is a preorder (Definition 1.51). A functor between preorders is called a *monotone map*.

Let \mathcal{P} and \mathcal{Q} be preorders, considered as categories. We'll write $P = \text{Ob } \mathcal{P}$, $Q = \text{Ob } \mathcal{Q}$, and $p \leq p'$ if there is a morphism between objects p and p' in either category; as usual, we won't bother giving the morphism a name since it's unique. A functor $F: \mathcal{P} \rightarrow \mathcal{Q}$ consists of

(i) A function $F: P \rightarrow Q$.

(ii) For every pair (p, p') in P a function $F_{p,p'}: \mathcal{P}(p, p') \rightarrow \mathcal{Q}(Fp, Fp')$.

Since \mathcal{P} and \mathcal{Q} are preorders, $\mathcal{P}(p, p')$ and $\mathcal{Q}(Fp, Fp')$ many contain at most one element. This means that if a function $F_{p,p'}$ exists, it's unique. So the key is to ask: when does such a function exist? The only problem arises when $\mathcal{P}(p, p')$ contains an element, but $\mathcal{Q}(Fp, Fp')$ doesn't. That is, if $p \leq p'$, then to define a functor, we must have $Fp \leq Fp'$. We thus say that F must *preserve the order*.

Of course, for F to be a functor, it must also preserve identities and composition. We'll leave it to you to explain why.

Exercise 3.7. Check that any function $F: P \rightarrow Q$ that preserves the order also preserves identities and composition. ◇

Example 3.8. Let **Items** be a preorder of items ordered by quality, and \mathbb{R}_\leq be the usual preorder on real numbers. Then the function **price**: **Items** $\rightarrow \mathbb{R}_\leq$ that sends an item to its price should be a monotone map: if item a is better quality than item b , it should be worth more money.

Exercise 3.9 (Upper sets). Let $\mathcal{P} = (P, \leq)$ be a preorder. An *upper set* (or an *upwards-closed set*) in \mathcal{P} is a subset $S \subseteq P$ such that if $p \in S$ and $p \leq p'$, then $p' \in S$.

Let $\mathbb{B}_\leq = (\mathbb{B}, \leq)$ be the usual preorder on the Booleans, with **true** $>$ **false**. In this exercise we'll show that a monotone map $F: \mathcal{P} \rightarrow \mathbb{B}_\leq$ is the same as an *upper set*.

1. Given a function $F: P \rightarrow \mathbb{B}$, write S_F for the subset of P consisting of all elements that are sent to **true**. Show that S_F is an upper set.
2. Suppose we have an upper set S of (P, \geq) . Let $F_S: P \rightarrow \mathbb{B}$ be the function sending p to **true** if $p \in S$, and to **false** otherwise. Show that F_S is a monotone map.

◇

Exercise 3.10. Let \mathbb{Z}_{\geq} be the usual preorder on the integers. Are the following functions monotone?

1. $f(n) = n$
2. $g(n) = 2n$
3. $h(n) = n + 2$
4. $i(n) = n^2$
5. $j(n) = 0$

◇

Some standard constructions

The following examples are a bit more abstract, but they work for almost any category. It's often helpful to have them on hand.

Example 3.11 (The identity functor). Given any category \mathcal{C} , we may define the *identity functor* $\text{id}_{\mathcal{C}}: \mathcal{C} \rightarrow \mathcal{C}$. For the functor $\text{id}_{\mathcal{C}}: \text{Ob } \mathcal{C} \rightarrow \text{Ob } \mathcal{C}$, we simply pick the identity mapping; so on objects, this functor sends each object A to itself. The same is true for morphisms: the identity functor $\text{id}_{\mathcal{C}}$ sends each morphism to itself. In particular, this choice of mappings preserves composition and identity in \mathcal{C} ; see Exercise 3.12.

Exercise 3.12. Check that $\text{id}_{\mathcal{C}}$ really does preserve identities and composition. ◇

Example 3.13 (Constant functors). Let \mathcal{C} and \mathcal{D} be categories. Given any object d in \mathcal{D} , we can define the *constant functor* $K_d: \mathcal{C} \rightarrow \mathcal{D}$ on d . This functor sends *every* object of \mathcal{C} to $d \in \text{Ob } \mathcal{D}$, and *every* morphism of \mathcal{C} to the identity morphism on d .

For example, we could talk about the constant functor $K_2: \mathbf{Set} \rightarrow \mathbb{N}$, where \mathbb{N} is the preorder of natural numbers. This maps every set to the natural number 2. It's a bit of a strange idea, but it's a functor!

Exercise 3.14. Show that any constant functor $K_d: \mathcal{C} \rightarrow \mathcal{D}$ obeys the two functor laws: preservation of composition and preservation of identities. ◇

Example 3.15 (Products give functors). Let \mathcal{C} be a cartesian category (ie. it has finite products). Then given any object a , we can define a functor $- \times a: \mathcal{C} \rightarrow \mathcal{C}$. This functor sends any object c to $c \times a$, and any morphism $f: c \rightarrow d$ to the morphism $f \times \text{id}_a: c \times a \rightarrow d \times a$.

For example, there is a functor $- \times \mathbb{Z}: \mathbf{Set} \rightarrow \mathbf{Set}$ that sends any set X to $X \times \mathbb{Z}$.

We can also define a bifunctor $- \times -: \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$. This functor maps a pair of objects

(c, d) in \mathcal{C} (that is, an single object in the product category $\mathcal{C} \times \mathcal{C}$) to the object $c \times d$ in \mathcal{C} , and a morphism $(f, g): (c, d) \rightarrow (c', d')$ in $\mathcal{C} \times \mathcal{C}$ to the morphism $f \times g: c \times \rightarrow c' \times d'$ in \mathcal{C} .

3.2.3 Functors and shapes

Remember our discussion about interpreting functions $f: a \rightarrow b$ as modeling set a inside set b ? We talked about a singleton set as a shape that embodies the idea of an element: its models in b are the elements of b . Functors also can be understood as modelling shapes, but this time shapes of categories inside other categories.

The categorical analogue of a singleton set is the discrete category $\mathbf{1}$ on a one element set. We first met this category in Example 1.33, and saw it was part of the family of discrete categories in Example 1.35. The category $\mathbf{1}$ has one object and no arrows except for the identity:

$$\mathbf{1} = \boxed{\begin{array}{c} \text{id}_1 \\ \curvearrowright \\ 1 \end{array}}$$

In analogy with the singleton set, a functor from $\mathbf{1}$ to any category \mathcal{C} picks out an object in \mathcal{C} . As such, it is a *walking object*. It has no other structure but what's necessary to illustrate the idea of an object. It's like when we say that somebody is a walking encyclopedia of Star Wars trivia. What we mean is that this person has no other distinguishing qualities besides being an expert on Star Wars. This is usually an unfair description of a living and breathing person, but in category theory we often encounter such patterns that have just the right set of features to embody a particular idea and literally nothing else.

Exercise 3.16. Let \mathcal{C} be an arbitrary category.

1. Show that there is a one-to-one correspondence between objects of \mathcal{C} and functors $\mathbf{1} \rightarrow \mathcal{C}$.
2. Show that there is a unique functor $\mathcal{C} \rightarrow \mathbf{1}$. ◇

Rather than interpreting a functor $F: \mathcal{C} \rightarrow \mathcal{D}$ as a model of \mathcal{C} in \mathcal{D} , we could interpret it as a grouping or sorting of objects (and morphisms) in \mathcal{C} according to \mathcal{D} . That is, for every object $d \in \mathcal{D}$ one could consider all the objects in \mathcal{C} that are sent to it by F and all the morphisms in \mathcal{C} that are sent to id_d by F . Let's denote this collection of objects and morphisms by $F^{-1}(d)$.

Exercise 3.17. Let $F: \mathcal{C} \rightarrow \mathcal{D}$ be a functor and let $d \in \text{Ob } \mathcal{D}$ be an object. Show that the objects and morphisms in $F^{-1}(d)$ form a category. ◇

Continuing with discrete categories, there is also a category $\mathbf{Disc}(2)$ with two objects $\text{Ob } \mathbf{Disc}(2) = \{1, 2\}$ and two identity morphisms id_1 and id_2 . This is the categorical

equivalent of a two-element set. A functor from $\mathbf{Disc}(2)$ to any category \mathbf{C} works like a selection of two objects $F1$ and $F2$ in the target, just like a function from a two-element set was selecting a pair of elements. In fact, this analogy can be made rigorous using the notion of adjunction; see ??.

The mapping out property with $\mathbf{Disc}(2)$ as the target is also interesting. It partitions objects in the source category into two groups, the ones mapped to 1 and the ones mapped to 2. But now we have to think about the arrows. All morphisms within the first group must be mapped into id_1 and all morphisms within the second group must go into id_2 . But if there are any morphisms between the two groups, they have nowhere to go. So every functor from \mathbf{C} to $\mathbf{Disc}(2)$, if it exists, must partition \mathbf{C} into two disconnected parts (one of these parts may be empty, though).

Things get even more interesting when our tiny category has arrows. For instance, we can add a morphism $\text{ar}: 1 \rightarrow 2$ to the two-object category, to arrive at the category $\mathbf{2}$ (see Example 1.36).

$$\mathbf{2} = \boxed{\text{id}_1 \hookrightarrow 1 \xrightarrow{\text{ar}} 2 \rightrightarrows \text{id}_2}$$

A functor from $\mathbf{2}$ to \mathcal{C} picks out a pair of objects in \mathcal{C} together with an arrow between them, namely $F(\text{ar})$. This is why $\mathbf{2}$ is often called a *walking arrow*.

Exercise 3.18. Let \mathcal{C} be an arbitrary category. Show that there is a one-to-one correspondence between morphisms of \mathcal{C} and functors $\mathbf{2} \rightarrow \mathcal{C}$. \diamond

Mappings out of \mathbf{C} into $\mathbf{2}$ also partition its objects into two groups. But this time any morphisms in \mathcal{C} that go from the first group to the second are mapped into our arrow ar . There is still no room for morphisms going back from the second group to the first: they have nowhere to go in $\mathbf{2}$.

Exercise 3.19. How many functors are there from \mathbf{Set} to $\mathbf{2}$? Write them down. \diamond

Going further, we can add another arrow to our two-object category, going in the opposite direction. The result is the *walking isomorphism* \mathbf{I} (Example 1.41):

$$\mathbf{I} = \boxed{\text{id}_1 \hookrightarrow 1 \begin{matrix} \xrightarrow{f} \\ \xleftarrow{f^{-1}} \end{matrix} 2 \rightrightarrows \text{id}_2}$$

Any functor from \mathbf{I} to an arbitrary category \mathcal{C} picks out an isomorphism in \mathcal{C} , as we now check.

Exercise 3.20. Given a category \mathcal{C} , show that isomorphisms in \mathcal{C} are in one-to-one correspondence with functors $\mathbf{I} \rightarrow \mathcal{C}$. Hint: A functor preserves composition and identity. \diamond

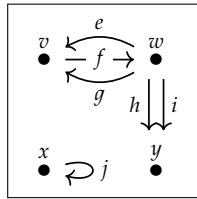
Exercise 3.21. What kind of sorting does the mapping out of \mathcal{C} to the walking isomorphism category define? \diamond

If, rather than arrows in either direction, we have two arrows in the same direction, the resulting tiny category serves as a model for graphs.

$$\mathbf{Gr} = \boxed{\begin{array}{ccc} \text{Edge} & \xrightarrow[\text{tgt}]{\text{src}} & \text{Vert} \\ \bullet & & \bullet \end{array}} \quad (3.22)$$

Consider a functor $G: \mathbf{Gr} \rightarrow \mathbf{Set}$; we will see that G represents a graph (recall Definition 1.43). First, it chooses two sets, $G(\text{Edge})$ and $G(\text{Vert})$ which we will call the set of edges and the set of vertices in G . Second, it chooses two functions src and tgt , each of which sends every edge $e \in G(\text{Edge})$ to a vertex: the source vertex $G(\text{src})(e)$ and the target vertex $G(\text{tgt})(e)$ of e .

Here we depict an example of such a functor G



$G(\text{Edge})$	$G(\text{src})$	$G(\text{tgt})$	$G(\text{Vert})$
e	w	v	v
f	v	w	w
g	w	v	x
h	w	y	y
i	w	y	
j	x	x	

We have written the elements of the sets $G(\text{Edge})$ and $G(\text{Vert})$ in the first columns of the respective tables. The results of applying functions $G(\text{src}), G(\text{tgt}): G(\text{Edge}) \rightarrow G(\text{Vert})$ to each edge $e \in G(\text{Edge})$ are written in the respective columns. The whole data structure that $G: \mathbf{Gr} \rightarrow \mathbf{Set}$ is thus represented in these two tables, but can be drawn as a graph, as shown. Elements of $G(\text{Vert})$ are drawn as vertices and elements of $G(\text{Edge})$ are drawn as arrows connecting their source to their target.

As you can see, category theory puts at our disposal a much larger variety of “shapes” from which to choose. Functors from these shapes let us describe interesting patterns in target categories. This is a very useful and productive intuition. When we say that a morphism $f: a \rightarrow b$ picks out a pattern of shape a in b , we are using this intuition. In fact, as we’ve seen earlier, functors *are* morphisms in the category \mathbf{Cat} .

Now that you know what a category is and what a functor is, you can further bootstrap your intuitions. Instead of thinking of objects as secretly being sets or bags of dots, think of objects as categories, with arrows between dots. Instead of thinking of morphisms as secretly being functions, think of them as being functors between categories. Then the whole idea of shapes makes much more sense. The only shape that a set can describe is a bag of dots. A function lets you see one bag of dots inside another bag of dots. But a functor lets you connect the dots, which makes it much closer to what we view as shapes in real life.

Similarly, categories as targets of functors provide “sorting hats”. This viewpoint will not be as useful to us, but it can still be interesting to consider.

3.2.4 The category of categories

We have seen that for every category \mathcal{C} , there is an identity functor $\text{id}_{\mathcal{C}}$. This name suggests identity functors are the identities for some notion of composition, and indeed this is the case: there is a natural notion of composition for functors.

Definition 3.23 (Composition of functors). Suppose we have functors $F: \mathcal{C} \rightarrow \mathcal{D}$, and $G: \mathcal{D} \rightarrow \mathcal{E}$. We can define a new functor $G \circ F: \mathcal{C} \rightarrow \mathcal{E}$, as follows. First, we need to give a function $\text{Ob } \mathcal{C} \rightarrow \text{Ob } \mathcal{E}$. This is given by composing our functions on objects for F and G . That is, given $c \in \text{Ob } \mathcal{C}$, we send it to $G(F c)$. Similarly, given a morphism $f: c \rightarrow c'$ in \mathcal{C} , we send it to $G(F f)$, which is a morphism $G(F c) \rightarrow G(F c')$ in \mathcal{E} .

Exercise 3.24. Show that if $F: \mathcal{C} \rightarrow \mathcal{D}$ and $G: \mathcal{D} \rightarrow \mathcal{E}$ are functors, then so is $G \circ F$, i.e. that it preserves identities and compositions. \diamond

Definition 3.25. The *category of categories*, denoted **Cat**, has categories as objects, and functors between them as morphisms. The identity functors are the identities, and composition of functors is the composition rule.

Exercise 3.26. Show that the category of categories is indeed a category, i.e. that it satisfies the unital laws and the associative law, as per Definition 1.30. \diamond

Exercise 3.27. In Example 1.56 we defined the product of two categories. Show that this category has the universal property of the product in the category **Cat** (as in Definition 2.19). That is, show that the product category really is a product. \diamond

3.3 Type classes

Before we talk about how we can use functors to structure code, it's useful to introduce one more feature of Haskell: type classes.

3.3.1 Polymorphism in Haskell

We said earlier that every Haskell term is required to have a type. But you may have noticed that some terms appear to have more than one type: they are what we call *polymorphic*. For example, consider the identity function `id = \x -> x`. This works for an input `x` of any type; we may have `id 3 :: Int` or `id "brown bear" :: String`. We thus say `id` is *parametrically polymorphic*: its type signature `id :: a -> a` contains a type variable `a`, and it has a uniform definition that works for all choices of this type.

Yet this sort of polymorphism does not explain all terms with ambiguous types. For example, what is the type of the equality operator (`==`)? We can pass it `2 :: Int` and `3 :: Int`, and it will return `False`. So here its type appears to be `(==) :: Int -> Int -> Bool`. But we can also pass it `"love" :: String` twice, and it will return `True`. So perhaps it is parametrically polymorphic, with type `(==) :: a -> a -> Bool`?

Unfortunately, not: not all types support a notion of equality, and so a uniform definition of (`==`) cannot be given. One example is function types. For example, if we define

```
f,g :: Int -> Int
f x = 2*(x + 1)
g x = 2*x + 2
```

then `f == g` results in an error.¹ Instead, the expression (`==`) is *overloaded*: it refers to multiple definitions.

In general, it is not possible to overload names, even for different type signatures. For example, if you try to define

```
f :: Int -> Int -> Bool
```

and

```
f :: String -> String -> Bool
```

the compiler will protest that you are trying to implement the same function twice.

But overloading is a very useful feature, and most languages implement some version of it. The problem is that the compiler has to figure out, at runtime, which version it's supposed to use. This is called *name resolution*, and some languages, like C++, have Byzantine rules for name resolution. In Haskell, name resolution is handled through the type system: as long as different definitions have different types, type inference can figure out which definition to use. The result is a form of polymorphism called *ad hoc polymorphism*. Ad hoc polymorphism is handled using *type classes*.

3.3.2 Defining type classes and instances

The function (`==`) is ad hoc polymorphic. Its type signature is

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

¹While it may seem like these two functions are equal in the sense that `f x` and `g x` are equal for all inputs, this is not a property that can be verified in finite time for all possible terms `Int -> Int`, and hence is not practical for implementation.

Here **Eq** is a type class, and the *type constraint* **Eq a =>** indicates that in the following type expression, the type variable **a** ranges over types that are instances of the type class **Eq**.

Haskell Note 3.28. More generally, a type class consists of a collection of *methods* that can be employed when working with a type that is an *instance* of the class. To define a type class, we use the keyword **class**. A type class definition has the following form:

```
class ClassName typeVariable where
  methods :: MethodTypes

  defaultDefinitions
```

Example 3.29. For example, the **Eq** class from **Prelude** is defined as follows:

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

  x /= y = not (x == y)
  x == y = not (x /= y)
```

Here this says that the class **Eq** contains two methods, called **(==)** and **(/=)**. Moreover, if no separate definition of **(/=)** is given, then it defaults to having the definition **x /= y = not (x == y)**. Similarly, if no separate definition of **(==)** is given, it defaults to the definition **x == y = not (x /= y)**. Note that to create an instance of **Eq** though, we must at least supply a definition of either **(==)** or **(/=)**.

Type classes provide names for methods. To make a type an instance of a class, we must give definitions for these methods.

Haskell Note 3.30. A type instance declaration has the following form:

```
instance ClassName typeVariable where
  methodDefinitions
```

Example 3.31. Consider the type **data Suit = Club | Diamond | Heart | Spade** from Example 2.53. We may make this into an instance of the type class **Eq** as follows:

```
instance Eq Suit where
```

```

Club == Club      = True
Diamond == Diamond = True
Heart == Heart     = True
Spade == Spade     = True
_ == _            = False

```

This defines a function (`==`) for the type **Suit**. Note that since the type class contains a default definition of (`/=`), our instance declaration also defines this function for **Suit**. For example, we have **Club** `/=` **Diamond** = **True**.

Here are some other type classes that are defined in **Prelude**:

Example 3.32. The **Show** class provides a single method **show**, which allows terms to be converted to strings. This can be used for, among other things, defining a way to print terms of a given type.

```

class Show a where
  show :: a -> String

```

Note that the default definitions are optional: we give no default definition for the function **show**.

Example 3.33. The numeric class **Num** gives basic numeric operations to a type, like addition, negation, and sign.

```

class Num a where
  (+), (-), (*) :: a -> a -> a
  negate, abs, signum :: a -> a

```

In particular, the types **Int**, **Integer**, and **Float** are all instances of **Num**, which allows us to use the `(+)` operator for data of all these types.

Exercise 3.34. The Gaussian integers are the complex numbers of the form $a+bi$, where a and b are integers. Addition, subtraction, multiplication, negation, and absolute value, and sign are defined for the Gaussian integers as they are for all complex numbers. (The sign of $a+bi$ is equal to the sign of a .) Make the type **(Int,Int)** an instance of the **Num** in a way that models the Gaussian integers. \diamond

Example 3.35. The **Ord** class is used for types of ordered data. Note the class constraint **Eq** $a \Rightarrow$ in the beginning of this definition. This says that any instance of **Ord** must also be an instance of **Eq**.

```
class Eq a => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max :: a -> a -> a

  min x y | x <= y      = x
          | otherwise   = y

  max x y | x <= y      = y
          | otherwise   = x
```

Haskell Note 3.36 (Algebraic structures as typeclasses). Type classes give structure for how to interact with terms of a given type. For example, since the type **Int** is an instance of the type class **Num**, we know that we may interact with terms of type **Int** as numerals, adding them, subtracting them, and so on. Similarly, since **String** is an instance of **Eq**, we know that we can ask whether two strings are equal or not. But you might have noticed that class definitions only contain type signatures.

As we have seen in many definitions in this book, including definitions of monoid, preorder, and category, an algebraic structure consists of some data subject to some laws.

For example, there are certain properties one might expect a function called `(==)` to have. These include

- (a) Reflexivity: for all values x , the expression $x == x$ is **True**.
- (b) Symmetry: if $x == y$ then $y == x$.
- (c) Transitivity: if $x == y$ and $y == z$, then $x == z$.
- (d) Preservation by functions: if $f :: Eq\ a, Eq\ b \Rightarrow a \rightarrow b, x, y :: a$, and $x == y$, then $f\ x == f\ y$.

Nowhere in the class definition of **Eq** does it state that `(==)` must have these properties.

Indeed, one is perfectly free to construct an instance of **Eq** that obeys none of these properties: as long as the type constraints are obeyed, the program will compile. But this would not be a good way to use Haskell to express our algebraic ideas.

Implicit in the names **Eq**, `==`, and `/=` is the claim that it is productive for us to think of these instances of the type class as behaving like equality, just as implicit in the name **Int** is the claim that it's productive to think of terms of this type as integers. Through the type system, the compiler handles some basic sanity checking regarding this claim. But it is ultimately the responsibility of the programmer to ensure this abstraction is reliable.

We'll be using type classes to help model structures from category theory, including functors, profunctors, monads and many more. In doing so, we'll follow the Haskell convention of simply defining the types of the key pieces of data. The laws these data must obey are not specified in code, but often written in the documentation. This guides usage of the type class. The more closely these structures obey these laws, the more reliable categorical insights are for reasoning about them, and so the more useful category theory is for writing code.

Exercise 3.37. Make **Suit** an instance of **Eq** in a way in which none of the properties of reflexivity, symmetry, and transitivity hold. \diamond

Exercise 3.38. Define a class **Monoid**, capturing the data of Definition 1.46. Make instances for the monoids $(\mathbb{Z}, 0, +)$ and $(\mathbb{B}, \text{true}, \text{AND})$. \diamond

3.4 Functors in Haskell

Let's return to our central metaphor: the types and functions of Haskell form a category. Since our path to realizing categorical ideas in code begins with this metaphor, in Haskell the word functor usually refers to an endofunctor on this category: a functor from it, to itself. So a Haskell functor maps types to types and functions to functions. In this section we'll introduce the Haskell type class **Functor**, and discuss some important examples.

3.4.1 The **Functor** type class

A Haskell functor maps types to types, and functions to functions. To begin with, this means that given a type, a Haskell functor returns another type. We've already met a way of doing this: polymorphic type constructors. For example, the type constructor **Maybe** accepts a type **a** and returns the type **Maybe a**, which is like **a** but with an extra term, **Nothing**.

So a polymorphic type constructor can serve as the on-objects part of a functor; what about the on-morphisms part? When we talk about what a functor $F: \mathcal{C} \rightarrow \mathcal{C}$ does on morphisms, we might say that it "lifts f to $F(f)$ ". If we were to extend **Maybe** to a functor, this says that if we have a function $f :: a \rightarrow b$, our functor should give a new function $f' :: \text{Maybe } a \rightarrow \text{Maybe } b$. One way of doing this is with the following definition:

```
f' :: Maybe a -> Maybe b
f' Nothing = Nothing
f' (Just a) = Just (f a)
```

Notice that we can make this `f'` construction for any function `f`. This means that we have a higher-order function

```
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap f Nothing = Nothing
fmap f (Just a) = Just (f a)
```

Thinking of `Maybe` as our functor $F: \mathcal{C} \rightarrow \mathcal{C}$, the type constructor plays the role of the on-objects map $F: \text{Ob } \mathcal{C} \rightarrow \text{Ob } \mathcal{C}$, which the polymorphic function `fmap` plays the role of the on-morphisms maps $F_{a,b}: \mathcal{C}(a, b) \rightarrow \mathcal{C}(Fa, Fb)$.

The `Functor` type class, defined in `Prelude`, captures the general pattern:

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

So a polymorphic type constructor becomes an instance of what Haskell calls a `Functor` when it is equipped with the additional function, `fmap`, that tells us how to lift functions.

Example 3.39 (Maybe). When one knows that a certain function $f: a \rightarrow b$ is partial—that not all inputs should return an output—it is often useful to turn it into a total function f' that has the same output as f on inputs where f is defined, and outputs a special value when f undefined.

This is done with the `Maybe` functor that we have just discussed. Here's the full definition

```
data Maybe a = Nothing | Just a
instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

Remark 3.40. When dealing with endofunctors, it's important to keep in mind the difference between functors and morphisms. A functor *acts* on objects—it says: give me one object, and I'll give you another (and similarly for morphisms). A morphism is an arrow *between* two objects: the source and the target. It *probes* the internal structure of an object.

An endofunctor in `Set`, for instance, could assign a set of oranges to a set of apples. But it would not map individual apples to individual oranges. That's what a function between those sets would do.

For example, there is a functor `Set` \rightarrow `Set` sending every set X to \emptyset ; it sends every morphism to the identity morphism on \emptyset . But there is never a *function* from an arbitrary set X to \emptyset (unless X itself is empty).

3.4.2 First examples of functors in Haskell

Let's give a few more elementary instances of the **Functor** type class.

Example 3.41 (Reader functor). Here is a functor **ReadInt** that takes a type **a** and returns the function type **Int -> a**:

```
data ReadInt a = MakeReadInt (Int -> a)
instance Functor ReadInt where
  fmap f (MakeReadInt g) = MakeReadInt (f . g)
```

Note here that **fmap** has type

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

There was nothing special about the choice of type **Int** in constructing this functor; we may replace it with any type **r**. We call any functor constructed in this way a **Reader** functor, as it turns any type **a** into a new type that reads in a value of **r** to create a value of **a**. We shall see later that these **Reader** functors can be extended to what are called *monads*, which provide further methods for handling this reading process.

Exercise 3.42. Give each of the following type constructors the structure of a functor by saying how to lift functions. That is, implement the following:

1. **mapDoub** :: (a -> b) -> (Double a -> Double b)
2. **mapWStr** :: (a -> b) -> (WString a -> WString b)
3. **mapUnit** :: (a -> b) -> (Unit a -> Unit b)

◇

It's also useful to have the more mathematical examples

Example 3.43 (Identity functor). In the case of **Id**, given a function **a -> b** we have to define a function **Id a -> Id b**. This mapping is accomplished using a *higher-order function*, that is a function that takes a function and returns a function:

```
mapId :: (a -> b) -> (Id a -> Id b)
```

Here are various implementation of lifting for the functor **Id**:

```
mapId1 f = \i -> MkId (f (unId i))
mapId2 f i = MkId (f (unId i))
mapId3 f (MkId x) = MkId (f x)
```

In the first, we use `unId` to retrieve the value with which the argument `i` was constructed, apply the function `f` to it, and construct a new value of the type `Id b` using `MkId`. In the second, we simplify the syntax by treating `mapId` as a function of two arguments. And in the third, we pattern match the argument directly.

The last version, `mapId3` is generally preferred because it exposes a nice sort of symmetry or perhaps commutativity: the `f` seems to “move past” the `MkId`.

Example 3.44 (Constant functor). We can repeat the same procedure for the constant functor:

$$C_{\text{Int}} = \Lambda a. \text{Int}$$

We first write down the polymorphic type constructor:

```
data CInt a = MkC Int
```

This time, though, the data constructor doesn’t use the `a` at all; for example `CInt 42` is a term of type `CInt a` for any `a`.

Strictly speaking, `CInt` is not a constant functor, since every type `a` is mapped into a different type `CInt a`. However, all these types are isomorphic, as we’ll see in Exercise 3.45; in fact they are “naturally isomorphic” in the sense of natural transformations, which we’ll come to later.

Exercise 3.45. Here we implement the two functions that witness the isomorphism between `Int` and `CInt a`.

1. Specify a function of type `CInt a -> Int`.
2. Specify a function of type `Int -> CInt a`.

◇

Exercise 3.46. Show that the polymorphic type constructor `CInt` can be given the structure of a functor by saying how it lifts morphisms. That is, provide a Haskell function `mapCInt` of the type `(a -> b) -> (CInt a -> CInt b)`.

◇

Example 3.47 (Are Haskell functors always functors?). Someone might ask: “if I define an instance of the `Functor` class in Haskell, is it always a functor in the sense of category theory?” The person is asking whether the functor laws—preservation of identity and composition—hold. The answer is no.

We start with an example functor and then give a non-example that’s quite similar. So here’s a good functor:

```
data Pair a = MkPair (a, a)
```



```
instance Functor Pair where
  fmap f MkPair (a1, a2) = MkPair (f a1, f a2)
```

This says that for any $f: A \rightarrow B$, we can take a pair of A 's and turn it into a pair of B 's, e.g. if we start with `isEven :: Int -> Bool` and apply `fmap isEven (3,4)`, we get `(False, True)`. This is a functor, because it preserves the identity and composition, e.g. `fmap id (3,4)` gives `(3,4)`.

Here's an instance of the Haskell type class `Functor` that does not correspond to any functor `Set → Set`.

```
data Pair a = MkPair (a, a)
instance Functor Pair where
  fmap f MkPair (a1, a2) = MkPair (f a2, f a1) --swap!!
```

Then `fmap id (3,4)` returns `(4,3)`, so `fmap id` is not `id`. It does not preserve identities.

Exercise 3.48. For each of the following type constructors, define two versions of `fmap`, one of which has a corresponding functor `Set → Set`, and one of which does not.

1. `data WithString a = WithStr (a, String)`
2. `data ConstStr a = ConstStr String`
3. `data List a = Nil | Cons (a, List a)` ◇

3.4.3 Bifunctors

If every pair of objects in a category \mathcal{C} has a product, we can define a functor $\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$. One way to think of this is as a functor in two arguments; hence it is sometimes called a *bifunctor*.

The library `Data.Bifunctor` contains the following typeclass:

```
class Bifunctor f where
  bimap :: (a -> a') -> (b -> b') -> (f a b -> f a' b')
```

It works on type constructors `f` that requires two type arguments, and is an analogue of `fmap` that asks for two functions rather than one, but still returns one function `f a b -> f a' b'`.

Let's see how products—pairs—define a bifunctor in Haskell. On objects, we see that the product takes two types `a` and `b` and returns a single type `(a,b)`; this is the type constructor. To inform the compiler that this type constructor as a bifunctor, we provide `bimap` as follows:

```
instance Bifunctor (,) where
  bimap f g = \p -> (f (fst p), g (snd p))
```

or, using pattern matching:

```
bimap f g (a, b) = (f a, g b)
```

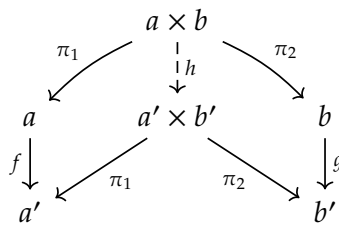
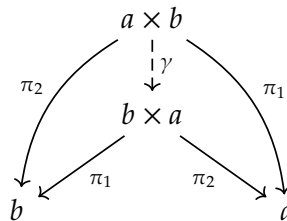


Figure 3.1: Functoriality of the product: $h = ((f \circ \pi_1), (g \circ \pi_2))$

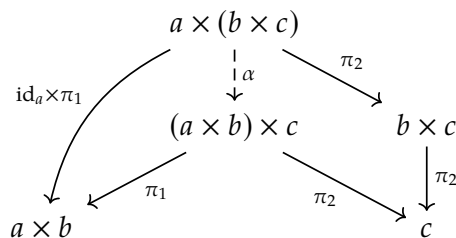
In fact, products give more structure: not only do they form a bifunctor, but this bifunctor gives what is known as a *symmetric monoidal structure* on the category. We won't say exactly what this means, but what is important for programming in Haskell, is that the universal property of the product given four isomorphisms.

These are

1. Symmetry



2. Associativity



3, 4. Two unit isomorphisms

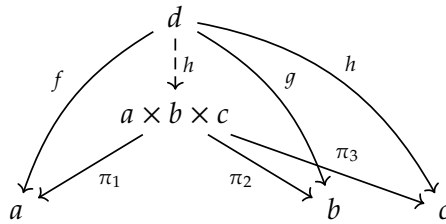


Exercise 3.49. Implement functions of the following type signatures:

1. `swap :: (a,b) -> (b,a)`
2. `assoc :: (a,(b,c)) -> ((a,b),c)`
3. `unitl :: a -> ((),a)`
4. `unitr :: a -> (a,())`
5. `double :: a -> (a,a) -- bonus!`

◇

Because of associativity, nested pairs can be simplified to tuples of multiple types. For instance, because the type $(a, (b, c))$ is isomorphic to $((a, b), c)$, we can without ambiguity use a tuple (a, b, c) of three types. As in Exercise 2.29 we could also define a triple products (or any finite product, for that matter) using a universal construction:

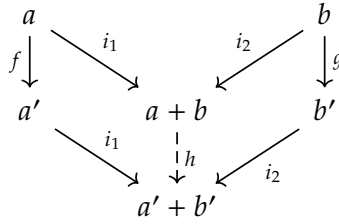


We could do this for n -many types, for any $n \in \mathbb{N}$. When $n = 0$ we see the empty tuple; hence the notation `()`.

To show that the coproduct defines a bifunctor, we must implement a function of the following type signature:

```
bimap :: (a -> a') -> (b -> b') -> (Either a b -> Either a' b')
```

We can look at it as a mapping out of a comproduct, so we can use the universality of the following diagram with $f: a \rightarrow a'$ and $g: b \rightarrow b'$



Reading this diagram, we get

```
instance Bifunctor Either where
  bimap f g = either (Left . f) (Right . g)
```

Or, in pattern matching syntax:

```
instance Bifunctor Either where
  bimap f g (Left a) = Left (f a)
  bimap f g (Right b) = Right (g b)
```

In analogy with Exercise 3.49 we can define symmetry, associator, and unitor isomorphisms for the coproduct.

Exercise 3.50. Implement functions of the following type signatures:

1. `swap :: Either a b -> Either b a`
2. `assoc :: Either a (Either b c) -> Either (Either a b) c`
3. `unitl :: Either Void a -> a`
4. `unitr :: Either a Void -> a`
5. `double :: Either a a -> a -- bonus!`

◇

3.4.4 A first glance at profunctors

We've seen before that both the product and the coproduct are bifunctors. Let's try to establish the functoriality of the exponential. First, let's check if c^b is functorial in c . To that end, let's assume that we have a morphism $g: c_1 \rightarrow c_2$ and try to lift it to the morphism $c_1^b \rightarrow c_2^b$.

To get such a morphism, it suffices by currying to obtain a morphism $c_1^b \times b \rightarrow c_2$, which we obtain as the following composite

$$c_1^b \times b \xrightarrow{\text{eval}} c_1 \xrightarrow{g} c_2$$

However, if we tried to do the same trick to establish functoriality of c^b in b , we would fail. The reason is that the exponential is *contravariant* in b ; it varies backwards. It cannot lift a morphism $b_1 \rightarrow b_2$, but it can lift a morphism going in the opposite direction: it can lift a morphism $g: b_2 \rightarrow b_1$ to obtain a morphism $c^{b_1} \rightarrow c^{b_2}$.

Exercise 3.51.

1. Implement a function of the type

```
(c1 -> c2) -> (b -> c1) -> (b -> c2)
```

2. Implement a function of the type

```
(b1 -> b2) -> (b2 -> c) -> (b1 -> c)
```

◇

So the exponential c^b is covariant in c and contravariant in b . Said categorically, if the exponential object is defined for every pair of objects in, we have a functor

$$-^{\circ}: \mathcal{C}^{\text{op}} \times \mathcal{C} \rightarrow \mathcal{C}.$$

Recall that the opposite category \mathcal{C}^{op} has the same objects as the original one, but all the arrows are reversed. This functor lifts a single morphism in $\mathcal{C}^{\text{op}} \times \mathcal{C}$, which is a pair of morphisms from \mathcal{C} , the first going in the opposite direction.

The functions from Exercise 3.51 can thus be combined into one function

```
dimap :: (a' -> a) -> (b -> b') -> ((a -> b) -> (a' -> b'))
dimap g' g = \h -> g . h . g'
```

Just like we had a type class for bifunctors (functors out of $\mathcal{C} \times \mathcal{C}$), there is a typeclass for bivariant functors (functors out of $\mathcal{C}^{\text{op}} \times \mathcal{C}$) in Haskell

```
class Profunctor p where
  dimap :: (a' -> a) -> (b -> b') -> (p a b -> p a' b')
```

We have just shown that the infix operator `->` is an instance of **Profunctor**

```
instance Profunctor (->) where
  dimap g' g h = g . h . g'
```

As our friend John Baez once said, profunctors are functors for pro's. For example, they generalize functors, as we'll now see.

Example 3.52. Suppose that $F: \mathcal{C} \rightarrow \mathcal{C}$ is a functor. There is a related profunctor called the *companion* of F , denoted \hat{F} . It is implemented as follows:

```
data Companion f a b = Companion ((f a) -> b)
```

```
instance Functor f => Profunctor (Companion f) where
    dimap g' g (Companion h) = Companion $ g . h . (fmap g')
```

We will see profunctors again later.

3.5 Natural transformations

When there is more than one functor between two categories, we may ask the question: How are these models related to each other? We need to define mappings between the images of two functors.

A natural transformation is a structure preserving mapping between functors.

3.5.1 Definition

A natural transformation α is a map between parallel functors; you can draw it like this:

$$\begin{array}{ccc} & F & \\ \mathcal{C} & \xrightarrow{\quad} & \mathcal{D} \\ & \Downarrow \alpha & \\ & G & \end{array}$$

But functors are themselves mappings; what's a mapping between mappings? The secret is that natural transformations really do all their work in the target category \mathcal{D} , though they take their cues from \mathcal{C} . For every object c in \mathcal{C} we get two objects, Fc and Gc in \mathcal{D} . The natural transformation α picks a morphism from Fc to Gc . This way we define, for every object c , the *component* α_c of the natural transformation. If, for some c , there are no morphisms in \mathcal{D} between Fc and Gc , there can be no natural transformation between F and G . The two functors are unrelated.

If functors were only mapping objects to objects, we would be done. But functors also map morphisms. Every morphism f in \mathcal{C} is mapped to two morphisms in \mathcal{D} : Ff and Gf . These morphisms can be composed with the components of the natural transformation in two different ways. Naturality means that these two compositions should be equal.

Definition 3.53 (Natural transformation). Let \mathcal{C} and \mathcal{D} be categories, and let $F, G: \mathcal{C} \rightarrow \mathcal{D}$ be functors. A *natural transformation* α from \mathcal{C} to \mathcal{D} , denoted $\alpha: F \Rightarrow G$, consists of a morphism $\alpha_c: F(c) \rightarrow G(c)$ for each $c \in \text{Ob}(\mathcal{C})$, called the *c-component*. These components are subject to a condition called *naturality*. Namely, for each $f: c_1 \rightarrow c_2$ in

the following square needs to commute in \mathcal{D} :

$$\begin{array}{ccc} F(c_1) & \xrightarrow{F(f)} & F(c_2) \\ \alpha_{c_1} \downarrow & & \downarrow \alpha_{c_2} \\ G(c_1) & \xrightarrow{G(f)} & G(c_2) \end{array} \quad (3.54)$$

3.5.2 Natural transformations in Haskell

A natural transformation is a family of morphisms, one per object. Haskell will never check the naturality condition Eq. (3.54), so to implement a natural transformation in Haskell we just need this family of functions, one for every type.

For example, consider the functions

```
singletonBool :: Bool -> [Bool]
singletonChar :: Char -> [Char]
singletonBool b = [b]
singletonChar c = [c]
```

We didn't really use anything about booleans or characters to do this. Natural transformations in Haskell are polymorphic functions: they allow us to work with all types at once. We can write

```
singleton :: a -> [a]
singleton x = [x]
```

More generally, for any two (endo-) functors, f and g , a natural transformation is a polymorphic function

```
natTrans :: f a -> g a
```

In the case of `singleton`, the functors were identity `Id` from Example 3.43 and list `[]`, and we could have just as well written

```
singleton :: Id a -> [a]
singleton (Id x) = [x]
```

Note that the above `natTrans` is defined for all `a`, and using a language pragma, one can write it as

```
{-# language RankNTypes #-}
natTrans :: forall a. f a -> g a
```

Since we'll be using natural transformations throughout the chapter, let's make a little infix operator, which can be inserted between two functors, to denote natural transformations:

```
{-# language TypeOperators #-}
type f ~> g = forall a. f a -> g a
```

Notice that **type** doesn't define a new type, it introduces a synonym for an existing type. The infix definition is equivalent to:

```
type (~>) f g = forall a. f a -> g a
```

which says that $(\sim>)$ takes two type constructors, **f** and **g** and constructs a type of polymorphic functions. The compiler knows that these are type constructors, because they are applied to the type variable **a**.

Example 3.55. Here is our infix operator $\sim>$ in action. We first define the type of all natural transformations between **Id** and **CInt**:

```
type IdToConst = Id ~> CInt
```

Then we can give an example term of this type:

```
seven :: IdToConst String
seven _ = MkC 7
```

In fact all examples of **IdToConst** will be like this; see Exercise 3.56 for a set-theoretic analogue.

Exercise 3.56. Consider the identity endofunctor $\text{id}: \mathbf{Set} \rightarrow \mathbf{Set}$ and the constant endofunctor $C_{\mathbb{N}}: \mathbf{Set} \rightarrow \mathbf{Set}$ where $C_{\mathbb{N}}(X) = \mathbb{N}$ for all $X \in \text{Ob}(\mathbf{Set})$. Show that every natural transformation

$$\begin{array}{ccc} \mathbf{Set} & \xrightarrow{\text{id}} & \mathbf{Set} \\ & \Downarrow \alpha & \\ \mathbf{Set} & \xrightarrow{C_{\mathbb{N}}} & \mathbf{Set} \end{array}$$

is constant, i.e. that if α is such a natural transformation then there is some $n \in \mathbb{N}$ such that for all $X \in \mathbf{Set}$ the component $\alpha_X: X \rightarrow \mathbb{N}$ is the constant function at n . \diamond

Haskell Note 3.57. When defining an operator, you can specify its fixity, associativity, and precedence, as in:

```
infixr 0 ~>
```

This means that `~>` is an *infix* operator that associates to the right and has the lowest precedence, zero. In comparison, the function composition operator `(.)` has the highest numeric precedence of 9 and function application is off the scale, with the highest precedence.

As we said, the naturality condition (the commutativity of Eq. (3.54)) is not directly expressible in Haskell but it is satisfied anyway. This is because the actual implementation of any natural transformation in Haskell can only be done within the constraints of the polymorphic lambda calculus, or system **F**, so the whole family of functions must be described by a single formula. There is no such restriction for the components of a natural transformation $\alpha: F \Rightarrow G$ in Definition 3.53. In principle, one could pick a completely different morphism $\alpha_a: Fa \rightarrow Ga$ for every object a . This would correspond to picking one function, say, for **Bool**, and a completely different one for **Int**, and so on. Such a thing is not expressible in system **F**. The one-formula-for-all principle leads to parametricity constraints, also known as "theorems for free." One such theorem is that a polymorphic function:

```
forall a. f a -> g a
```

for any two functors expressible in system **F** is automatically a natural transformation satisfying naturality conditions.

The alternative to parametric polymorphism is called "ad hoc polymorphism." It allows for varying the formulas between types. The typeclass mechanism we've seen earlier implements this idea. For instance, the implementation of `fmap` varies from functor to functor.

Example 3.58 (Lists to Maybes). In Example 3.39 we defined the **Maybe** functor. Here we'll show that there is a natural transformation from **List** (i.e. `[]`) to **Maybe**.

```
safeHead :: [] ~> Maybe
safeHead [] = Nothing
safeHead (a: as) = Just a
```

Exercise 3.59. Implement a natural transformation

```
uncons :: [a] -> Maybe (a, [a])
```

◇

Haskell Note 3.60. The function `uncons` is defined in a library, so you can use it if you put the import statement at the beginning of your file:

```
import Data.List
```

If the library is in your path, you can load it into GHCi using the command

```
:load Data.List
```

3.6 Bonus: Representable functors and the Yoneda embedding

To end this chapter, we want to provide one more lesson on categorical thinking. Namely, we want to give a sense of a beautiful theorem, known as the Yoneda lemma, which sits at the core of category theory. Roughly, the Yoneda lemma says that an object in a category is no more and no less than its web of relationships with all other objects. The Yoneda lemma formalizes the fact that thinking in terms of relationships is as powerful as we'll ever need. It will also help us think in what is known as a “point-free” way, in terms of generalized elements.

A category is a web of relationships. Let a be an object in a category \mathcal{C} . Given any object x in \mathcal{C} , we can ask what a looks like from the point of view of x . The answer to this is encoded by the hom-set $\mathcal{C}(x, a)$, which is the set of all morphisms from x to a . Collecting these views over all objects x of \mathcal{C} , we can define a functor, known as the *functor represented by a* .

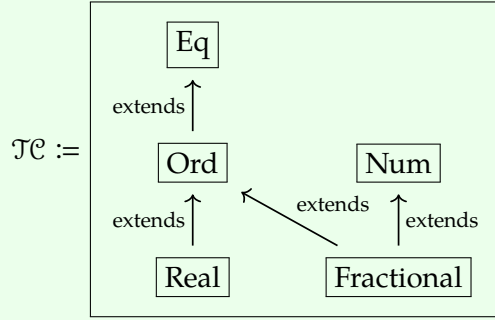
Definition 3.61. Let a be an object in a category \mathcal{C} . We may define a functor

$$y_a: \mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}$$

sending each object x of \mathcal{C} to the set $\mathcal{C}(x, a)$, and sending each morphism $m: x \rightarrow y$ to the function $y_a(m): \mathcal{C}(y, a) \rightarrow \mathcal{C}(x, a)$ given by sending the element $f: y \rightarrow a$ to the element $(f \circ m): x \rightarrow a$.

We call y_a the *functor represented by a* .

Example 3.62 (In a poset). If \mathcal{C} is a poset and $a \in \mathcal{C}$ is an object, the representable functor on a answers the question “is x less than or equal to a ?” for every x . The answer is Yes, if the hom-set $\mathcal{C}(x, a)$ is a singleton, and No, if it’s empty. For example, consider the poset



$$\begin{aligned}
 y_{\text{Ord}}(\text{Eq}) &= \emptyset \\
 y_{\text{Ord}}(\text{Ord}) &= \{\text{extends}\} \\
 y_{\text{Ord}}(\text{Num}) &= \emptyset \\
 y_{\text{Ord}}(\text{Real}) &= \{\text{extends}\} \\
 y_{\text{Ord}}(\text{Fractional}) &= \{\text{extends}\}
 \end{aligned}$$

from ?? . The functor $y_{\text{Ord}}: \mathcal{TC}^{\text{op}} \rightarrow \mathbf{Set}$ is the table shown right above.

Example 3.63. Let’s think about the category \mathbf{Set} . We’ll write $\underline{1} = \{*\}$ for some set with one element. What is the functor represented by $\underline{1}$? First, it is a functor $y_{\underline{1}}: \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Set}$. Where does it send a set X ? Well, what are the functions from $X \rightarrow \underline{1}$? Let’s suppose we have a function $f: X \rightarrow \underline{1}$. We have to send every element of $x \in X$ to some element $f(x)$ of $\underline{1}$. But there is only one element of $\underline{1}$; its name is $*$. So $f(x)$ must equal $*$. Thus there is only one function from X to $\underline{1}$, and hence $y_{\underline{1}}$ sends every set to a one element set. We recognize it as the constant functor.

Exercise 3.64. Consider the functor $y_{\mathbb{B}}: \mathbf{Set}^{\text{op}} \rightarrow \mathbf{Set}$, where $\mathbb{B} = \{\text{true}, \text{false}\}$.

1. What is $y_{\mathbb{B}}(\{1, 2, 3\})$?
2. Consider the function $f: \{a, b, c\} \rightarrow \{1, 2, 3\}$ given by $f(a) = f(b) = 1$ and $f(c) = 2$. Write out $y_{\mathbb{B}}(f)$ as a function $y_{\mathbb{B}}(\{1, 2, 3\}) \rightarrow y_{\mathbb{B}}(\{a, b, c\})$.
3. Explain how you can think of an element $y_{\mathbb{B}}(X)$ as a subset of X for each set X .
4. Given a function $f: W \rightarrow X$, what does $y_{\mathbb{B}}(f)$ do to subsets, given the perspective you came up with in part 3. ◇

Exercise 3.65. We said that the functor represented by a is a functor $y_a: \mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}$. Prove that it does indeed obey the conditions in the definition of functor, Definition 3.1. ◇

So for any category \mathcal{C} and an object $a \in \mathcal{C}$, we can look at a ’s part in the network of relationships that is \mathcal{C} , and we can package it as a set-valued functor. Why bother with this? The Yoneda lemma says that knowing this information completely characterizes—determines—the object a up to unique isomorphism. We’ll just state this particular part of the lemma, to show how it looks in the language of category theory.

Theorem 3.66 (The Yoneda embedding). Let a and b be objects of a category \mathcal{C} . If there is a natural isomorphism between the functors y_a and $y_b: \mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}$, then a is isomorphic to b .

Exercise 3.67. Suppose we have a natural transformation

$$\begin{array}{ccc} \mathcal{C} & \begin{array}{c} \xrightarrow{y_a} \\ \Downarrow \alpha \\ \xrightarrow{y_b} \end{array} & \mathbf{Set} \end{array}$$

Use it to find a morphism $a \rightarrow b$. Hint: find a special element of $y_a(a)$ and plug it in to the component α_a . \diamond

We'll leave the full statement and the proof of the Yoneda lemma to another resource; an excellent presentation is given in Leinster for example. For now we want to unpack some of its philosophical consequences, and discuss how they affect how we think about category theory and programming.

Let's imagine the Yoneda embedding as a game. First, we choose a category \mathcal{C} that we both completely understand. Then I pick a secret object a in \mathcal{C} . Your goal is to find an object that's isomorphic to my secret object.

You're allowed to get information about the object in two ways. First, if you name an object x , I must tell you the set $y_a(x) = \mathcal{C}(x, a)$ abstractly as a set, but I don't have to tell you anything about how its elements are related to the morphisms you see in \mathcal{C} . Second, if you name a morphism $m: x \rightarrow x'$, I have to tell me the function $y_a(m): \mathcal{C}(x', a) \rightarrow \mathcal{C}(x, a)$ between those abstract sets.

The Yoneda lemma says that you can always win this game; see Exercise 3.68.

Let's think about how I might win it in a poset. Suppose you pick some element a of a poset P . I can keep naming elements of P , and ask you if they are less or equal to a (see Example 3.62). With every question I can narrow down the choices until all that remains is a and the elements that are isomorphic to it. Here, x is isomorphic to a if $x \leq a$ and $a \leq x$.

Another helpful example is in the category \mathbf{Set} . In fact, in this setting I can win the game just by naming one object. Which object? The object $\underline{1} := \{1\}$. To see this, think about the functions f from $\underline{1}$ to any set a . Such a function sends the unique element 1 of $\underline{1}$ to some element $f(1)$ of a . That's it. So functions $f: \underline{1} \rightarrow a$ are the same as elements of a . In other words, $\mathbf{Set}(\underline{1}, a) \cong a$. So I ask you $y_a(\underline{1})$, you give me a set, and I say "that's a !"

In the category \mathbf{Set} , we only have sets (objects) and functions (morphisms); we don't know what an *element* is. But the above strategy shows that we don't need to, as long as we know which object is $\underline{1}$: an element of X is the same thing as a function $\underline{1} \rightarrow X$. We call a function $\underline{1} \rightarrow X$ a *global element* of X .

Note that from this viewpoint, evaluation of functions is a special case of composition. Suppose that we have an element $x \in X$, and a function $f: X \rightarrow Y$, and want to figure out $f(x) \in Y$. The global element corresponding to $f(x)$ is simply the composite of $f: X \rightarrow Y$ with the global element $x: \underline{1} \rightarrow X$ corresponding to x .

$$\begin{array}{ccc} & \underline{1} & \\ x \swarrow & & \searrow y \\ X & \xrightarrow{f} & Y \end{array}$$

The category **Set** is very special in this regard: the Yoneda embedding game can be won just by naming a single object. In a more general setting, the Yoneda embedding game is not so immediate. It thus helps not just to talk of global elements, but about a notion of *generalized element*.

A generalized element of x of shape c is just another name for a morphism $c \rightarrow x$. Nonetheless, it's useful to think in these terms. While a set is defined by its (global) elements, the Yoneda embedding says that in any category, an object is (roughly speaking, that is up to isomorphism) defined by its generalized elements.

Exercise 3.68 (Challenge). Give a strategy for winning the above “Yoneda” game in a general category \mathcal{C} . (You can either assume that \mathcal{C} has finitely many objects and morphisms or that you have infinite amount of time and patience.) \diamond

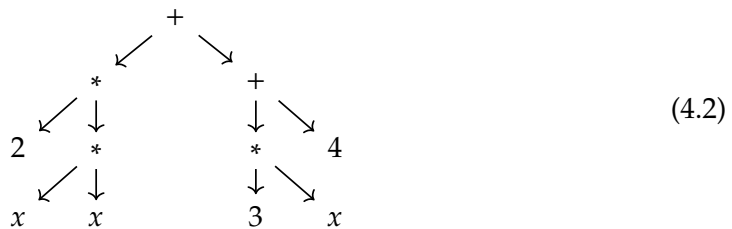
Algebras and recursive data structures

When we think about algebra, we think of solving equations (or sets of equations) with variables like x or y . There are two parts to an algebra: one is the creation of an expression and the other is the evaluation.

From the point of view of a programmer, an expression is modeled as a tree whose nodes are *operations*, like $+$ or $*$, and whose leaves are *terminal symbols*, either constants or variables (placeholders for values). For instance, the expression

$$2x^2 + 3x + 4 \tag{4.1}$$

corresponds to a tree



The tree in Eq. (4.2) is called a *parse tree*; it's how a parser would see the expression from (4.1). A parser takes such expressions and converts them into tree-like data structures, ones specifically chosen for the types of operations and terminal symbols the programmer wants to consider. Here the tree-like data structure decorates each node with either **Plus** or **Times** and decorates each leaf with either **Const** or **Var**.

<pre> data Expr = Plus Expr Expr Times Expr Expr Const Double Var String </pre>	<pre> -- Form an expression by either -- adding two expressions -- multiplying two expressions -- having a constant in hand, or -- having a variable name in hand. </pre>
--	---

This is a recursive data structure: its definition refers to itself, and how this looks category-theoretically is the subject of the present chapter.

Exercise 4.3. The parse tree in Eq. (4.2) actually includes an implicit choice of how to parenthesize the expression in Eq. (4.1); what is it? \diamond

When constructing an expression of the above form, you can bootstrap yourself by first constructing the non-recursive expressions—those that don’t include **Expr** again—namely the **Const** or **Var** constructors. These in turn can be passed to the recursive constructors **Plus** and **Times**. The expression Eq. (4.2) would be written as

```
expr :: Expr
expr = Plus (Times (Const 2)
                  (Times (Var "x") (Var "x")))
          (Plus (Times (Const 3) (Var "x"))
              (Const 4))
```

4.1 The string before the knot

One might make an analogy between the Haskell data type **Expr** on page 97 and some sort of fractal: it is built out of smaller versions of itself.¹ We can also imagine a recursive data type as being like a knot, or looped string. Our goal in this section is to explain what serves as the string and what serves as the knot.

The string of a recursive data type is a functor, and the knot is its initial algebra.

To begin understanding **Expr**, we define a non-recursive type constructor that looks almost the same, except with a placeholder **a** replacing the recursive branches.

```
data ExprF a = PlusF a a      --As above, except a in place of Expr
             | TimesF a a
             | ConstF Double
             | VarF String
```

Before we go on, let’s verify that this is a functor, just so you see the track we’re on.

¹In fact, there is a formal relationship in terms of (“colored”) operads. Free operads correspond to context-free grammars; see [MakkaiPowerHermida], and fractals like Sierpinski triangles, etc. correspond to fixed points of certain operad algebras [Leinster].

Exercise 4.4.

1. Implement the functor instance `fmap` for `ExprF`.
2. If this were a functor $F: \mathbf{Set} \rightarrow \mathbf{Set}$, what would it be?
3. What is $F(\emptyset)$?
4. What is $F(F(\emptyset))$?

◇

What can we do with the functor `ExprF`? Before we said that we build up expressions from the leaves, i.e. the constructors that don't include `Expr` (or what is now `a`). Interestingly, this corresponds to a simple idea: apply the functor to `Void`. This will allow us to construct the leaves, but nothing else

```
type Leaf = ExprF Void
```

For instance, we can construct

```
e3, ex :: Leaf
e3 = ConstF 3
ex = VarF "x"
```

But that's about the end of it: if we wanted to use the other two constructors to construct terms of type `Expr Void`, we would have to provide them with terms of type `Void`, and there aren't any.

But we can do something else instead, hinted at in Exercise 4.4. Namely, if we want to construct depth-2 trees, we apply `ExprF` to the newly constructed leaves. We define a new type

```
type Expr2 = ExprF Leaf    -- ExprF (ExprF Void)
```

With this new type we can build some shallow expressions like

```
e3x, e4, ex2, e2 :: Expr2
e3x = TimesF e3 ex -- 3 * x
e4  = ConstF 4
ex2 = TimesF ex ex -- x * x
e2  = ConstF 2
```

Continuing this process, we can define deeper and deeper tree types

```
type Expr3 = ExprF Expr2
type Expr4 = ExprF Expr3
```

and build up larger subexpressions, all the way to our original expression tree:

```
e3xp4, e2x2 :: Expr3
e3xp4 = PlusF e3x e4      -- 3 * x + 4
e2x2  = TimesF e2 ex2     -- 2 * x * x

expr' :: Expr4            -- Finally!
expr' = PlusF e2x2 e3xp4 -- 2 * x * x + 3 * x + 4
```

Notice that we didn't use recursion at all. On the other hand, every time we wanted to increase the depth of our tree, we had to define a new type. We ended up with the following type

```
expr' :: ExprF (ExprF (ExprF (ExprF Void)))
```

Here's a crazy idea: what if we could tie a knot that would make the output of **ExprF** also be its input type. This would let us apply **ExprF** infinitely many times, and so we should get a data type that can deal with a tree of any depth. Amazingly enough, as we will soon see, this procedure can be made rigorous, and yield the original recursive definition of **Expr** (or, at least, something isomorphic to it). This relies on something important about our type constructor **ExprF**, namely that it is a functor, as you checked in Exercise 4.4.

We will get to how to tie the knot in Section 4.3, but first we want to say what you can do with recursive data types. This not only motivates the formalism, it's an integral part of it!

4.2 What you can do with recursive data types

Our running example of **Expr** and **ExprF** illustrates how we can (painstakingly) create expressions starting from a functor. But algebraic expressions don't do much for us in the abstract; the point of such an expression is to evaluate it.

What does it mean "to evaluate" an expression? It means replacing the whole tree with a single value of some chosen target type by performing operations as specified. The obvious choice for the target type would be **Double**, since the **Const** leaf contains it.

But there are other options as well, e.g. **String**! We could take the expression tree from Eq. (4.2) for instance, and evaluate it as a single string, say **"2*x*x+3*x+4"**. Or we could hand it off to something that estimates the time each operation would take and then returns to us the total time for this computation.

We could come up with any number of ways to evaluate the expression, in terms of any target type we could think of, as long as we abandon our preconceptions about

the meaning and properties of the operators in question. We could evaluate $+$ using $-$ and $*$ as `const (const 0)`. So what exactly are the rules for evaluating?

Let's start with the case of evaluating to the target type **Double**, as it makes most sense to us. Here's the type:

```
eval1 :: ExprF a -> Double
```

We start with the leaves. The **ConstF** leaf is obvious:

```
eval1 (ConstF d) = d    -- evaluate the double d as itself
```

To evaluate the variable leaf, we have to decide what value to assign to **"x"**. Let's say our evaluator will set **"x"** equal to 2 (we can create a separate evaluator for each interesting value of **"x"**, or we could make a function that takes **"x"** and returns the appropriate evaluator)

```
eval1 (VarF "x") = 2
```

We are pattern matching the constructor **VarF** from Section 4.1 and, inside it, pattern matching the string **"x"**. This is not an exhaustive match, so we'll provide the default pattern as well. We'll set, arbitrarily, all other variables to zero

```
eval1 (VarF _) = 0
```

Haskell Note 4.5. The underscore `_` is a match-all *wildcard pattern*, but the matching is done in the order of definitions, so it will be tried only after the first match, with the string **"x"**, fails. In a more professional implementation, we would look up the name of the variable in some kind of environment that matches variable names with values.

But now we have a problem: how to implement the operators? The obvious (and correct) choice doesn't type check

```
eval1 (PlusF x y) = x + y
```

The compiler tells us that it "Couldn't match expected type 'Double' with actual type 'a'". This is remedied by making the type of `a` be **Double**. Our evaluator will have the type

```
eval1 :: ExprF Double -> Double
```

The idea is that, when the time comes to evaluate the **PlusF** or **TimesF** node, we will have already evaluated the child nodes, and they would be the values of type **Double**.

All that's left is to combine these results. Here's the complete evaluator

```
eval1 :: ExprF Double -> Double
eval1 (ConstF d)    = d
eval1 (VarF "x")     = 2
eval1 (VarF _)       = 0
eval1 (PlusF x y)    = x + y
eval1 (TimesF x y)   = x * y
```

There's just one thing missing: How do we combine these partial evaluators into one recursive algorithm that would evaluate any recursive expression tree. This is what we'll be studying in the next section, with the help of category theory. For now, let's gather together the important components of an algebra we've been using so far, without being too specific.

We need an endofunctor F —here we used **ExprF**. Even if the more general categorical setting, this must still be an endofunctor $F: \mathcal{C} \rightarrow \mathcal{C}$ rather than an arbitrary functor between any two categories; the reason is simply that we'll need to recursively apply it to itself.

Once we have the endofunctor we can talk about algebras for it. These are just the “evaluators” for in the above sense; it is these we call F -algebras. We did above this by choosing a type a (namely **Double**), and a morphism $Fa \rightarrow a$. Notice that, had we chosen a different type, say **String**, we would have to implement a different evaluator, namely **ExprF String -> String**.

Exercise 4.6. Implement an evaluator **showEx :: ExprF String -> String** that produces a text version of the expression, as on page 100. \diamond

Now that we have made connection with the high-school definition of algebra, we can start exploring what else fits our abstract description. For instance, what if we simplified the tree by replacing binary nodes with unary nodes? A tree that doesn't branch is just a list.

In the expression tree, we stored values at the leaves, but a non-branching tree can only have one leaf. So let's instead store values at the nodes and, for the sake of an example, let's store integers. We get:

```
data ListF a = Node Int a
              | Leaf
```

We can represent an empty list by replacing **a** with **Void**

```
type List0 = ListF Void
```

Lists of up to length one can be generated by

```
type List1 = ListF List0
```

and so on. Each next type can accommodate lists longer by one integer.

Here are two examples of evaluators `ListF Int -> Int`

```
evalSum (Node n x) = n + x
evalSum Leaf = 0
```

and

```
evalProd (Node n x) = n * x
evalProd Leaf = 1
```

Our intuitions about algebras as having something to do with operations on numbers can get us only so far. A list, for instance, doesn't have to contain numbers. Here's a more general definition of a list functor which, as we'll see later, leads directly to the definition of a list:

```
data ListF c a = NilF | ConsF c a
```

This definition is parameterized by the type of the contents `c`. We also use more traditional names for the leaf and node constructors of a list. Categorically, this functor is written as $1 + c \times a$, as a coproduct of the terminal object 1 and a product of c and a .

4.3 Algebras

On page 102 and in Exercise 4.6 respectively, we gave a programs

```
eval1 :: ExprF Double -> Double
showEx :: ExprF String -> String
```

the first of which evaluates an expression as a `Double`, the second of which pretty-prints an expression as a `String`. The commonality is captured by the following Haskell type:

```
type ExprFAlg a = (ExprF a) -> a
```

It is in this idea—called an algebra—that the recursive knot will be tied in Section 4.4. For now, we have motivated the following:

An F -algebra is just a morphism $Fa \rightarrow a$ for some particular a .

Definition 4.7 (F -algebra). Let $F: \mathcal{C} \rightarrow \mathcal{C}$ be an endofunctor. An F -algebra (c, φ) is

- (a) An object $a \in \mathcal{C}$, called the *carrier*
- (b) A morphism $\varphi: Fa \rightarrow a$, called the *structure map*.

In Haskell, we define:

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

An **Algebra** is parameterized by a type constructor **f** and a type **a**.

Notice that the function $(f\ a) \rightarrow a$ is *not* polymorphic (there is no **forall** there). We define an algebra as a *particular* function for a *particular* choice of type **a**. There is no additional condition imposed on this function. There are no laws to be enforced.

We can say now that **ExprFAlg** above is an **Algebra** with the functor **ExprF** in place of **f**.

Exercise 4.8. Implement something of type **Algebra ExprF Double**. Hint: we already have above somewhere, so just find it and make sure it type checks. \diamond

Definition 4.9 (Morphism of F -algebras). Given two F -algebras $Fc \xrightarrow{\varphi} c$ and $Fd \xrightarrow{\psi} d$, an *algebra morphism* $f: (c, \varphi) \rightarrow (d, \psi)$ consists of a morphism $f: c \rightarrow d$ in \mathcal{C} such that the following diagram commutes:

$$\begin{array}{ccc} Fc & \xrightarrow{Ff} & Fd \\ \downarrow \varphi & & \downarrow \psi \\ c & \xrightarrow{f} & d \end{array}$$

You might have guessed what's coming. Since we were able to define mappings between algebras, chances are they form a category.

Proposition 4.10 (Category of F -algebras). For every endofunctor F there is a category $F\text{-Alg}$, whose objects are F -algebras, morphisms are morphisms of F -algebras, and whose composition and identities are given by those in \mathcal{C} .

Exercise 4.11. Prove that $F\text{-}\mathbf{Alg}$ is indeed a category. That is, check that the composite of two F -algebra morphisms is again an F -algebra morphism, that the identity is an F -algebra morphism, and then that this data obeys the unit and associative laws. \diamond

4.4 Initial algebras

Now that we have a category of F -algebras, we can ask whether it has an initial object (see Section 2.2.2). We'll see that, when it does, it ties the recursion knot we've been discussing.

Definition 4.12 (Initial algebra). An *initial algebra* for a functor F is an initial object in the category of F -algebras.

The above definition is a little terse, so let's unpack it. Given a functor $F: \mathcal{C} \rightarrow \mathcal{C}$, an initial algebra (i, φ) for F is an object i and morphism $\varphi: Fi \rightarrow i$ such that for every F -algebra $\psi: Fa \rightarrow a$ there exists a unique morphism $\text{cata}_{a,\psi}: i \rightarrow a$ in \mathcal{C} such that the following diagram commutes

$$\begin{array}{ccc} Fi & \xrightarrow{F(\text{cata}_{a,\psi})} & Fa \\ \varphi \downarrow & & \downarrow \psi \\ i & \xrightarrow{\text{cata}_{a,\psi}} & a \end{array}$$

In mathematical literature, you'll often find the notation μF for the carrier of the initial algebra for the functor F .

Definition 4.13 (Catamorphism). The unique morphism $\text{cata}_{a,\psi}: i \rightarrow a$ from the carrier of the initial algebra is called the *catamorphism* for $\psi: Fa \rightarrow a$.

Here we have an example of an initial object that is truly a universal producer. If you think of Fa as a list of ingredients to produce an a , and the structure map ψ as the recipe to produce it, then the object i is the “universal replicator” that, given this list and the recipe, can produce an a . The trick is that the same object i will work with any recipe. Continuing with this analogy, the initial algebra is a recipe φ for producing the universal replicator given a list of universal replicators. That's where the recursion kicks in.

From the programming point of view, this is the main reason for using algebras: to define recursive data structures. We start by defining a (non-recursive) functor, which serves as an inventory (a sum) of universal nodes that can store arbitrary items (e.g., **NilF** for an empty node and **ConsF** for a unary node). Then we define a recursive data type by plugging in, in the place of the arbitrary item, the very data type we are defining.

Example 4.14. Recall the list functor $1 + c \times a$

```
data ListF c a = NilF | ConsF c a
```

Let's construct the initial algebra for it. We'll call its carrier **List** *c* (it's parameterized by the type of the payload, *c*). Its structure map is a function

```
phi :: ListF c (List c) -> List c
```

There are two cases to consider:

```
phi NilF          = _
phi (ConsF c lst) = _
```

Both are supposed to produce a **List** *c*. We just make them into two constructors of **List** *c*

```
data List c = Nil | Cons c (List c)
```

The first, **Nil**, takes no arguments (or, equivalently, a unit argument); and the second, **Cons**, takes a *c* and a **List** *c*. The first constructs an empty list and the second prepends a new value to an existing list. Notice how the type argument *a* in the definition of **ListF** was replaced by the type we are constructing.

We can then implement **phi** in terms of these constructors

```
phi NilF          = Nil
phi (ConsF c lst) = Cons c lst
```

Example 4.15. Continuing with the previous example, let's focus on a list of integers and consider the following algebra with the **Int** carrier

```
evalSum :: Algebra (ListF Int) Int
evalSum (ConsF n x) = n + x
evalSum NilF = 0
```

If **List** **Int** is indeed the initial algebra, there must be a unique function from it to the carrier **Int**, the catamorphism, that uses **evalSum**

```
cataSum :: List Int -> Int
```



```

cataSum Nil          = evalSum NilF
cataSum (Cons n ns) = evalSum (ConsF n (cataSum ns))

```

This catamorphism calculates the sum of all the elements in the list.

Exercise 4.16. Implement a catamorphism for the following algebra

```

evalProd :: Algebra (ListF Int) Int
evalProd (ConsF n x) = n * x
evalProd NilF = 1

```

◇

4.4.1 Lambek's lemma

The evidence that initial algebras somehow tie the recursion knot is found in Lambek's lemma.

Theorem 4.17 (Lambek's lemma). The structure map of the initial algebra is an isomorphism.

Proof. Notice that if (i, φ) is an initial algebra, then $(Fi, F\varphi)$ is an algebra as well: $F\varphi: F(Fi) \rightarrow Fi$. Initiality tells us that there is a unique algebra morphism, meaning there is a unique function $h: i \rightarrow Fi$ making the following diagram commute:

$$\begin{array}{ccc}
 Fi & \xrightarrow{Fh} & F(Fi) \\
 \varphi \downarrow & & \downarrow F\varphi \\
 a & \xrightarrow{h} & Fi
 \end{array} \tag{4.18}$$

Our goal is to show that h is the inverse of the structure map φ . It's easy to show that φ itself is an algebra morphism, because the following diagram trivially commutes

$$\begin{array}{ccc}
 F(Fi) & \xrightarrow{F\varphi} & Fi \\
 F\varphi \downarrow & & \downarrow \varphi \\
 Fi & \xrightarrow{\varphi} & i
 \end{array}$$

Pasting the two commuting diagrams along the common arrow $F\varphi$, we arrive at the conclusion that the outer rectangle in the following diagram commutes:

$$\begin{array}{ccccc}
 Fi & \xrightarrow{Fh} & F(Fi) & \xrightarrow{F\varphi} & Fi \\
 \varphi \downarrow & & \downarrow F\varphi & & \downarrow \varphi \\
 i & \xrightarrow{h} & Fi & \xrightarrow{\varphi} & i
 \end{array}$$

Therefore, the composite $\varphi \circ h$ is an algebra morphism. Moreover, this is an algebra morphism between (i, φ) and itself. Initiality tells us that there can only be one such morphism. Since the identity morphism is an algebra morphism, we must have that $\varphi \circ h = \text{id}_i$.

To show that φ and h are inverse, we still need to show that $h \circ \varphi = \text{id}_{Fi}$. So consider Eq. (4.18) again. Starting with the commuting condition, we apply the functor laws and the previous result, $\varphi \circ h = \text{id}_i$:

$$\begin{aligned} h \circ \varphi &= (F\varphi) \circ (Fh) \\ &= F(\varphi \circ h) \\ &= F(\text{id}_i) = \text{id}_{Fi}. \end{aligned}$$

This completes the proof. □

We have shown that the initial algebra of an endofunctor F is a type i and an isomorphism between a and Fi . This ties the knot of recursion, because it means that i is a fixed point of F . Using the notation μF for the initial algebra, the fixed point condition can be written as

$$F(\mu F) \cong \mu F$$

4.5 Recursive data structures

Here's some intuition behind initial algebras. We have an endofunctor F that describes just one level of a recursive data structure. We've seen it applied to the case of lists, as well as to our original example of expression trees; see Section 4.1, where we discussed this:

```
data ExprF a = PlusF  a a
             | TimesF a a
             | ConstF Double
             | VarF   String
```

The functor **ExprF** generates leaves (**Double** and **String**) and nodes (for **PlusF** and **TimesF**). Each binary node contains a pair of placeholders of the same arbitrary type. The recursive version of the tree replaces these placeholders with the recursive versions of the tree. For instance, the node

```
PlusF a a
```

becomes

```
Plus Expr Expr
```

We also went through the exercise of manually increasing the allowed depth of a tree by stacking the applications of **ExprF** on top of **Void**. Think of this as consecutive approximation of the desired result, which is a full-blown recursive tree. If you are familiar with the Newton's method of calculating roots, this is a very similar idea. When you keep applying the same function over and over, you are getting closer and closer to a fixed point (under some conditions). A fixed point is a value with the property that one more application of the function doesn't change the value. Informally, the fixed point has been reached after applying the function infinitely many times, and infinity plus one is the same as infinity.

A fixed point S_F of an endofunctor F can be defined the same way. It's a solution to the equation

$$S_F \cong F(S_F)$$

What's wonderful is that this equation itself can be used as a Haskell definition of the corresponding data type, and that the resulting type is the initial algebra. Let's first write a simple version and then analyze the actual implementation from the library **Data.Fix**.

Here is the almost literal translation of the defining equation, except that Haskell requires us to explicitly name the data constructor:

```
data S f = MakeFix (f (S f))
```

This data type is parameterized by a type constructor **f**, which in all practical applications will be a **Functor**. The right-hand side is a data constructor that applies **f** to the fixed point we are in the process of defining. This is where the magic of recursion happens; it ties the recursion knot.

The beauty of this definition is that it decomposes the problem of defining recursive data structures into two orthogonal concerns: one abstracts recursion in general and the other is the choice of the particular shape we are going to recursively expand.

Before we go on, note that we get the inverse function quite easily:

```
unFix :: S f -> f (S f)
unFix (MakeFix a) = a
```

Haskell Note 4.19. Here's the definition of the fixed point type used in the library **Data.Fix**

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

It's equivalent to the one above, but it contains a lot of puns, so let's analyze it step by step. The left-hand side is a type constructor, which takes a type constructor **f** as its

argument, just like **S** did above. We use **newtype** in place of **data**, the only difference being related to performance: since **newtype** is more performant, we'll use it every time we are allowed to.^a

The right hand side is a data constructor, which is given the same name as the type constructor: **Fix**. We use the record syntax, so we don't have to define the accessor separately. This is as if we have defined it more explicitly

```
unFix :: Fix f -> f (Fix f)
```

Compare this with the type of the data constructor

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

^a**data** can be replaced by **newtype** if there is exactly one data constructor with exactly one field.

Given a functor F , its *least fixed point* is a fixed point i with $Fi \cong i$ that is “least” in the sense that it has a unique algebra map to any other fixed point. Certainly the carrier of the initial algebra has this property. Therefore, as long as **Fix** f is uniquely defined, we can use it for constructing initial algebras (and, later, terminal coalgebras).

An algebra, in Haskell, is defined by a functor **f**, the carrier type **a** and the structure map. This is neatly summarized in one type synonym which we saw on page 104:

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

Lambek's lemma tells us that the initial algebra is an isomorphism. Indeed, the structure map for the algebra whose carrier is **Fix** f has the type signature (replacing **a** with **Fix** f)

```
f (Fix f) -> Fix f
```

This is exactly the type signature of the data constructor **Fix**. Its inverse is the accessor **unFix**. In fact, any data type that is defined using **newtype** automatically establishes an isomorphism.

4.5.1 Returning to expression trees

Let's go back to our initial example, which was based on the following functor

```
data ExprF a = PlusF a a
             | TimesF a a
             | ConstF Double
             | VarF String
derive Functor
```

Haskell Note 4.20. In Haskell, most algebraic data structures have an instance for a **Functor** which automatically satisfies functor laws. In fact the compiler can derive functor instances automatically if, as we did here, we make `derive Functor` part of the type definition. This requires invoking the following pragma at the head of the file

```
{-# language DeriveFunctor #-}
```

We can define the fixed point for the above functor—like any functor—using **Fix**:

```
type Ex = Fix ExprF
```

This new data structure is fully equivalent to the original recursive definition of **Expr**, except that it requires a little more bookkeeping, the constructor **Fix** shows up repeatedly. This is why it's convenient to define *smart constructors* that take care of performing the appropriate incantations

```
var :: String -> Ex
var s = Fix (VarF s)

num :: Double -> Ex
num x = Fix (ConstF x)

mul :: Ex -> Ex -> Ex
mul e e' = Fix (TimesF e e')

add :: Ex -> Ex -> Ex
add e e' = Fix (PlusF e e')
```

We are using the data constructor **Fix** and passing it terms of the type **ExprF** *x*, where *x* is either irrelevant (leaves) or is of the type **Ex**.

With the help of these functions, we can recreate our original expression from page 100

```
-- 2 x^2 + 3 x + 4
ex'' = add (mul (num 2)
               (mul (var "x")(var "x")))
          (add (mul (num 3) (var "x")) (num 4))
```

Unlike before, we have one type **Ex**, rather than an infinite hierarchy of types **Expr1**, **Expr2**, **Expr3**, **Expr4**,.... This was made possible using initial algebras.

4.5.2 The essence of recursion

Recursion is a very useful tool in problem decomposition. When faced with a large problem we decompose it into smaller problems and try to solve them separately. Recursion happens when the smaller problems have the same “shape” as the bigger problem. We can then apply the same method to solve these smaller problems, and so on.

The key to implementing a recursive solution is to define a “recursive step.” Imagine that you have successfully solved all the subproblems. The recursive step takes all these solutions and combines them to obtain the solution to the current problem. Here’s the implementation of the workhorse of all recursive examples, the factorial. Notice that it’s written in a way that emphasizes the idea of the recursive step. The placeholder `a` is put in the exact place where the solution of the subproblem should magically appear.

```
fact n = if n <= 0
        then 1
        else n * a
  where
    a = fact (n - 1)
```

The subproblem, in this case, is the evaluation of the factorial of the predecessor of `n`.

The `where` clause in Haskell let’s us give names to local expressions or functions that are used elsewhere in a function. The `where` clause has access to the arguments of the function (here, `n`) and to everything defined in the global scope (here, the `fact` function itself).

The same idea works for recursive *data structures* in place of recursive *functions*. The functor (`ExprF` in our example) plays the role of a recursive step. It has placeholders for “solutions” to smaller subproblems. When we want to define our recursive expression tree, we bootstrap ourselves by assuming that we have solved the problem of defining smaller recursive trees and plugging them into the holes in our functor. This is the meaning of `f` (**Fix** `f`) in the definition of the initial algebra.

Now that we have defined recursive data structures, we can define recursive functions on them, applying the same strategy to the evaluation of the recursive expression. Suppose that we want to calculate a result that is a **Double**. The recursive step in this case is to assume that we have successfully evaluated the subexpressions, that is, we have filled the holes in our functor with **Doubles**. We have a functor-full of **Doubles**, i.e. a term of the type `ExprF Double`. All we have to do is to combine partial evaluations into one final result, in other words, provide a function

```
ExprF Double -> Double
```

But this is exactly what's needed to define an algebra **Algebra ExprF Double**. The algebra is precisely the recursive step that we were looking for. We also have a machine that accepts an algebra and cranks up the recursion, namely catamorphism. Let's discuss it again in the light of **Fix**.

4.5.3 Algebras, catamorphisms, and folds

Recall that an algebra for a functor **f** is defined using the carrier type **a** and the structure map

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

We stress again that the structure map is *not* a polymorphic function: the carrier **a** is fixed in the type constructor. In other words, there is no place to put **forall a**.

As mentioned above, an algebra defines a single recursive step **f a -> a**. What we need is a way to apply this algebra recursively to the recursive data structure **Fix f**, a fixed point of the functor **f**. This fixed point depends only on the functor **f**. In principle, it has no knowledge of the type **a**, which is the carrier of the algebra. Our goal is to extract a term of type **a** from a term of type **Fix f**. Since **Fix f** is the carrier of the initial algebra, we know that there is a unique algebra morphism to our algebra: the catamorphism from Definition 4.13. But knowing that the morphism exists is not the same as having a recipe for implementing it. And this is where the Lambek's lemma kicks in. Here's a diagram that combines the Lambek's lemma, the isomorphism between **f (Fix f)** and **Fix f**, and the definition of the algebra morphism we will call **cata alg**.

$$\begin{array}{ccc}
 f(\text{Fix } f) & \xrightarrow{\text{fmap } (\text{cata alg})} & f a \\
 \text{unFix} \updownarrow \text{Fix} & & \downarrow \text{alg} \\
 \text{Fix } f & \xrightarrow{\text{cata alg}} & a
 \end{array}$$

Because the diagram commutes, we can read it as

$$\text{cata alg} = \text{alg} \circ \text{fmap } (\text{cata alg}) \circ \text{unFix}$$

or, using Haskell notation

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

We can now finish our running example. Our recursive step for the functor **ExprF** is an algebra with the carrier **Double** and the evaluator (structure map) given by

```

eval1 :: Algebra ExprF Double
eval1 (ConstF x) = x
eval1 (VarF "x") = 2
eval1 (VarF _) = 0
eval1 (PlusF x y) = x + y
eval1 (TimesF x y) = x * y

```

We can use the catamorphism to evaluate the expression `ex''` from page 111:

```

> cata eval1 ex''
> 18.0

```

The catamorphism is a powerful tool in our toolbox. It takes care of the difficult part of defining recursive algorithms by abstracting the recursion. Defining an algebra, which is a non-recursive evaluator, is much simpler than implementing a full-blown recursive algorithm. In fact, it is often so much easier that it's worth rethinking an algorithm in terms of a data structure, rather than explicit flow of control. In a sense, a recursive data structure is a convenient visualization of an often complex flow of control.

4.6 Coalgebras, anamorphisms, and unfolds

As a rule of thumb, every notion in category theory has its dual, given by reversing the arrows. Do this to the notion of algebra $Fa \rightarrow a$ and you get that of coalgebra $a \rightarrow Fa$.

```

-- for any functor f
type Algebra    f a = f a -> a
type Coalgebra  f a = a -> f a

```

Just as an algebra can be thought as an evaluation—turning a data structure into a single value—a coalgebra can be thought of as generating a data structure from a seed. The carrier type `a` is the type of the seed. The idea of a coalgebra is that you are given a seed and you use it to create a single level of a recursive data structure. The single level is described by a functor, and the recursive data structure is described by its fixed point. In the process you create new seeds and plant them in the holes defined by the functor. The machinery that cranks the recursion is dual to catamorphism; it's called an anamorphism:

```

ana :: Functor f => Coalgebra f a -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg

```


The above Haskell code is clearly analogous to that defining catamorphism on page 113. Just like a catamorphism is the unique map from the initial algebra to your choice of algebra, an anamorphism is the unique map from your choice of coalgebra to the terminal coalgebra. A coalgebra morphism is defined by a commuting square analogous to the one for algebra morphisms but with the arrows reversed:

$$\begin{array}{ccc} Fc & \xrightarrow{Ff} & Fd \\ \varphi \uparrow & & \uparrow \psi \\ c & \xrightarrow{f} & d \end{array}$$

In terms of the mathematics, we have a dual version of Lambek's lemma that holds for coalgebras; it shows that the *terminal* F -coalgebra is a fixed point of F . The initial algebra is the *least fixed point* and the terminal coalgebra is the *greatest fixed point* of F . In mathematical literature, you'll often find the notation νF for the carrier of the terminal coalgebra. Lambek's lemma tells us that νF is a fixed point of F :

$$F(\nu F) \cong \nu F$$

It turns out that there is always a canonical morphism from the initial algebra to the terminal coalgebra, $\mu F \rightarrow \nu F$.

Exercise 4.21. Suppose that F is a functor. Use Lambek's lemma to show that

1. the initial F -algebra is also a coalgebra
2. the terminal F -coalgebra is also an algebra
3. the resulting catamorphism and anamorphism are the same: they define the canonical map $\mu F \rightarrow \nu F$.

◇

In **Set** this morphism is an injective function which embeds the elements of the (carrier set of) the initial algebra in the terminal coalgebra. This is not a bijection, though. In fact, in a category where the initial object is different from the terminal object there is a simple example of an endofunctor whose initial algebra is different from the terminal coalgebra.

Proposition 4.22. Consider the identity functor Id in \mathcal{C} . Its initial algebra has the initial object as its carrier, and its terminal coalgebra has the terminal object as its carrier.

Proof. For any object a , we have $Id\ a = a$. Thus the identity morphism id_a serves as both an Id -algebra and an Id -coalgebra with carrier a . A morphism between Id -algebras is just a morphism between their carriers, and similarly for Id -coalgebras.

Thus the category of Id -algebras is isomorphic to that of Id -coalgebras, and both are isomorphic to the category \mathcal{C} . In particular, the initial algebra is the initial object 0 , and the terminal coalgebra is the terminal object 1 . The unique morphism from the

initial object to the terminal object is the canonical mapping from the initial algebra to the terminal coalgebra. There is no guarantee though that the inverse mapping exists and, as a matter of fact, there isn't one in **Set**. \square

The situation in Haskell is different than in **Set**. We only have one definition of a fixed point combinator **Fix** and it works for both initial algebras and terminal coalgebras. This has to do with Haskell's laziness, and we'll come back to this point later. Because of laziness, it's perfectly okay to define a data structure whose terms stretch on forever. The data stored in such a structure is only evaluated when and to the extent that the program tries to access it. By using infinite data structures we can transform how we think about algorithms. Instead of coding a system of recursive functions, we can instead work on generating and traversing a single data structure. Let's see how it works in practice.

4.6.1 The type of streams, as a terminal coalgebra

The simplest non-trivial infinite data structure is a generalization of a list called a **Stream**. It is generated by a functor that drops the nullary list constructor **Nil**. It's the constructor which allows the recursion to terminate—therefore streams never terminate.

```
data StreamF a x = StreamF a x
    deriving Functor
```

This functor takes any type **x** and pairs it with **a**. The fixed point of this functor is the type of infinite streams

```
type Stream a = Fix (StreamF a)
```

We can expand this definition by tying the recursive knot—replacing **x** with the result of the recursion

```
data Stream a = MkStream a (Stream a)
```

We can map out from a **Stream a** by pattern matching on its constructor **MkStream**. The head of the stream is an **a** and the tail is another **Stream a**, of which the head is an **a** and the tail is another **Stream a**, of which... ad infinitum. Again, a stream never ends.

```
head :: Stream a -> a
tail :: Stream a -> Stream a

head (MkStream a s) = a
```

```
tail (MkStream a s) = s

-- in terms of Fix
head (Fix (StreamF a s)) = a
tail (Fix (StreamF a s)) = s
```

But how do we create a term of type `Stream a`? We can't possibly pre-fill it with infinitely many choices of `a`'s. This is where an anamorphism shows its usefulness. An anamorphism can (lazily) generate an infinite data structure from a seed.

Let's start with some anamorphisms that are built into the language. For instance, the infinite list of integers starting with 2 can be created using the syntax

```
intsFrom2 :: [Int]
intsFrom2 = [2..]
```

Exercise 4.23. Implement a function `intsFrom :: Int -> Stream Int` that uses an anamorphism to generate a `Stream` of integers starting from `n`. In other words `intsFrom 2` should be the same thing as `[2..]`. \diamond

Categorically, a stream is generated by the product functor $F_a x = a \times x$, where a is a fixed object. You might be wondering what the initial algebra is for this functor. Notice that, if we chose a to be the terminal object 1, this functor is equivalent to the identity functor (the terminal object is the unit of the product). We've seen before that, at least in **Set**, the initial algebra for this functor is different from the terminal coalgebra. In fact, in **Set** the initial algebra for the product functor is the empty set

Exercise 4.24. Show that the initial object 0 together with the identity morphism is the initial algebra for the functor $F_a x = a \times x$. (Hint: what is the product $a \times 0$?) \diamond

However, as we'll discuss soon, the situation is different in Haskell.

For now, let's move on to creating more interesting terms of type `Stream a`.

4.6.2 The stream of prime numbers

As an example of using streams in practice, let's generate the stream of all prime numbers using a version of the sieve of Eratosthenes. We start with a seed that is the list `[2..]` of all integers greater than one. Notice that it begins with a prime number. We'll keep it this way after every recursive step. First, we extract this number, then we eliminate all its multiples from the tail. This decimated list becomes our new seed. Notice that, again, it begins with a prime number.

We package this idea in a coalgebra `sieve` whose carrier is the type `[Int]` of all integer lists, the type of all the seeds we mentioned above.

```

IntStreamF = StreamF Int
sieve :: Coalgebra IntStreamF [Int]
-- sieve :: [Int] -> IntStreamF [Int]
sieve (p : ns) = IntStreamF p (filter (notdiv p) ns)
    where notdiv p n = n `mod` p /= 0

```

Let's analyze this code, since it contains some new syntax. The argument is a list, so we pattern match it to the infix constructor `:` (corresponding to **Cons** in the more verbose implementation of the list). If we were to release this code to the public, we would make the pattern matching total, and include the case of an empty list `[]`. Here, we just assume that nobody will dare to call us with a finite list. The result is a pair disguised as a stream functor constructor. Its first component is the prime number `p :: Int` from the head of the list. The second is a filtered tail, where `filter` is a library function that takes a predicate and passes through only those elements of the list that satisfy it.

```

filter :: (a -> Bool) -> [a] -> [a]

```

In our case we keep only those integers that are not divisible by the prime `p`. Of course, there are infinitely many of these, but this does not bother us because we will never need to compute all of them. The predicate `notdiv` is implemented in the `where` clause. It performs division modulo and compares the result to zero using the inequality² operator `/=`. In Haskell you can use a two-argument function in infix notation if you surround it with inverted single quotes, as we did here with the function `mod`.

By applying the anamorphism to this coalgebra `sieve` we get a map `[Int] -> Stream Int`. That is, from an infinite list of integers we generate an infinite stream. In particular, from the list `[2..]`, we generate the list of prime numbers:

```

primes = ana sieve [2..]

```

If you want to display this stream, you'd probably want to convert it to a list first. This can be done using an algebra. That's because **Stream** `a` is not only a terminal coalgebra but also an initial algebra.

Exercise 4.25. Implement a function that converts a **Stream** to an (infinite) list

```

toList :: Stream a -> [a]

```

Hint: Implement an algebra

²In many programming languages this operator is encoded as `!=`.

```
alg :: Algebra (StreamF a) [a]
```

and apply a catamorphism to it. ◇

In order to display some primes, use the function `take` which truncates the (possibly infinite) list down to a chosen size.

```
Prelude> take 10 (toList primes)
[2,3,5,7,11,13,17,19,23,29]
```

This pattern of applying a catamorphism immediately after an anamorphism is common enough to deserve its own function called a *hylomorphism*.

```
hylo :: Functor f => Algebra f a -> Coalgebra f b -> b -> a
hylo f g = f . fmap (hylo f g) . g
```

Haskell Note 4.26. In many cases the use of a hylomorphism results in better performance. Since Haskell is lazy, the data structure that the algebra consumes is generated on demand by the coalgebra. The parts of the data structure that have already been processed are then garbage collected, freeing the memory to be used to expand new parts of it. This way it's possible that the complete data structure is never materialized in memory and its structure serves as a scaffolding for directing the flow of control. It's often easier to imagine a flow of control as a data structure, rather than a network of mutually recursive function calls.

Exercise 4.27 (Merge sort). Implement merge sort using a hylomorphism. Here's the idea: The seed (the carrier of the coalgebra) is the list to be sorted. Use this function

```
split :: [a] -> ([a], [a])
split (a: b: t) = (a: t1, b: t2)
  where
    (t1, t2) = split t
split l = (l, [])
```

to split the list into two lists and use them as new seeds. Make sure you deal correctly with empty lists.

The carrier of the algebra is again a list (this time it's actually a sorted list, but this cannot be reflected in the type). Your partial results are sorted lists. You combine them using this function.

```

merge :: Ord a => [a] -> [a] -> [a]
merge (a: as) (b: bs) =
  if a <= b
  then a : merge as (b: bs)
  else b : merge (a: as) bs
merge as [] = as
merge [] bs = bs

```

Make sure your program also works for empty lists (it should return an empty list).

◇

4.7 Fixed points in Haskell

So far we've been using the same **Fix** type constructor to generate both the initial algebras and terminal coalgebras. One could argue that **Fix** generates greatest fixed points, so it can be legitimately used to generate terminal coalgebras, but how can it be used to also generate initial algebras? We've seen that, in many categories, including **Set**, the carriers of initial algebras are not necessarily the same as those of the terminal coalgebras for the same functor, see Proposition 4.22. The intuition is that initial algebras describe finite data structures like lists or trees, whereas terminal coalgebras include infinite data structures like streams or infinite trees.

Let's look at a simple example. Let's implement a geometric sequence with a ratio **a**. To that end, we define a stream of **Double** using the following coalgebra

```

geomCoa :: Double -> Coalgebra (StreamF Double) Double
geomCoa a x = StreamF x (a * x)

```

Here **x** is the current seed. We set the seed for the tail of the stream to be **a * x**.

Applying the anamorphism to this coalgebra we can generate an infinite sequence, a geometric progression:

```

geom :: Double -> Stream Double
geom a = ana (geomCoa a) 1

```

But how about using the same fixed point as an initial algebra? We've seen before that, in **Set**, the initial algebra for the pair functor is empty. But here's an example of an algebra for this functor.

```

toListAlg :: Algebra (StreamF a) [a]
toListAlg (StreamF a as) = a : as

```

for which we can implement a catamorphism from **Fix** (**StreamF** a)

```
toList :: Stream a -> [a]
toList = cata toListAlg
```

This works, because of Haskell’s laziness. This catamorphism converts one lazy data structure to another lazy data structure and, as long as we don’t try to demand it all at once, we can look at bits of it. For instance, we can compose it with `take 10` to retrieve the first ten elements.

But here’s another algebra that ads the head to the tail of the stream:

```
sumAlg :: Algebra (StreamF Double) Double
sumAlg (StreamF a s) = a + s
```

In Haskell we can implement a catamorphism for this algebra:

```
-- don't run it!
sumAll = cata sumAlg
```

This catamorphism calculates the sum of the infinite series. It’s a legitimate Haskell function. It compiles and runs. Granted, if you try to print or pattern-match the result, it will run forever, but this is the price of doing business in any Turing complete language. This function “returns” an element of the type **Double**—the bottom, \perp .

With this caveat, in Haskell, the same fixed point combinator **Fix** generates initial algebras and terminal coalgebras. However, it’s also possible to encode the two fixed points, the least one and the greatest one, separately.

4.7.1 Implementing initial algebras by universal property

Once again, the trick is to use the universal property directly. The initial algebra can be defined by its mapping out property.

```
{-# language RankNTypes #-}
newtype Mu f = Mu (forall a. (f a -> a) -> a)
```

This definition works because for every least fixed point one can define a catamorphism, which can be rewritten as

```
cata :: Functor f => Fix f -> (forall a . (f a -> a) -> a)
cata (Fix x) = \alg -> alg (fmap (flip cata alg) x)
```

Haskell Note 4.28. The function `flip` simply reverses the order of arguments of its (function) argument.

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f b a = f a b
```

What the definition of **Mu** is saying is that it's an object that has a map to the carrier **a** of any algebra **f a -> a**; that map is of course the catamorphism.

It's easy to define a catamorphism in terms of **Mu**, since **Mu** is quite literally the very essence of catamorphism

```
cataMu :: Functor f => Algebra f a -> Mu f -> a
cataMu alg (Mu cata) = cata alg    -- cata :: forall a. (f a -> a) -> a
```

The challenge is to construct terms of type **Mu f**. This is not as straightforward as constructing terms of the type **Fix f**, but it's feasible. For instance, let's convert a list of **a**'s to a term of type **Mu (ListF a)**

```
mkList :: forall a. [a] -> Mu (ListF a)
mkList as = Mu cata
  where cata :: forall x. (ListF a x -> x) -> x
        cata unf = go as
        where
          go [] = unf NilF
          go (n: ns) = unf (ConsF n (go ns))
```

This Haskell code is a little tricky because we have to use the type **a** defined in the type signature of `mkList` to define the type signature of the helper function `cata`. For the compiler to identify the two, we have to use the pragma

```
{-# language ScopedTypeVariables #-}
```

This pragma extends the scope of the definition of **a** to the whole body of the function.

You can now verify that

```
cataMu myAlg (mkList [1..10])
```

produces the correct result for the following algebra

```
myAlg :: Algebra (ListF Int) Int
myAlg NilF = 0
myAlg (ConsF a x) = a + x
```


4.7.2 Implementing terminal coalgebras by universal property

The terminal coalgebra, on the other hand, is defined by its mapping in property. This requires a definition in terms of existential types. If Haskell had an existential quantifier, we could write the following definition for the terminal coalgebra

```
data Nu f = Nu (exists a. (a -> f a, a))
```

When somebody gives you a value of an existential type, they guarantee that the type `a` exists, without specifying what it is. They usually provide some way of working with it. Here, you are given a function `a->f a` and some value of the type `a`. This situation is somewhat familiar to object-oriented programmers. They are often given an object that hides some data, but it provides “methods” that can be used to operate on it.

Existential types can be encoded in Haskell using the so called Generalized Algebraic Data Types or GADTs

```
{-# language GADTs #-}
data Nu f where
  Nu :: (a -> f a) -> a -> Nu f
```

Again the idea is that for every greatest fixed point one can define an anamorphism

```
ana :: Functor f => forall a. (a -> f a) -> a -> Fix f
ana coa x = Fix (fmap (ana coa) (coa x))
```

We can uncurry it

```
ana :: Functor f => forall a. (a -> f a, a) -> Fix f
ana (coa, x) = Fix (fmap (curry ana coa) (coa x))
```

A universally quantified mapping out

```
forall a. ((a -> f a, a) -> Fix f)
```

is equivalent to a mapping out of an existential type (in pseudo-Haskell)

```
(exists a. (a -> f a, a)) -> Fix f
```

which is the type signature of the constructor of `Nu f`.

The intuition is that, if you want to implement a function from an existential type—a type which hides some other type `a` to which you have no access—your function has to be prepared to handle any `a`. In other words, it has to be polymorphic in `a`.

Since in an existential type we have no access to the hidden type, it has to provide both the “producer” and the “consumer” for this type. Here we are given a value of type `a` on the produces side, and the function `a -> f a` as the consumer. All we can do is to apply this function to `a` to obtain the term of the type `f a`. Since `f` is a functor, we can lift our function and apply it again, and get something of the type `f (f a)`. Continuing this process, we can obtain arbitrary powers of `f` acting on `a`. We get a recursive data type.

An anamorphism in terms of `Nu` is given by

```
anaNu :: Functor f => Coalgebra f a -> a -> Nu f
anaNu coa a = Nu coa a
```

Notice however that we cannot directly pass the result of `anaNu` to `cataMu` because it’s no longer obvious that the initial algebra is the same as the terminal coalgebra for a given functor. A hylomorphism relies on the fact that, in Haskell, we can identify initial algebras with terminal coalgebras.

Monads

5.1 A teaser

Starting with the category of types and functions, it's possible to construct new categories that share the same objects (types), but redefine the morphisms and their composition.

A simple example is the category with the same types but only partial computations. These are computations that are not defined for all values of their arguments. We can model such computations using the **Maybe** data type.

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

A partial computation from **a** to **b** can be implemented as a regular old function

```
a -> Maybe b
```

When the partial computation succeeds, this function wraps its output in **Just**, otherwise it returns **Nothing**. Most programming languages have some equivalent of **Maybe** (often called *option* or *optional*), or use exceptions to implement partial computations.

What we are trying to do here is to create a new category that we will call **Kl(Maybe)**. This category has the same objects as **Hask**, but its morphisms are different. Let's denote morphisms in **Kl(Maybe)** using a tick $f : a \rightarrow b$. Then a morphism $f : a \rightarrow b$ in **Kl(Maybe)** is represented by a function $f :: a \rightarrow \text{Maybe } b$ in Haskell.

To define a category, we have to define objects, morphisms, identities, and composition. We've already said we want our category to have the same objects as **Hask** and the above sort of maybe morphisms. So let's consider composition.

The issue is that two composable Kleisli morphisms $f : a \rightarrow b$ and $g : b \rightarrow c$ don't quite look composable when you unwind the definitions. They are represented by morphisms $f :: a \rightarrow \text{Maybe } b$ and $g :: b \rightarrow \text{Maybe } c$, and these aren't directly composable in **Hask**; the codomain of f isn't quite the domain of g . Before we say how to deal with this, let's write out what we want, both mathematically and in Haskell.

Mathematically, given $f: a \rightarrow b$ and $g: b \rightarrow c$, we want something of type $a \rightarrow c$. In Haskell, just like the usual sort of composition can be regarded as a function that takes two functions as input

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

and produces their output, we need a function that takes two Kleisli morphisms and produces their composite. That is we need something of the following type:

```
(<=<) :: (b -> Maybe c) -> (a -> Maybe b) -> (a -> Maybe c)
```

Here is a way we could implement this: if the first function returns **Nothing**, don't call the second function. Otherwise, call it with the contents of **Just**

```
g <=< f = \a -> case f a of
    Nothing -> Nothing
    Just b   -> g b
```

If g and f are representatives of two composable morphisms in $\mathbf{Kl}(\text{Maybe})$ then $g \leq f$ produces a representative of their composition. This Kleisli composition operator is often called the *fish*. A Kleisli identity is called *return*.

Next we need an identity morphism. It is represented by

```
idMaybe :: a -> Maybe a
idMaybe a = Just a
```

We will see in Exercise 5.2 that this identity is indeed unital with respect to fish composition and that the fish composition is also associative. Therefore $\mathbf{Kl}(\text{Maybe})$ is indeed a category; it's called a Kleisli category for the functor **Maybe**. It's a very useful category that allows us to compose partial functions.

Proposition 5.1. There is a category $\mathbf{Kl}(\text{Maybe})$ whose objects are sets, whose morphisms $a \rightarrow b$ are functions $a \rightarrow b + \underline{1}$, whose identity $\text{id}_a: a \rightarrow a$ is the inclusion $\eta_a: a \rightarrow a + \underline{1}$, and for which the composite of $f: a \rightarrow b$ and $g: b \rightarrow c$ is given by

$$a \xrightarrow{f} b + \underline{1} \xrightarrow{g + \underline{1}} (c + \underline{1}) + \underline{1} \cong c + (\underline{1} + \underline{1}) \xrightarrow{c + \underline{1}} c + \underline{1}$$

Proof. See Exercise 5.2. □

Exercise 5.2. Show that $\mathbf{Kl}(\text{Maybe})$ is indeed a category by proving unitality and associativity:

1. For any Kleisli morphism $f: a \rightarrow b$, one has $f \circ \text{id}_a = f$.

2. For any Kleisli morphism $f: a \rightarrowtail b$, one has $\text{id}_b \circ f = f$.
3. For any three Kleisli morphisms $f: a \rightarrowtail b$, $g: b \rightarrowtail c$, and $h: c \rightarrowtail d$, the equation $(h \circ g) \circ f = h \circ (g \circ f)$ holds.

◇

5.2 Different ways of working with monads

5.2.1 Monads in terms of the “fish”

This procedure of constructing a Kleisli category can be generalized to functors other than **Maybe**, as long as we can generalize the notion of identities and composition used above. What was it that made these definitions work?

It turns out that the laws that made $\mathbf{Kl}(\text{Maybe})$ work as a category are captured in the statement that the functor $a \mapsto a + \underline{1}$ can be given the structure of a *monad*. Monads are things that mathematicians did not know about before category theory, but they formalize a great deal of structure, e.g. the notion of algebraic theory can be formulated completely in terms of monads.

In Haskell, the **Monad** type class is defined in the Prelude, but the following definition using Kleisli arrows is equivalent

```
class Functor m => Monad m where
  (<=<) :: (b -> m c) -> (a -> m b) -> (a -> m c)
  return :: a -> m a
```

This says that any functor can be a monad, as long as you tell it what the fish and return are.

In order for the Kleisli category to really be a category, these definitions need to satisfy a couple laws. These laws would sure that the composition is associative and also unital with respect to return. Luckily the laws are very easy to formulate: they are just the laws of a category—associativity and unitality

```
-- one checks mentally that the following equations hold:
(h <=< g) <=< f = h <=< (g <=< f)
f <=< return    = f
return <=< f    = f
```

5.2.2 Monads in terms of join

Monads can also be defined in another, way more typical in mathematics.

Definition 5.3. Let \mathcal{C} be a category. A *monad* on \mathcal{C} consists of a tuple (M, η, μ) where

$M: \mathcal{C} \rightarrow \mathcal{C}$ is a functor, and $\eta: \text{id}_{\mathcal{C}} \rightarrow M$ and $\mu: M \circ M \rightarrow M$ are natural transformations

$$\begin{array}{ccc} \mathcal{C} & \xrightarrow{\text{id}_{\mathcal{C}}} & \mathcal{C} \\ & \Downarrow \eta & \\ \mathcal{C} & \xrightarrow{M} & \mathcal{C} \end{array} \quad \begin{array}{ccc} \mathcal{C} & \xrightarrow{M} & \mathcal{C} \\ & \Downarrow \mu & \\ \mathcal{C} & \xrightarrow{M} & \mathcal{C} \end{array}$$

such that the following diagrams commute:

$$\begin{array}{ccc} M & \xrightarrow{M \circ \text{id}_{\mathcal{C}}} & M \circ M \\ & \searrow & \downarrow \mu \\ & & M \end{array} \quad \begin{array}{ccc} M \circ M \circ M & \xrightarrow{M \circ \mu} & M \circ M \\ \mu \circ M \downarrow & & \downarrow \mu \\ M \circ M & \xrightarrow{\mu} & M \end{array}$$

Example 5.4 (The List monad). In Haskell, we denote the type of **a**-lists by `[a]`; for the math, let's write $\text{List}(A)$. That is, A is a set and $\text{List}(A) = 1 + A + A^2 + \dots$ is the set of lists—of arbitrary finite length—for which every element is in A . It turns out that List is the functor part of a monad on **Set**, namely (List, s, f) where $s: \text{id}_{\text{Set}} \rightarrow \text{List}$ is called *singleton* and $f: \text{List} \circ \text{List} \rightarrow \text{List}$ is called *concatenate*.

We saw that List is a functor in ??; it means that you can map any function, say $g: A \rightarrow B$ over a list of A 's to get a list of B 's with the same length; the function g is applied to each spot.

To give a natural transformation, one first gives the components and then checks that they satisfy the naturality condition. For any set A , we define $s_A: A \rightarrow \text{List}(A)$ to be the function sending $a \in A$ to the singleton list $[a] \in \text{List}(A)$. This is natural because for any function $g: A \rightarrow B$, the following commutes:

$$\begin{array}{ccc} A & \xrightarrow{g} & B \\ s_A \downarrow & & \downarrow s_B \\ \text{List}(A) & \xrightarrow{\text{List}(g)} & \text{List}(B) \end{array}$$

This says that starting with an element $a \in A$, one can either make the singleton list and then map g over it or apply g directly and make the singleton list; the results in both cases are the same. Thus we have given our natural transformation s .

Next we give f , the components and naturality of the concatenate transformation. For any set A we define $f_A: \text{List}(\text{List}(A)) \rightarrow \text{List}(A)$ to be the function sending a list of lists of A 's to the concatenated list, e.g.

$$f([[a_1, a_2], [], [a_2, a_3, a_1]]) = [a_1, a_2, a_2, a_3, a_1].$$

The mathematics of the concatenate map are given in Exercise 5.5. It is natural because for any function $g: A \rightarrow B$ and any list of lists, one can either concatenate it to a list

and then map g over it, or map g over each individual list—in fact over each element in each individual list—and then concatenate; the results in both cases are the same.

Now that we have our natural transformations s and f , the last thing to do is check that the diagrams from Definition 5.3 commute. The first is actually two: it says that if you start with a list ℓ and then singletonize every entry in it, or singletonize the entire list—in each case getting a list of lists—and then concatenate, you get back where you started. For example, start with $[a, b, c]$. Singletonizing each entry gives $[[a], [b], [c]]$, and when you concatenate you get back $[a, b, c]$. Similarly, singletonizing the entire list gives $[[a, b, c]]$ and when you concatenate you get back $[a, b, c]$. Thus the first diagram commutes.

The second diagram says if you start with a list of lists of lists and then concatenate the first inner layer then the second inner layer, it's the same as concatenating the second inner layer then the first inner layer. For example, starting with $[[[a, b], [a, c, c]], [], [b, b]]$, one can remove the innermost brackets to get $[a, b, a, c, c], [b, b]$ and then concatenating again $[a, b, a, c, c, b, b]$. This is the same as what you get by removing the middle brackets first, $[[a, b], [a, c, c], [], [b, b]]$ and then removing the innermost brackets $[a, b, a, c, c, b, b]$.

So List is a monad!

Exercise 5.5. The set $\text{List}(A)$ is given by the following disjoint union of products:

$$\text{List}(A) := \coprod_{n \in \mathbb{N}} A^n = \coprod_{n \in \mathbb{N}} \prod_{i \in \underline{n}} A = A^n 1 + A + A^2 + A^3 + \dots$$

where $A^n = A \times A \times \dots \times A$. Recalling that products distribute over coproducts—even infinite coproducts—in **Set**,

$$A \times \prod_i B_i \cong \prod_i A \times B_i$$

formally define the concatenate function $\text{List}(\text{List}(A)) \rightarrow \text{List}(A)$. \diamond

Now let's discuss how to think about this from a programming point of view. We said that **Maybe** is a monad. Let's abstract it and consider a functor **m**; what structures does it have that make composition work right?

If $f: a \rightarrow b$ and $g: b \rightarrow c$ are Kleisli arrows, i.e. functions $f :: a \rightarrow m\ b$ and $g :: b \rightarrow m\ c$, how can we compose them? The first piece we need is that **m** is a functor, i.e. it has an instance of **fmap**. Let's start a program for the fish, leaving a hole for what we don't yet know:

```
g <=< f = \a -> let mb = f a      -- f :: a -> m b,  g :: b -> m c
                mmc = fmap g mb -- (fmap g) :: m b -> m m c
                in _ mmc        -- not sure what to do with mmc yet
```

After applying the `fmap` and composing, we got something of type `m m c`, whereas we need something of the type `m c`. This happened with the `Maybe` monad too; in Proposition 5.1 one can see that we first applied f , then $\text{Maybe}(g) = g + \underline{1}$, and we were left with something of type $\text{Maybe}(\text{Maybe}(c)) = c + \underline{1} + \underline{1}$. Our next step was and again is to find some function `m (m a) -> m a`; then we could implement the Kleisli arrow. Hence we arrive at the definition of a monad given in Definition 5.3; in Haskell its parts are:

```
class Functor m => Monad m where
  join :: m (m a) -> m a
  return :: a -> m a
```

We've already said that `return` is the Haskell name for what we called η ; similarly `join` is the Haskell name for μ . One can derive the laws for this definition from the laws for the Kleisli definition, or again just see Definition 5.3.

Now we can finish the above definition

```
g <=< f = \a -> let mb = f a
                 mmc = fmap g mb
                 in join mmc           -- join :: m m c -> m c
```

5.2.3 Monads in terms of bind

In the above definition of Kleisli composition, or the following even shorter description

```
g <=< f = \a -> join (fmap g (f a))
```

we used both `join` and `fmap`. Let's rewrite it as a single function

```
bind :: m b -> (b -> m c) -> m c
bind mb g = join (fmap g mb)
```

This new function leads to yet another definition of `Monad` (with some minor renaming):

```
class Monad m where
  (>=) :: m a -> (a -> m b) -> m b  -- this is `bind`
  return :: a -> m a
```

The `bind` function is represented by an infix operator. This definition imposes no `Functor` constraint because it turns out that `fmap` can be constructed from `bind` and `return`. It is implemented as


```
liftM :: Monad m => (a -> b) -> m a -> m b    -- this is fmap
liftM f ma = ma >>= (\a -> return (f a))
```

At one time this was the official definition of **Monad** in Haskell, until the additional constraint of **Applicative** was added. We'll discuss this more later in ??.

Just like we can extract **fmap** from **bind**, we can also extract **join**; indeed this is done in the library **Control.Monad**:

```
join :: Monad m => m (m a) -> m a
join mma = mma >>= id
```

5.2.4 Monads in terms of the **do** notation

The bind operator takes a monadic value and applies a Kleisli arrow to it; for example it takes a **ma :: Maybe a** and applies a partial function **a -> Maybe b**, or it takes a list **l :: [a]** and applies a non-deterministic function **a -> [b]**. One can build programs by composing Kleisli arrows and applying them to monadic values. This way one can essentially develop point-free notation in the Kleisli category. But point-free notation is often difficult to read, especially if Kleisli arrows are created inline using lambdas. Look for instance at the definition of **liftM**; we could have made this even more point free:

```
liftM1 f ma = ma >>= (\a -> return (f a))    --original
liftM2 f = (>>= (\a -> return (f a)))        --using sections
liftM3 f = (>>= return . f)                  --using composition
liftM4 = flip (>>=) $ (return .)             --fully point-free
```

This is slick and shows that **liftM** really uses nothing but the monad structures, but again it is not generally considered easy to read **liftM4**, even for experts.

The **do** notation goes in the opposite direction, using variables to help us keep track of the computation process in stages. It has a distinctly imperative look:

```
a <- ma
```

The left arrow notation suggests assignment, somehow extracting **a** from the monadic argument **ma**. We'll read it as "a takes from ma." The extracted variable is usually later used in the body of the **do** block. For example, here is **liftM** written in **do** notation

```
liftM f ma = do
    a <- ma      -- "a takes from ma"
    return (f a) -- apply f to it
```

A Haskell **do** block has an implicit monad; every line in the block will implicitly involve that monad. The above code works for every monad, but some code works only for a particular monad **m**, e.g. the following which only makes sense for the **IO** monad:

```
main :: IO ()
main = do
  putStrLn "Tell me your name"
  -- _ <- putStrLn "Tell me your name"
  name <- getLine
  putStrLn ("Hello " ++ name)
```

All the lines of the **do** block have a similar structure, except for the last line. Indeed, all but the last line has the form **newvar <- ma**, where **ma :: m a** and **newvar :: a**. Sometimes **newvar** is dropped from the notation and one simply writes **ma**; this occurs when its value is of no consequence and will never be used again, i.e. it is implicitly **_ <- ma**. In particular, in the case of unit type **a = ()**, the value is never of any consequence, since **()** has only one value. Finally, the last line is of the form **m x**, the type of the whole **do** block.

One can see these rules at work in the **do** code for **liftM** on page 131. Let's take a closer look in the case of **main** just above by first checking the types:

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

```
Prelude> :t getLine
getLine :: IO String
```

Since **getLine :: IO String**, we must have that **name :: String** by our rules above; this is why we can append **name** to **"Hello "**.

Here's yet another example, saying for every pair of monadic values there is a monadic pair:

```
pairM :: Monad m => m a -> m b -> m (a, b)
pairM ma mb = do
  a <- ma      -- "a takes from ma"
  b <- mb      -- "b takes from mb"
  return (a, b) -- return the pair
```

Here's what it does on lists:

```
prelude> pairM [1,2,3] ["a", "b"]
[(1, "a"),(1,"b"),(2, "a"),(2,"b"),(3, "a"),(3,"b")]
```

The comments in quotes, e.g. `-- "a takes from ma"` represent a surprisingly useful fiction for thinking about these lines. If `m` is a monad and `a` is a type, it doesn't quite make mathematical sense to take values from some `ma :: m a`. But when `m` is the list monad or the **Maybe**, **IO**, or any other monad, this fiction does not seem to misguide us. Still, it will be grounding for us to know what is actually happening behind the scenes when we strip off the syntactic sugar from a **do** block.

What's really happening in a **do block** To begin, let's reiterate the pattern of code in **do** blocks: every line but the last is implicitly of the form `newvar <- ma` with `newvar :: a` and `ma :: ma` for some `a`, and the last line is of the form `m x`, which is the type of the whole **do**-block.

Here's how you de-sugar this into a series of **binds** and lambdas. First take every non-last line `newvar <- ma` and rewrite it as `ma >>= \newvar;` leave the last line as it is.

For example, following this procedure, the above `pairM` function would be rewritten as follows:

```
pairM ma mb =           -- do
  ma >>= \a ->           --   a <- ma
    mb >>= \b ->         --   b <- mb
      return (a, b)      --   return (a, b)
```

This isn't just a composite of binds `>>=`; it's an intricately nested structure. It involves a function `a -> m b -> m (a, b)`, which is known as a *strength* for the functor `m`.

```
strengthM :: a -> m b -> m (a, b)
strengthM a mb = do
  b <- mb
  return (a, b)
```

Although it seems to use bind, it actually is just `liftM = fmap`; see the code on page 131 and also Exercise 5.6. Going back to `pairM`, we're given `mb :: m b` so we get a map `a -> m (a, b)`, and then we can bind that one to `ma`, obtaining the desired result.

Exercise 5.6. Let `f` be a functor.

1. Load `{-## Language TupleSections ##-}` and find the type of `\a -> (a,)`.
2. Use it to construct a map `a -> f b -> f (a, b)`.
3. Explain why it does the same thing as **StrengthM**.

◇

The point is that **do** blocks are quite easy to read by thinking of imperative style programming: do this, then do this, etc. But behind the scenes there's a lot of theory at work!

5.2.5 Monads and effects

The introduction by Eugenio Moggi ([Moggi]) of monads into functional programming was prompted by the need to provide semantics to functions that have side effects. A function $f: A \rightarrow B$ simply takes every element of A to an element of B ; what does it mean to have it based on user input, or to have it display something while it's processing, or to have it use a random number generator or some internal state? All of these things can be used in normal programming languages, so functional programmers felt left out! Of course, they had purity on their side, which came with the ability to reason about their programs in ways that others couldn't. But by introducing monads, functional programmers could rejoice because now they had nothing to envy from imperative programmers.

Here are some of the effects that are important in programming but can't be directly modeled as pure functions.

- Partiality: Computations that may not terminate
- Exceptions: Partial functions that may fail in various ways
- Nondeterminism: Computations that may return many results
- Side effects: Computations that access/modify state
 - Read-only state, i.e. a file with parameters for this run of the program
 - Write-only state, i.e. a log in which to record data about this run
 - Read/write state, i.e. variables that can be used or modified at any time
- Continuations: computations on programs with fixed output type
- Interactive Input and output

It turns out that each of them can be implemented as Kleisli arrows for an appropriate monad. The monad for partial functions is **Maybe**, which we've discussed above: a Kleisli arrow $a \rightarrow \text{Maybe } b$ is a partial function from a to b . In fact, this is a special case of an exceptions monad.

Let's look at some of these.

5.3 Examples of monads

5.3.1 The exceptions monads

Let E be a set. There is a functor $\mathbf{Set} \rightarrow \mathbf{Set}$ sending $X \mapsto X + E$. For example, if $E = \underline{1}$ then this functor adds a single new element to any set. It's a functor because for any function $f: X \rightarrow Y$, there is an induced function $f + E: X + E \rightarrow Y + E$ that takes any element $a \in X + E$ and applies f if $a \in X$ and applies id_E if $a \in E$.

To make this into a monad, we need to give a unit (return) $\eta_X: X \rightarrow X + E$ and multiplication (join) $\mu_X: X + E + E \rightarrow X + E$. The map η_X is just the coproduct inclusion, sending any element of x to itself $x \mapsto x$, and the map μ_X sends $x \mapsto x$ and any element of E , whether it's in the first or second component, to itself $e \mapsto e$.

In Haskell, this monad would look as follows:

```

data Exception e a = Just a | Exception e
instance Functor (Exception e) where
  fmap f (Just a) = Just (f a)
  fmap f (Exception e) = Exception e

instance Monad (Exception e) where
  (Just a)      >>= f = Just (f a)
  (Exception e) >>= f = Exception e
  return a = Just a

```

5.3.2 The list monad and nondeterminism

A computation that may return more than one possible output for the same input can be represented in Haskell by returning a list of outputs for each input; let's call these non-deterministic functions. When composing two nondeterministic functions we want to apply the second non-deterministic function to every possible outcome of the first non-deterministic function. Let's have the identity function be deterministic, i.e. produce one output; this is represented by a singleton list. We discussed this mathematically in Example 5.4. In code, we have

```

instance Monad [] where
  as >>= k = concat (fmap k as)
  return a = [a]

```

Exercise 5.7. Write a function `concat` that concatenates a list of lists into a list

```
concat :: [[a]] -> [a]
```

◇

Now that we know that `[]` is a monad, we can use `do`-notation. The following program produces a cartesian product of two lists using the `do` notation for the list monad

```

pair :: [a] -> [b] -> [(a, b)]
pair as bs = do
  a <- as
  b <- bs
  return (a, b)

```

```
Prelude> pair [1,2,3] ["a","b"]
[(1,"a"),(1,"b"),(2,"a"),(2,"b"),(3,"a"),(3,"b")]
```

```
Prelude> do a <- [1,2,3]; b <- ["a", "b"]; return (a,b)
[(1,"a"),(1,"b"),(2,"a"),(2,"b"),(3,"a"),(3,"b")]
```

5.3.3 The writer monads

Recall that a monoid is a set M together with an element $e \in M$ and a binary operation $(*) : M \times M \rightarrow M$, such that $e * m = m = m * e$ and $m * (n * p) = (m * n) * p$ for all $m, n, p \in M$. For any monoid $(M, e, *)$, there is an associated monad on **Set**. It sends each set X to the set $M \times X$. This is functorial: given $f : X \rightarrow Y$, there is an obvious map $(M \times f) : (M \times X) \rightarrow (M \times Y)$. It is a monad because it has a natural unit (return) and multiplication (join).

In Haskell this is called the writer monad (for M). The idea is that a Kleisli arrow $X \rightarrowtail Y$ is a function $X \rightarrow M \times Y$; it takes an $x \in X$ and outputs both an element of M and an element of Y . Think of the element of M as an entry in a log file; for example if M is the set of strings, e is the empty string, and multiplication is just concatenation of strings. When composing Kleisli arrows, we concatenate the log entries, adding a new line to the file.

```
data Writer m a = Writer m a
    deriving Functor
```

Recall that a monoid in Haskell is something that has implemented the above unit and multiplication as above, though they are called `mempty` and `mappend`, also written `(<>)`. Here's the implementation:

```
instance Monoid [a] where -- for example String = [Char]
    mempty = []           -- unit = empty string
    (<>)    = (++)         -- mult = concatenate strings
```

```
instance Monoid m => Monad (Writer m) where
    (Writer m a) >>= k = Writer (m <> m') a'
    where Writer m' a' = k a
    return a = Writer mempty a
```

Exercise 5.8.

1. Implement a `Monoid` instance for `Int`.
2. Would it make sense to use `Writer Int` to keep track of time spent?
3. Would it make sense to use `Writer Int` to keep track of number of cores used?



5.3.4 The reader monads

Writer monads and reader monads sound analogous, but are they really? The writer monad is parameterized by a choice of monoid—like strings under concatenation—whereas the reader monad is parameterized only by a choice of set or type, like the three-element set $\{A, B, C\}$ or **A** | **B** | **C**. Whereas the writer monad repeatedly appends more and more information into something like a log file; the reader monad never changes the file being read from.

The reason things are so different—at least in **Set**—is a reflection of a certain asymmetry that’s a bit surprising at first. We just reviewed the definition of monoid; let’s put a formal definition in parallel with the dual notion, that of comonoid. Recall that a category \mathcal{C} has finite products iff it has a terminal object 1 and for every pair of objects $c, d \in \mathcal{C}$, there is a product $c \leftarrow c \times d \rightarrow d$. The categories **Set** and **Hask** have finite products.

Definition 5.9. Let \mathcal{C} be a category with finite products.

A *monoid* in \mathcal{C} consists of a tuple (c, η, μ) , where $c \in \mathcal{C}$ is an object, and $\eta: 1 \rightarrow c$ and $\mu: c \times c \rightarrow c$ are functions making the following diagrams commute:

$$\begin{array}{ccccc} c \times 1 & \xrightarrow{c \times \eta} & c \times c & \xleftarrow{\eta \times c} & 1 \times c \\ \cong \uparrow & & \downarrow \mu & & \uparrow \cong \\ c & \xlongequal{\quad} & c & \xlongequal{\quad} & c \end{array}$$

$$\begin{array}{ccc} c \times c \times c & \xrightarrow{c \times \mu} & c \times c \\ \mu \times c \downarrow & & \downarrow \mu \\ c \times c & \xrightarrow{\mu} & c \end{array}$$

A *comonoid* in \mathcal{C} consists of a tuple (c, ϵ, δ) , where $c \in \mathcal{C}$ is an object, and $\epsilon: c \rightarrow 1$ and $\delta: c \rightarrow c \times c$ are functions making the following diagrams commute:

$$\begin{array}{ccccc} c \times 1 & \xleftarrow{c \times \epsilon} & c \times c & \xrightarrow{\epsilon \times c} & 1 \times c \\ \cong \downarrow & & \uparrow \delta & & \downarrow \cong \\ c & \xlongequal{\quad} & c & \xlongequal{\quad} & c \end{array}$$

$$\begin{array}{ccc} c \times c \times c & \xleftarrow{c \times \delta} & c \times c \\ \delta \times c \uparrow & & \uparrow \delta \\ c \times c & \xleftarrow{\delta} & c \end{array}$$

But here’s the surprise: whereas monoids abound—for example a set with three elements has seven monoid structures (see <https://oeis.org/A058129>)—comonoids do not. Each object has exactly one comonoid structure, no matter what!

Proposition 5.10. Let \mathcal{C} be a category with finite products. Then for every object $c \in \mathcal{C}$ there is exactly one comonoid structure possible on c .

Proof. There is only one function $c \rightarrow 1$ by definition of 1 being terminal, so ϵ is fixed; it remains to show that δ is too. We first want to see that the functions $c \times \epsilon$ and $\epsilon \times c$, followed by the isomorphisms $c \times 1 \rightarrow c$ and $1 \times c \rightarrow c$, must be the usual projections $c \times c \rightarrow c$; we leave this to the reader in Exercise 5.11. But now δ must be the diagonal $\delta = (c, c)$ by the universal property of products. \square

Exercise 5.11. Show that if (c, ϵ, δ) is a comonoid, then the functions $c \times \epsilon$ and $\epsilon \times c$, followed by the isomorphisms $c \times 1 \rightarrow c$ and $1 \times c \rightarrow c$, must be the usual projections $c \times c \rightarrow c$. \diamond

The notion of comonoid seems to be pointless! In fact, we will soon discuss monoidal categories, of which categories with finite products are special cases. In general monoidal categories, there can be non-trivial and interesting comonoids. But it's true that for where we are in this chapter, the notion of comonoid seems to disappear.

Yet there was a point in bringing it up. The apparent asymmetry between the writer monad and the reader monad is not that one requires a monoid and the other doesn't require anything, it's that one requires monoids and the other requires comonoids. The symmetry and analogy between writer and reader is in full force; there's just something deeper going on with the math.

In the remainder of this section on the reader monad, one can watch for uses of the functions $\epsilon: c \rightarrow 1$ and $\delta: c \rightarrow c \times c$; while they are easy to hide because they're part of the basic structure of a category with products, they are there behind the scenes. Whenever the elements of c are discarded, we're using ϵ , and whenever the elements of c are duplicated, we're using δ .

Definition 5.12 (Reader monad). Let $C \in \mathbf{Set}$ be a set. There is a functor sending a set X to X^C , the set of functions $C \rightarrow X$; on a morphism $f: X \rightarrow Y$, it sends a function $C \rightarrow X$ to its composite with f to get a function $C \rightarrow Y$. This functor $-^C: \mathbf{Set} \rightarrow \mathbf{Set}$ is part of a monad, called the *reader monad on C*. The unit (return) $\eta_X: X \rightarrow X^C$ sends $x \mapsto (c \mapsto x)$.^a The multiplication (join) $\mu_X: (X^C)^C \rightarrow X^C$ sends a function $f: C \rightarrow X^C$ to the function $c \mapsto f(c)(c)$.^b

^aThe element $c \in C$ is discarded here.

^bThe element $c \in C$ is duplicated here.

One can think of a Kleisli arrow $f: x \twoheadrightarrow y$ for the reader monad on C as being parameterized by a file $c \in C$: what f does depends on what's in c . The return does not use the file, and the composition of Kleisli arrows uses duplication to ensure that we're using the same file c for each arrow.

Let's get to some code.

```
newtype Reader c a = Reader (c -> a) -- c is the input file type
deriving Functor
```



```
runReader :: Reader c a -> c -> a    -- drop the data constructor
runReader (Reader f) c = f c         -- for later convenience
```

It's easy to define the reader functor; it's just a little more involved to give its return and bind. But again we'll see duplication and discarding of the input file.

```
instance Monad (Reader c) where
  ra >=> k = Reader f
    where f c = (runReader k (runReader ra c)) c -- duplicate c
  return a = Reader (\_ -> a)                  -- discard c
```

Exercise 5.13. Give an example of a program that uses the reader monad, written in **do** notation. For full credit, it should be a program that doesn't just work for any monad, but instead only makes sense for the reader monad. \diamond

5.3.5 The state monads

For every type **s** (or set S) there is a state monad, where states are terms of type **s**. For example, suppose every time you run a function, you're allowed to look in your virtual notebook, see what was the last thing you wrote there, and use it in your calculations; you're also allowed to add a new entry to your notebook. The type of each entry in your notebook is **s**.

The way we just described it, a Kleisli morphism $f: A \rightarrowtail B$ would be a function $f = (f_B, f_S): A \times S \rightarrow B \times S$: it takes in an input $a \in A$ and also an element $s \in S$ (the "current state") and returns an output $f_B(a, s) \in B$ as well as a "new state" $f_S(a, s)$.

But Kleisli morphisms $f: A \rightarrowtail B$ are supposed to go from A to $M(B)$ for some monad M . To fit our map $A \times S \rightarrow B \times S$ into that form, we curry the S , to give $A \rightarrow (B \times S)^S$. The S -state monad has $B \mapsto (B \times S)^B$ as its underlying functor. Here it is in Haskell.

```
newtype State s a = State (s -> (a, s))
  deriving Functor

runState :: State s a -> s -> (a, s) -- drop the data constructor
runState (State f) s = f s           -- for later convenience
```

```
instance Monad (State s) where
  sa >=> k = State (\s -> let (a, s') = runState sa s
```

```

                                in runState (k a) s')
return a = State (\s -> (a, s))

```

Exercise 5.14. Give an example of a program that uses the state monad, written in **do** notation. For full credit, it should be a program that doesn't just work for any monad, but instead only makes sense for the state monad. \diamond

5.3.6 The continuation monads

For any type **r** (or set R) we can imagine working within a program where everyone knows the goal is to get an **r** at the end. It's like a maze where you're working backward: you're always thinking about the places from which you can get to the exit **r**. If someone talks about a function from A to B in this context, everyone knows that they really are trying to say "if you can exit from B , this function will let you exit from A ".

The way we described it, a Kleisli morphism $f: A \rightarrowtail B$ would be a function $R^B \rightarrow R^A$: it takes a function $B \rightarrow R$, i.e. way to exit from B , to a function $A \rightarrow R$, which is a way to exit from A .

But Kleisli morphisms $f: A \rightarrowtail B$ are supposed to go from A to $M(B)$ for some monad M . To fit our map $R^B \rightarrow R^A$ into that form, we uncurry to get a map $A \times R^B \rightarrow R$ and then flip and curry to obtain $A \rightarrow R^{(R^B)}$. The R -continuations monad has $B \mapsto R^{(R^B)}$ as its underlying functor.

```

data Cont r a = Cont ((a -> r) -> r)

runCont :: Cont r a -> (a -> r) -> r    -- drop the data constructor
runCont (Cont k) h = k h                 -- for later convenience

```

Exercise 5.15. Show that for any set $R \in \mathbf{Set}$, the map sending $B \mapsto R^{(R^B)}$ is functorial. This can be done in math style, or by writing an implementation of **fmap** for **Cont** **r**. \diamond

```

instance Monad (Cont r) where
  ka >=> kab = Cont (\hb -> runCont ka (\a -> runCont (kab a) hb))
  return a = Cont (\ha -> ha a)

```

Exercise 5.16. Give an example of a program that uses the continuations monad,

written in **do** notation. For full credit, it should be a program that doesn't just work for any monad, but instead only makes sense for the continuations monad. ◇

5.3.7 The **IO** monad

For better or for worse, the most important monad in Haskell is probably **IO**. It's something of a blessing, in that it's the monad that makes Haskell competitive in an ecosystem of other programming languages. In a purely functional language there is not input or output. Consider the function `getLine`, which is supposed to get a line of input from the user. If this were a pure function, it would always return the same string. This is clearly wrong, so we need something to get input and output into the picture. It's a blessing that the **IO** monad let's input and output back into the picture.

But it's also something of a curse, because the **IO** monad is not really a monad! It's implemented deep within the compiler of Haskell, not in the standard way. One can imagine it (though again it's not and could not be implemented this way) as a kind of state monad, where the state in question is the state of the entire rest of the world outside the program. From this point of view, when `getLine` takes input from the user and uses it in the program, it's really reading the entire state of the world and using that. And when `putStrLn` provides output to the user, it's really updating the entire state of the world, a state that doesn't change except for the way it is affected by the program. This fiction is in some sense bizarre, but it is solid enough to have as our model.

One way to think about how Haskell deals with input and output is that it creates a long strategy for what it will output and what it will do with each sort of input. It then postpones input and output until after the program—which is a pure function—is executed. A Haskell program produces a detailed set of instructions for the runtime to execute: to access the input devices, produce output, connect to the internet, write to disk, etc. This set of instructions, or strategy, is encapsulated inside the programs use of **IO**. Every Haskell program contains `main`, which is of type **IO ()**, where `()` is of course unit type. We call it an **IO object** because it has no other data. It is declared as

```
main :: IO ()
```

This **IO** object is produced by your program and is executed by the runtime. Since **IO** is a monad which is baked deep into the language, you can create a value of the type **IO ()** either directly using `return` or by composing smaller **IO**-producing functions. Here's the simplest, do-nothing program

```
main :: IO ()
main = return ()
```

A slightly more interesting program produces **IO ()** by calling a library function `putStrLn` to display a string followed by a newline

```
main :: IO ()
main = putStrLn "Hello World!"
```

The important baked-in definitions like `putStrLn`, `getLine`, `getChar` have the following types, and anything more complicated can be composed, either using `bind`, or the `do` notation from simple `IO`-functions like the following:

```
putStrLn :: String -> IO ()
getLine  :: IO String
getChar  :: IO Char
```

The `IO` monad is peculiar in that it is not implemented from anything simpler that you can take apart. It thus provides no way of accessing its contents.

For instance, to get a character of input, you would use `getChar`, but there is no way to bring this quasi-Char into the open inside your program as a real `Char`. You can manipulate your quasi-Char by applying `fmap` to it; indeed `IO`, like every monad, is also a functor. Or you could bind it to a Kleisli arrow and get another `IO` object. But you can never retrieve a bare `Char` out of it. There is no morphism `IO Char -> Char` that extracts the `Char` as input.

Exercise 5.17. There is no morphism `IO Char -> Char` that extracts the `Char` as input, but there are certainly morphisms of that type. Write one. ◇

The `do` notation may give you the impression that you can access a string `name` in the program we saw on page 132:

```
main :: IO ()
main = do
  putStrLn "Tell me your name"
  name <- getLine
  putStrLn ("Hello " ++ name)
```

but as we saw in ??—when we discussed what’s really going on in a `do` block—that `name` is just a name of the argument to a lambda function that will produce another `IO` object.

Exercise 5.18. Write a Haskell program in `do` notation of type `Int -> IO Int` that takes an integer, and prompts the user for whether it should be displayed and then zeroed out or not. If so, it outputs the integer and then returns 0. If not, it outputs “ok” and returns the integer as is. ◇

Monoidal Categories

6.1 Lax Monoidal Functors

Exercise 6.1. Show that the category **1** equipped with the obvious tensor product is the terminal object in **MonCat** ◇

Exercise 6.2. Show that a lax monoidal functor from the monoidal category **1** to the category of endofunctors $[C, C]$ with functor composition as the tensor product defines a monad. ◇

6.1.1 Monad as Applicative

A functor lets you lift a function

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

It doesn't, however, let you lift a function of two variables. What we would like to see is

```
lift2 :: (a -> b -> c) -> f a -> f b -> f c
```

Granted, a function of two variables, in the curried form, is just a function of a single variable returning a function. If we look at the first argument as

```
a -> (b -> c)
```

we could `fmap` it over the second argument `f a` to get

```
f (b -> c)
```

But we are stuck now, because we don't know how to apply a function inside a functor to an argument inside a functor. For that we need a new capability, for which we will define an infix operator

```
(<*>) :: f (a -> b) -> f a -> f b
```

We can then implement `lift2` as

```
lift2 fab fa fb = fmap fab fa <*> fb
```

We won't need parentheses if we define this operator to be right associative. There is also an infix operator `<$>` that is a synonym for `fmap`, which lets us write `lift2` in an even more compact form

```
lift2 fab fa fb = fab <$> fa <*> fb
```

Notice that this is enough functionality to lift a function of any number of variables, for instance

```
lift3 fabc fa fb fc = fabc <$> fa <*> fb <*> fc
```

With one more function called `pure` we can even lift a function of zero variables, in other words, a value

```
pure :: a -> f a
```

Altogether we get a definition of an applicative functor

```
class Functor f => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
  pure  :: a -> f a
```

Every monad is automatically an applicative functor. They share the same polymorphic function `a -> f a` under two different names, and for every monad we have the implementation of `ap`, which has the same type signature as `<*>`

```
ap :: Monad m => m (a -> b) -> m a -> m b
ap mab ma = do
  f <- mab
  a <- ma
  return (f a)
```

In principle, one could use a monad as an applicative with a slightly more awkward syntax: using `ap` instead of `<*>` and `return` instead of `pure`. It was enough of a nuisance, though, that a decision was made to include **Applicative** as a superclass of **Monad**, which brings the applicative names `<*>` and `pure` into the scope of the monad. This is the official definition of a **Monad** copied from `Control.Monad`

```
class Applicative m => Monad m where
  (>=)      :: forall a b. m a -> (a -> m b) -> m b
  (>>)      :: forall a b. m a -> m b -> m b
  m >> k = m >= \_ -> k
  return    :: a -> m a
  return    = pure
```

The operator `(>>)` is used when the result of the first action is not needed. It is, for instance, used in desugaring the `do` lines that omit the left arrow (see example in the description of the **IO** monad).

In Haskell, an **Applicative** functor is equivalent to a lax monoidal functor. This is the Haskell definition of the latter

```
class Functor f => Monoidal f where
  unit :: f ()
  (>*<) :: f a -> f b -> f (a, b)
```

Indeed, starting with an **Applicative**, we can implement `unit` as

```
unit = pure ()
```

and `>*<` as

```
fa >*< fb = fmap (,) fa <*> fb
```

Conversely, given a **Monoidal** functor we can implement

```
ff <*> fa = fmap (uncurry ($)) (ff >*< fa)
```

where the *apply* operator `($)` is defined as

```
($) :: (a -> b) -> a -> b
f $ a = f a
```

and

```
pure a = fmap (\() -> a) unit
```

6.2 Strength and enrichment

We include this section here, even though it requires some things we'll get to later. Do not be worried if you don't understand everything for now.

In Example 3.47 we explained that Haskell functors are somehow *less* than true functors because they need not preserve identity and composition. But in another sense they are *more* than mere functors: they are strong and enriched. While we cannot explain that fully at the moment, we can give the flavor.

In a category like **Set**, objects are sets but also for every two objects there is again a *set* of morphisms: the collection of morphisms between two objects is again an object. But for an arbitrary category \mathcal{C} , the collection of morphisms between objects c, d is just a set, not an object of \mathcal{C} . A category of the first kind is called *closed*: the collection of morphisms from c to d is again an object, which we can denote d^c . It is sometimes called the *function type* or *exponential object*.

So **Set** is closed; in fact it is *Cartesian closed* which means that it also has finite products and there is a relationship:

$$\mathbf{Set}(b, d^c) \cong \mathbf{Set}(b \times c, d)$$

This is called the Currying adjunction and we'll get to it soon. **Hask** is also Cartesian closed: it has products and function types satisfies the same equation. That is, there is an isomorphism between Haskell functions of the type $b \rightarrow (c \rightarrow d)$ and those of the type $(b, c) \rightarrow d$.

Functors $F: \mathcal{C} \rightarrow \mathcal{D}$ between arbitrary categories need to send morphisms to morphisms, i.e. elements of the *set* $\mathcal{C}(c_1, c_2)$ to elements of the *set* $\mathcal{D}(Fc_1, Fc_2)$. That is, we get a function $F_{c_1, c_2}: \mathcal{C}(c_1, c_2) \rightarrow \mathcal{D}(Fc_1, Fc_2)$. But if $\mathcal{D} = \mathcal{C}$ and it is Cartesian closed, then we might ask $F_{c, c'}$ to be a *morphism in \mathcal{C} itself*! That is, we want it to be a morphism $F_{c_1, c_2}: c_2^{c_1} \rightarrow (Fc_2)^{Fc_1}$.

This is implicitly what is going on when we write `fmap :: (a -> b) -> (f a -> f b)`. The function `fmap` is not just a set-theoretic way of sending each term of type $a \rightarrow b$ to a term of type $f a \rightarrow f b$, but a function written in Haskell, a morphism in the category **Hask**.

In a cartesian closed category \mathcal{C} , enrichment and strength are the same thing. What is strength? A functor $F: \mathcal{C} \rightarrow \mathcal{C}$ is called *strong* if, for every c, c' there is a natural morphism $c \times F(c') \rightarrow F(c \times c')$, called the *strength*. By uncurrying, the strength is equivalent to a map $c \rightarrow F(c \times c')^{F(c')}$, and so if F is enriched then we just need a function $c \rightarrow (c' \times c)^{c'}$, and we get that by currying the identity. Conversely, given a strength we need a function $c_2^{c_1} \rightarrow (Fc_2)^{Fc_1}$. This is equivalent to a function $c_2^{c_1} \times Fc_1 \rightarrow Fc_2$; to obtain it we just use the composite $c_2^{c_1} \times Fc_1 \rightarrow F(c_2^{c_1} \times c_1) \rightarrow F(c_2)$, where the first map is the strength and the second is application of F on the evaluation morphism $c_2^{c_1} \times c_1 \rightarrow c_2$.

Again, none of this is too important for now. It's "enrichment" for those who want it. But we thought it was worth discussing because one may hear that all functors in

Haskell are strong, and it is good to know that this condition is equivalent to asking that they are enriched, i.e. that there is a natural map of type `fmap :: (a -> b) -> (f a -> f b)`.

Profunctors

7.1 Profunctors revisited

In the early days of category theory, people talked of two different kinds of functors $\mathcal{C} \rightarrow \mathcal{D}$: covariant and contravariant. Both kinds send objects and morphisms of \mathcal{C} to objects and morphisms of \mathcal{D} , but contravariant ones flip the direction of every morphism. But this way of thinking was not very useful in practice. A contravariant functor from \mathcal{C} to \mathcal{D} is just a covariant functor $\mathcal{C}^{\text{op}} \rightarrow \mathcal{D}$ which is just the same as a covariant functor $\mathcal{C} \rightarrow \mathcal{D}^{\text{op}}$. It's more convenient if every time we draw an arrow between categories, it always indicates the same type of thing, namely a (covariant) functor. And who's to say whether \mathcal{C} or \mathcal{C}^{op} is the “preferred one”, i.e. whether a functor $\mathcal{C}^{\text{op}} \rightarrow \mathcal{D}$ should be thought of as contravariant (out of \mathcal{C}) or as covariant (out of \mathcal{C}^{op}).

This works for functors with multiple arguments too. If you have a functor that takes two objects or morphisms from category \mathcal{C} and returns an object or morphism from category \mathcal{D} , you might want to call it a bifunctor. But technically, we just make think of it as a regular old functor $\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{D}$ whose domain happens to be a product category. See the arrow between categories? It always means regular old functor.

However, while the above is more suited for mathematics, we run into a familiar theme when it comes to Haskell. The fact that Haskell never looks beyond itself—everything's about **Hask**, every “functor” goes from **Hask** to **Hask**—means that we very well know whether **Hask** or **Hask**^{op} is the preferred one! If we see a functor out of **Hask**^{op}, that's contravariant, silly! And a functor **Hask** \times **Hask** \rightarrow **Hask** is a bifunctor, you silly Copernicus person!

Well so be it. This course is mainly about Haskell, but it's suppose to set it in a broader context, so we need to be comfortable with both ways of thinking. In fact, if you are going to spend most of your time in Haskell land, it really does make sense to think of certain functors as covariant and others as contravariant.

Sometimes in category theory, the most interesting sorts of things are the most mundane, the overlooked obvious. And one such case is that of the *Hom functor*.

Exercise 7.1. A classmate comes up and says “did you know know that for every category \mathcal{C} , the homs form a functor to **Set**?” She goes on to explain that it’s a functor $\text{hom}: \mathcal{C}^{\text{op}} \times \mathcal{C} \rightarrow \mathbf{Set}$.

1. What are the objects of $\mathcal{C}^{\text{op}} \times \mathcal{C}$?
2. Given an object of $\mathcal{C}^{\text{op}} \times \mathcal{C}$, she’s saying that you can take hom of it and get a set. What set is it?
3. Given two objects in $\mathcal{C}^{\text{op}} \times \mathcal{C}$, what is a morphism between them?
4. Given such a morphism in $\mathcal{C}^{\text{op}} \times \mathcal{C}$, your classmate is saying that there’s a function from hom of the first to hom of the second. What is it?
5. Does hom as you’ve now given it on objects and morphisms respect identities and composition?

◇

The hom-functor is a very special case of something called a *profunctor*. Profunctors generalize important notions you may hear bandied about in category theory communities, terms like *presheaf* and *copresheaf*. But what are they?

Definition 7.2. Let \mathcal{C} and \mathcal{D} be categories. A *presheaf* on \mathcal{C} is a functor $\mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}$. A *copresheaf* on \mathcal{D} is a functor $\mathcal{D} \rightarrow \mathbf{Set}$. And a *profunctor* P from \mathcal{C} to \mathcal{D} , denoted $P: \mathcal{C} \nrightarrow \mathcal{D}$, is just a functor

$$P: \mathcal{C}^{\text{op}} \times \mathcal{D} \rightarrow \mathbf{Set}.$$

It’s kinda funny that the name “copresheaf” sounds more complicated, but that the concept is a little easier, having no $^{\text{op}}$ involved. We’ll see *some* justification for that later when we talk about the Yoneda embedding, but in some sense it’s really just the result of a historical accident. Namely in the geometric contexts where these notions first came up, people tended to find presheaves more commonly than copresheaves.

Example 7.3. Let \mathcal{C} be any category and $c \in \mathcal{C}$ an object. Then $F := \mathcal{C}(c, -)$ is a copresheaf: it assigns a set $F(c_1)$ to every object $c_1 \in \mathcal{C}$ and a function $F(g): F(c_1) \rightarrow F(c_2)$ to every morphism $g: c_1 \rightarrow c_2$. Namely, $F(c_1)$ is the set of maps $\mathcal{C}(c, c_1)$; it’s a set, right? And given g , we assign $F(g): \mathcal{C}(c, c_1) \rightarrow \mathcal{C}(c, c_2)$ to be the function that sends $f: c \rightarrow c_1$ to $g \circ f$.

The copresheaf $\mathcal{C}(c, -)$ is said to be *represented* by c . Similarly, the presheaf $\mathcal{C}^{\text{op}} \rightarrow \mathbf{Set}$ represented by c is $\mathcal{C}(-, c)$.

Example 7.4. Let \mathcal{E} be the category whose objects are street-intersections in a certain city. A morphism $c \rightarrow c'$ is a route that a car is allowed to take from c to c' . We could take $F: \mathcal{E} \rightarrow \mathbf{Set}$ to assign $F(e)$ to be the set of ways that a person can get from the central train station to e . This includes all routes, not just car. Now given a morphism $g: e \rightarrow e'$, i.e. a car route, we get a function $F(e) \rightarrow F(e')$: if one has a way to get from

the central station to e , then by taking a car along the final leg to their journey, they have a way to get to e' .

Example 7.5. Let $\mathcal{C} = [\bullet \rightarrow \bullet]$ be the walking arrow category. For any sets A, B and function $f: A \rightarrow B$, we can think of f as a copresheaf $\mathcal{C} \rightarrow \mathbf{Set}$.

Anyway, why do we say that profunctors generalize presheaves and copresheaves? It mainly comes down to the fact that if $\mathbf{1}$ denotes the one-object, one morphism (just identity) category, then for any category \mathcal{C} we have

$$\mathbf{1} \times \mathcal{C} \cong \mathcal{C} \cong \mathcal{C} \times \mathbf{1}. \quad (7.6)$$

Exercise 7.7. Use the isomorphisms in (7.6) to justify the statements:

1. a presheaf on \mathcal{C} is just a profunctor from ____ to ____
2. a copresheaf on \mathcal{D} is just a profunctor from ____ to ____.

◇

Let's give a glossary of the above terms in the form of Haskell code; much of this was already covered in Chapter 2, but we recall it for your convenience.

Remember that to implement a functor $\mathbf{Hask} \rightarrow \mathbf{Hask}$, one gives a type constructor and then says how it is an instance of the type class **Functor** by defining `fmap`:

```
class Functor f where                -- in Haskell, functors are covariant
  fmap :: (a -> b) -> (f a -> f b)
```

If you want to define a contravariant functor $\mathbf{Hask}^{\text{op}} \rightarrow \mathbf{Hask}$, it's just a different type class all together

```
class Contravariant f where
  fmap :: (b -> a) -> (f a -> f b)
```

In Haskell, most contravariant functors are constructed from function types since hom-functors are contravariant in the first argument.

Exercise 7.8. Give an example of a contravariant functor in Haskell.

1. Give the type constructor.
2. Implement `fmap`.

◇

Moving on, if you want to implement a bifunctor, i.e. a two-argument covariant functor, you give the type constructor and you implement something called `bimap`:

```
class Bifunctor f where
  bimap :: (a -> a') -> (b -> b') -> (f a b -> f a' b')
```

Important examples of bifunctors are products and coproducts; see Section 3.4.3.

Finally, if you want to implement a profunctor in Haskell, i.e. a mixed variance functor $\mathbf{Hask}^{\text{op}} \times \mathbf{Hask} \rightarrow \mathbf{Hask}$, you give the type constructor and you implement something called `dimap`.

```
class Profunctor p where
  dimap :: (a' -> a) -> (b -> b') -> (p a b -> p a' b')
```

As mentioned above, profunctors can be seen as a generalization of the hom-functors your classmate told you about in Exercise 7.1. In [SevenSketches], a profunctor was described as a bridge between two categories, as though adding in morphism-like-things between objects in two different worlds. In the case of the hom-profunctor for \mathcal{C} , the objects live in the same world (\mathcal{C}), but that's a special case. But the reason we say it's as though a profunctor $P: \mathcal{C}^{\text{op}} \times \mathcal{D} \rightarrow \mathbf{Set}$ provides morphism-like-things between objects in \mathcal{C} and objects in \mathcal{D} is that if $x \in P(c, d)$ is a morphism-like-thing from c to d , and $f: c' \rightarrow c$ and $g: d \rightarrow d'$ are real morphisms in \mathcal{C} and \mathcal{D} , then we can do a compose-like-operation to get a morphism-like-thing $P(f, g)(x)$ from c' to d' . When you get used to it, this really feels like composition, and in the special case that P is the hom-functor, it is.

An alternative viewpoint is that profunctors generalize relations between sets. We can think of a relation, like “greater than $>$ ”, as a predicate on a pair of objects: if elements are related in this way, the predicate returns true; if they're not, it returns false. Instead of thinking in terms of true/false or black/white, we are sometimes interested in witnesses or proofs that the relationship holds; this might be called the *Curry-Howard* perspective. Instead of assigning a Boolean to a pair of objects, we can assign a whole set of witnesses. If the set of witnesses is empty, the two objects are deemed unrelated. But in general, we may want to keep track of all the witnesses to a relationship. Now if we wanted to say “if Alice witnesses that c is related to d , then for any map $c' \rightarrow c$, we want Alice to bear witness to the fact that c' is related to d ”. This idea is nicely expressed by a **Set**-valued profunctor.

Exercise 7.9. Suppose that X and Y are sets; we can consider them as discrete categories \mathcal{X} and \mathcal{Y} (Example 1.35). Is it true that there is a one-to-one correspondence between relations $R \subseteq X \times Y$ and profunctors $P: \mathcal{X} \rightarrow \mathcal{Y}$? Explain. \diamond

Profunctors provide profound proficiency to the programming professional. Indeed, many Haskellers have heard of lenses. In the coming sections, we'll explain this relatively new and very valuable programming paradigm.

7.2 Ends and Coends

Definition 7.10 (Wedge). Given a profunctor $P: \mathcal{C} \nrightarrow \mathcal{C}$, i.e. a functor $P: \mathcal{C}^{\text{op}} \times \mathcal{C} \rightarrow \mathbf{Set}$, a *wedge* over P consists of a set x , together with a function

$$\alpha_c: x \rightarrow P(c, c)$$

for every $c \in \mathcal{C}$, such that all these functions together satisfy the *wedge condition*: for every morphism $f: c \rightarrow d$ in \mathcal{C} , the following diagram commutes:

$$\begin{array}{ccc} & x & \\ \alpha_c \swarrow & & \searrow \alpha_d \\ P(c, c) & & P(d, d) \\ P(\text{id}_a, f) \searrow & & \swarrow P(f, \text{id}_b) \\ & P(c, d) & \end{array} \quad (7.11)$$

We denote this wedge by (x, α) , where α denotes the whole family of functions.

Definition 7.12 (End). Given a profunctor $P: \mathcal{C} \nrightarrow \mathcal{C}$, an *end* is a universal wedge over P , i.e. a wedge (end, π) such that, for every other wedge (x, α) over P there is a unique function $h: x \rightarrow \text{end}$ such that for every $c \in \mathcal{C}$ the following diagram commutes

$$\begin{array}{ccc} & x & \\ & \downarrow h & \\ & \text{end} & \\ \alpha_c \swarrow & & \swarrow \pi_c \\ & P(c, c) & \end{array} \quad (7.13)$$

We usually use following integral notation for ends

$$\text{end}(P) = \int_{c \in \mathcal{C}} P(c, c),$$

with the “integration variable” c at the bottom of the integral sign.

Example 7.14 (Natural transformations as end). An important example of an end is the set of natural transformations between two functors $F, G: \mathcal{C} \rightarrow \mathcal{D}$:

$$\text{Nat}(F, G) \cong \int_{c \in \mathcal{C}} \mathcal{D}(Fc, Gc).$$

What’s going on here? First of all, we need to see that there’s a wedge, and second we need to see that it’s universal.

But before we can do that, we need to say what the profunctor is. Just following our

nose from (7.13), we see $\mathcal{D}(Fc, Gc)$ in the place of $P(c, c)$, so we surmise that $\mathcal{D}(F-, G-)$ is a profunctor. You can check that it is in Exercise 7.15.

So let $\text{Nat}(F, G)$ be the set of natural transformations. Is it a wedge over $\mathcal{D}(F-, G-)$? To say so, we need to give a function $\text{comp}_c : \text{Nat}(F, G) \rightarrow \mathcal{D}(Fc, Gc)$ for every $c \in \mathcal{C}$. Let's say $q : F \rightarrow G$ is a natural transformation; that means that for every $c \in \mathcal{C}$ we have a component $q_c : F(c) \rightarrow G(c)$, and that these are natural in the appropriate sense. So what should we make $\text{comp}_c(q) : \mathcal{D}(Fc, Gc)$ be? Let's make it be $\text{comp}_c(q) := q_c$. You can check that this definition satisfies the required commutativity of (7.11) in Exercise 7.16.

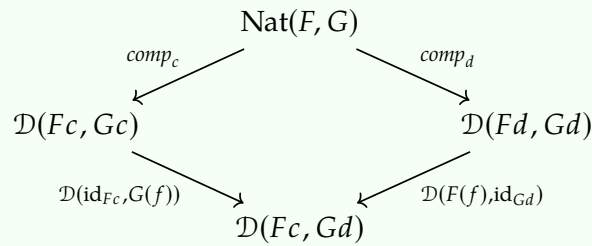
Now we need to see that this wedge is actually universal. In some sense, this is true just because the definition of wedge in this case is a rephrasing of the definition of natural transformation. To actually prove it, one takes an arbitrary wedge (x, α) , constructs a function $h : x \rightarrow \text{Nat}(F, G)$, shows that h satisfies the property (7.13), and finally shows that h is unique with respect to this. We leave this to you in Exercise 7.17.

Exercise 7.15. Let \mathcal{C}, \mathcal{D} be categories and $F, G : \mathcal{C} \rightarrow \mathcal{D}$ be functors. Check that $P := \mathcal{D}(F-, G-)$ is a profunctor, by answering the following.

1. What does $P : \mathcal{C}^{\text{op}} \times \mathcal{C} \rightarrow \mathbf{Set}$ do on objects?
2. What does P do on morphisms?
3. Does P preserve identities and compositions?

◇

Exercise 7.16. For any natural transformation $q : F \rightarrow G$, let $\text{comp}_c(q) := q_c$ as in Example 7.14. Show that for any $f : c \rightarrow d$ in \mathcal{C} , the following diagram commutes:



◇

Exercise 7.17. Let $F, G : \mathcal{C} \rightarrow \mathcal{D}$ be functors and let (x, α) be a wedge for the profunctor $\mathcal{D}(F-, G-)$.

1. Construct a function $h : x \rightarrow \text{Nat}(F, G)$.

2. Show that for every $c \in \mathcal{C}$, the following diagram commutes:

$$\begin{array}{ccc}
 & x & \\
 & \downarrow h & \\
 \alpha_c \swarrow & \text{Nat}(F, G) & \searrow \text{comp}_c \\
 & \mathcal{D}(Fc, Gc) &
 \end{array}$$

3. Show that h is unique with respect to that property. ◇

The mapping in property of an end can be expressed as

$$\text{Set}(x, \int_c P(c, c)) \cong \int_c \text{Set}(x, P(c, c))$$

This is a very useful property that is used in many derivations.

The end is defined by its mapping in property and, in many ways, it is similar to a product. It also has a dual, a coend, which behaves more like a coproduct. We define a co-wedge

Definition 7.18 (Co-wedge). Given a profunctor $P: \mathcal{C}^{\text{op}} \times \mathcal{C} \rightarrow \mathbf{Set}$, a co-wedge is a set x equipped with a family of functions parameterized by a , an object of \mathcal{C}

$$\alpha_a: P(a, a) \rightarrow x$$

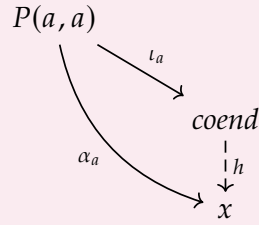
satisfying the co-wedge condition, which states that, for every pair of objects a and b in \mathcal{C} and a morphism $g: b \rightarrow a$, the following diagram commutes

$$\begin{array}{ccccc}
 & P(a, b) & & & \\
 P(id_a, g) \swarrow & & \searrow P(g, id_a) & & \\
 P(a, a) & & & & P(b, b) \\
 \alpha_a \searrow & & \swarrow \alpha_b & & \\
 & x & & &
 \end{array}$$

A coend satisfies a mapping-out universal condition

Definition 7.19 (Coend). Given a profunctor $P: \mathcal{C}^{\text{op}} \times \mathcal{C} \rightarrow \mathbf{Set}$, a coend is a universal co-wedge, that is a co-wedge (coend, ι) such that, for every other co-wedge (x, α) there

is a unique function $h: coend \rightarrow x$ such that the following diagram commutes



A coend is also written as an integral, but this time with the object at the top of the integral sign

$$coend = \int^{a \in C} P(a, a)$$

The mapping out property if the coend is expressed as

$$Set\left(\int^c P(c, c), x\right) \cong \int_c Set(P(c, c), x)$$

Notice that the coend on the left hand side becomes the end on the right hand side.

7.3 Profunctors in Haskell

```
class Profunctor p where
  dimap :: (a' -> a) -> (b -> b') -> (p a b -> p a' b')
```

The simplest example of a profunctor is the hom-functor which, in Haskell, is represented by the arrow type constructor `->`. This is how it lifts a pair of functions (one of them going in the opposite direction)

```
instance Profunctor (->) where
  dimap f g h = g . h . f
```

An end is represented by a data structure whose data constructor `End` takes a polymorphic object that can be seen as an infinite tuple (product) of all the diagonal elements of a given profunctor. Here, we write it as a GADT, in anticipation of the dual definition of a coend

```
data End p where
  End :: (forall a. p a a) -> End p
```

The following language pragmas are needed for this definition to work

```
{-# language RankNTypes #-}
{-# language GADTs #-}
```

A projection extracts one particular diagonal element of the profunctor from an end

```
piEnd :: End p -> p a a
piEnd (End paa) = paa
```

The universal property of the end can be summarized by two sides of an isomorphism. Given a type `x` and a family of projections `alpha` there is a unique function from `x` to `End p`

```
end :: Profunctor p => (forall a. x -> p a a) -> (x -> End p)
end alpha x = End (alpha x)
```

Conversely, given a mapping from some `x` to `End p`, we can generate all the projections from `x` by precomposing it with the projections from `End p`

```
unend :: Profunctor p => (x -> End p) -> (forall a. x -> p a a)
unend h = piEnd . h
```

The set of natural transformations between two functors can be represented by an end of the following profunctor

```
data NatP f g a b = NatP (f a -> g b)
```

```
instance (Functor f, Functor g) => Profunctor (NatP f g) where
  dimap f g (NatP h) = NatP (fmap g . h . fmap f)
```

```
type Nat f g = End (NatP f g)
```

Generalized Algebraic Data Types or GADTs can be used to encode existential types. In pseudo Haskell we would write the definition of a coend as

```
type Coend p = exists a. p a a
```

Using GADTs we can write it as

```
data Coend p where
  Coend :: p a a -> Coend p
```

The idea is that any type that occurs in the definition of the data constructor but is absent from the definition of the type constructor is automatically treated as an existential. Here, `a` is such a type.

Just like the end had projections, a coend has injections

```
inCoend :: Profunctor p => p a a -> Coend p
inCoend paa = Coend paa
```

You can generate a term of the type **Coend** *p* by providing a single diagonal element of the profunctor. This is analogous to being able to generate a term of the **Either** type by injecting just one of the components of the coproduct.

The universal mapping-out property is described by the pair of functions that form an isomorphism. The first one says that, given a type *x* and a polymorphic family of functions from all the diagonal elements of the profunctor *p*, there is a unique function from **Coend** *p* to *x*

```
coend :: Profunctor p => (forall a. p a a -> x) -> (Coend p -> x)
coend f (Coend paa) = f paa
```

The other says that, given a function from **Coend** *p* to *x*, we can obtain a whole polymorphic family of injections by postcomposing it with the *coend* injections

```
uncoend :: Profunctor p => (Coend p -> x) -> (forall a. p a a -> x)
uncoend h = h . inCoend
```

In all this discussion we never mentioned the (co-)wedge conditions. Should we assume that those have to be checked independently on a case-by-case basis, as it was with, say, monad laws? It turns out that (co-)wedge conditions are automatically satisfied due to parametricity.

7.4 Category of profunctors

Profunctor composition

$$(P \circ Q)(a, b) = \int^c P(a, c) \times Q(c, b)$$

Analogous to composition of relations. There exists a *c* that is related to *a* through *P* and to *b* through *Q*. The proof of this relation is a pair of proofs, hence cartesian product of sets.

Yoneda lemma, given a functor $F: C \rightarrow \mathbf{Set}$

$$\mathbf{Nat}(C(x, -), F) \cong Fx$$

Expressing the set of natural transformations as an end

$$\int_c \mathbf{Set}(C(x, c), Fc) \cong Fx$$

Ninja co-Yoneda

$$\int^c C(c, x) \times Fc \cong Fx$$

Proof using Yoneda embedding

$$\text{Set}\left(\int^c C(c, x) \times Fc, s\right) \cong \int_c \text{Set}(C(c, x) \times Fc, s)$$

We can now use the product/exponential adjunction or currying to get

$$\int_c \text{Set}(C(c, x), s^{Fc})$$

We can perform the “integration” over c using the contravariant version of the Yoneda lemma to get s^{Fx} . Finally, we know that in **Set** the exponential is isomorphic to the hom-set $\text{Set}(Fx, s)$. We get

$$\text{Set}\left(\int^c C(c, x) \times Fc, s\right) \cong \text{Set}(Fx, s)$$

Yoneda lemma then tells us that the representing objects must be isomorphic

$$\int^c C(c, x) \times Fc \cong Fx$$

With this result in hand, we can evaluate profunctor composition in which one of the profunctors is the hom-functor

$$(p \circ C(-, -))ab = \int^c pac \times C(c, b) \cong pab$$

The hom-functor is therefore the right identity of profunctor composition. Similarly, using the contravariant version of the co-Yoneda identity, we can prove that it is also a left identity.

As a sanity check, let’s see what we get when we use profunctor composition to compose two hom-functors. Indeed, we get the formula that’s compatible with morphism composition

$$\int^c C(a, c) \times C(c, b) \cong C(a, b)$$

For every pair of composable morphisms, there is a unique composite morphism.

Notice that left and right identities for profunctor composition are not equalities but (natural) isomorphisms. There is also an isomorphism that asserts the associativity of the profunctor composition. So we almost have a category, except that the laws are satisfied up to isomorphism. Such a category is called a *bicategory*.

7.4.1 Category of profunctors in Haskell

```
data ProCompose p q a b where
  ProC :: p a c -> q c b -> ProCompose p q a b
```

```
instance (Profunctor p, Profunctor q) => Profunctor (ProCompose p q)
  where
    dimap f g (ProC pac qcb) = ProC (dimap f id pac) (dimap id g qcb)
```

Ninja Yoneda

```
data YoP f x a b = YoP ((x -> a) -> f b)
```

```
instance Functor f => Profunctor (YoP f x) where
  dimap g h (YoP k) = YoP (\xa' -> fmap h (k (g . xa')))
```

Alternative encoding of $(f\ x)$

```
type Yo f x = End (YoP f x)
```

If we expand the components of this isomorphism, we get the equivalence of two types

```
forall a. ((x -> a) -> f a) ~ f x
```

In other words, the data type $f\ x$ can be encoded as a higher order polymorphic function. This is easy to see if you consider that you can call this function with an identity function as an argument (choosing a equal to x). The function then returns a value of the type $f\ x$

```
f2yo :: forall a. ((x -> a) -> f a) -> f x
f2yo f = f id
```

Continuation passing style as Yoneda for the identity functor

```
forall a. ((x -> a) -> a) ~ a
```

Very common in web programming and in building compilers.

Conversely, if we are given a value of the type $f\ x$, we can construct a polymorphic function

```
yo2f :: f x -> forall a. ((x -> a) -> f a)
yo2f fx = \h -> fmap h fx
```

You can easily convince yourself that these two functions are the inverse of each other.
Ninja co-Yoneda

```
data CoYoP f x a b = CoYoP (a -> x) (f b)
```

```
instance Functor f => Profunctor (CoYoP f x) where
  dimap g h (CoYoP k fb) = CoYoP (k . g) (fmap h fb)
```

Alternative encoding of (f x)

```
type CoYo f x = Coend (CoYoP f x)
```

Expanding it, we get the equivalence of types (pseudo Haskell)

```
exists a. (a -> x, f a) ~ f x
```

The only thing you can do with the left hand side is to `fmap` the function over `f a` and get a value of the type `f x`. Conversely, given an `f x` value, you can create a pair

```
(id, f x)
```

7.5 Day convolution

Given two functors $F, G: C \rightarrow Set$, their Day convolution is defined as

$$(F \star G)x = \int^{(a,b)} Fa \times Gb \times C(a \otimes b, x)$$

Unit with respect to Day convolution

$$Jx = C(I, x)$$

where I is the unit with respect to the tensor product.

Let's plug it into the definition of Day convolution

$$(F \star J)x = \int^{(a,b)} Fa \times C(I, b) \times C(a \otimes b, x)$$

A coend over a pair of objects is equivalent to a double coend over individual objects¹, one of which can be immediately performed to produce

$$\int^a Fa \times C(a \otimes I, x)$$

We can now use the unit law and perform the integration over a to get

$$(F \star J)x \cong Fx$$

The proof of left identity is analogous.

Exercise 7.20. Show that Day convolution is associative up to isomorphism. \diamond

It follows that Day convolution defines a monoidal structure in the category of functors from a monoidal category \mathbf{C} to \mathbf{Set} . Any time we have a monoidal structure it's natural to ask about monoids with respect to that structure. Here, a monoid would be a functor $F: \mathbf{C} \rightarrow \mathbf{Set}$ equipped with two natural transformations

$$\mu: F \star F \rightarrow F$$

$$\eta: J \rightarrow F$$

satisfying the monoid laws. Let's expand the first definition. The component of the natural transformation μ is a member of the set of natural transformations, which can be expressed as an end

$$\int_x \text{Set}((F \star F)x, Fx) = \int_x \text{Set}\left(\int^{(a,b)} Fa \times Fb \times C(a \otimes b, x), Fx\right)$$

Co-continuity of the hom-set gives us

$$\int_x \int_{(a,b)} \text{Set}(Fa \times Fb \times C(a \otimes b, x), Fx)$$

Using the Yoneda lemma, we can integrate over x to get

$$\int_{(a,b)} \text{Set}(Fa \times Fb, F(a \otimes b))$$

To every natural transformation μ there corresponds a natural transformation

$$Fa \times Fb \rightarrow F(a \otimes b)$$

Similarly, expanding the definition of η we get

$$\int_x \text{Set}(C(I, x), Fx) \cong FI$$

Taken together with monoid laws, we get the definition of a lax monoidal functor.

Proposition 7.21. Given a monoidal category \mathbf{C} , a monoid in the monoidal category of functors from \mathbf{C} to \mathbf{Set} with Day convolution is a lax monoidal functor.

¹This is called the Fubini theorem for coends

7.5.1 Day convolution in Haskell

The formula for Day convolution can be directly translated to Haskell using an existential data type (here, encoded as a GADT)

```
data Day f g x where
  Day :: f a -> g b -> ((a, b) -> x) -> Day f g x
```

7.6 Optics

7.6.1 Motivation

The need for optics arose from practical considerations, first in database applications, then in functional programming. In both domains, we have to deal with deeply nested data structures that cannot be modified in place (in database world, any modification has to be transacted). Read-only access to deeply nested data structures is not a problem, but modifications require some more thought. In principle, if you want to mutate even the smallest part of a large data structure, you should make a fresh copy of it, and incorporate the change as part of its construction. This is not only ineffective, but also requires a lot of bookkeeping.

The performance of this type of modification can be drastically improved by using *persistent data structures*, which are specifically designed for functional programming. Since pure functions never modify the data after it's been constructed, large portions of data structures can be transparently shared rather than copied. Garbage collection then takes care of deallocating unused portions.

In imperative programming, you can traverse a data structure keeping track of current position in a single pointer, and then perform a mutation in place using that pointer. In functional programming, you have to keep track not only of the location, but also how you got there, in order to be able to reassemble the mutated data structure. These generalized pointers are called optics, since they focus on a particular part of a data structure—lenses being the most basic ones.

In category theory, optics provide a way of peering inside objects. Normally, we treat objects as indivisible primitives, but in a monoidal category we have the option of composing objects using the tensor product. If objects can be composed, then it makes sense to ask if they can be decomposed back into their constituents. We know we can do this with cartesian products by using projections. We can also do it with coproducts using pattern matching. But, in general, a tensor product has no special mapping out property. Granted, a mapping out of any object provides us with some information about its structure, but how do we distinguish between mappings that focus on a subobject and the ones that don't? The real test is in being able to recompose the object from its constituent parts. This is why all optics come with pairs of mappings: one extracting the current focus, and one putting the new focus back.

7.6.2 Tensorial optics

The general idea of optics is that you can decompose an object (data structure) into the *focus* and the *residue*. You're not interested in the residue, other than it can be recombined with the (possibly different) focus. This idea is described categorically as the following construction—an optic parameterized by four objects

$$Ostab = \int^c C(s, c \otimes a) \times C(c \otimes b, t)$$

We can read it as: there exists a residue c such that there is a morphism that decomposes the source s into a tensor product of the focus a and this residue. The other morphism takes a modified focus b and recomposes it with the residue to produce the target object t .

The way a and c are put together is purposefully kept vague, we just assume that there is a tensor product in some monoidal category \mathbf{C} . We can then specialize this definition to the case of the cartesian product and coproduct and try to simplify the result.

7.6.3 Lens

For instance, substituting a product for the tensor product, we get

$$Lstab = \int^c C(s, c \times a) \times C(c \times b, t)$$

This can be simplified using the mapping in property of the product

$$\int^c C(s, c) \times C(s, a) \times C(c \times b, t)$$

We can now use the co-Yoneda lemma applied to the following functor

$$Fc = C(s, a) \times C(c \times b, t)$$

This results in replacing c with s in the rest of the expression

$$C(s, a) \times C(s \times b, t)$$

An element of this set is a pair of morphisms

$$\begin{aligned} get &: s \rightarrow a \\ set &: s \times b \rightarrow t \end{aligned}$$

The interpretation is that *get* lets you extract the focus from the source and *set* replaces the focal part of s with a new object b to produce the target t . This pair of morphisms is known as a lens.

7.6.4 Lens in Haskell

Here is a lens encoded in Haskell

```
data Lens s t a b = Lens { get :: s -> a
                          , set :: (s, b) -> t }
```

As an example, lets implement a lens that focuses on the right side of a pair.

```
pLens :: Lens (c, a) (c, b) a b
```

```
pLens = Lens get set
  where
    get :: (c, a) -> a
    get (_, a) = a
    set :: ((c, a), b) -> (c, b)
    set ((c, _), b) = (c, b)
```

7.6.5 Prism

Substituting a coproduct for the tensor product in the definition of the optic leads to the following derivation

$$Pstab = \int^c C(s, c + a) \times C(c + b, t)$$

The coproduct has the mapping out property, so we can replace this with

$$\int^c C(s, c + a) \times C(c, t) \times C(b, t)$$

Rearranging the terms and using the co-Yoneda lemma to eliminate $C(c, t)$, we end up with

$$C(s, t + a) \times C(b, t)$$

A pair of morphisms

$$\begin{aligned} match &: s \rightarrow t + a \\ build &: b \rightarrow t \end{aligned}$$

is called a *prism*. The idea is that *match* tries to extract the focus a but it may fail, in which case it returns a t . *build* creates the target t by injecting b into it.

7.6.6 Prism in Haskell

In Haskell, we would encode it as

```
data Prism s t a b = Prism { match :: s -> Either t a
                             , build :: b -> Either t b }
```

For instance, consider the simple case where s is a sum type `Either c a` and t is `Either c b`.

```
ePrism :: Prism (Either c a) (Either c b) a b
```

If you're given a term `Right a`, `match` should return `a`. Given a term `Left c`, it should recode it as the term of the `Left c` of the type `Either c b`

```
ePrism = Prism match build
  where
    match :: Either c a -> Either (Either c b) a
    match (Right a) = Right a
    match (Left c)  = Left (Left c)
    build :: b -> Either c b
    build = Right
```

7.7 Profunctor optics

When dealing with nested data structures, we have to be able to compose optics. The focus of one lens may be decomposable using another lens. The composition of two *gets* is easy, it's just function composition. But the composition of two *sets* requires some fiddling around.

Exercise 7.22. Create a lens $Lstab$ as a pair of morphisms *get* and *set* that is a composition of $Lstuv$ and $Luvab$, also given in terms of *get* and *set*. \diamond

With this composition, lenses (and, similarly, prisms) can be considered arrows in a category whose objects are pairs of objects in \mathbf{C} . There is, however, a better representation of optics in which composition is very simple—it's just the composition of morphisms in \mathbf{C} . It's the profunctor representation.

7.7.1 Tambara modules

Definition 7.23. Let \mathbf{C} be a monoidal category. A Tambara module is a profunctor $P: \mathbf{C}^{\text{op}} \times \mathbf{C} \rightarrow \mathbf{Set}$ equipped with a family of functions

$$\alpha_{c,a,b}: P(a, b) \rightarrow P(c \otimes a, c \otimes b)$$

satisfying appropriate naturality conditions^a, and which is compatible with the monoidal structure on \mathbf{C} .

^a α must be natural in a and b and dinatural in c

Let's analyze this definition using the interpretation of profunctors as proof-relevant relations. If object a is related to b then, for any object c , the Tambara condition tells us that a combined with c is related to b combined with c . Such a relation is “focused” on a and b , in the sense that it's not broken by embedding those two objects in a common context provide by c .

Moreover, because of functoriality, this relation gets automatically extended to all objects that have a mapping to the source and a mapping out from the target of the profunctor.

For instance, if the set $P(a, b)$ is non-empty, then the set $P(c \otimes a, c \otimes b)$ is non-empty and, if there is a pair of morphisms $f: s \rightarrow c \otimes a$ and $g: c \otimes b \rightarrow t$, then the set $P(s, t)$ is also non-empty, because there is a function $P(f, g) \circ \alpha_{c,a,b}$ that maps $P(a, b)$ to $P(s, t)$.

The totality of such relations that are compatible with a particular tensor product—or the Tambara modules for this monoidal category—form a category. Morphisms in that category are natural transformations that preserve the Tambara structure.

Let's now turn the argument around. Let's pick a pair of objects a and b and another pair s and t . We will now vary the profunctors over the whole Tambara category. If it so happens that every time a is related to b it follows that s is related to t , then we can deduce that, secretly, there must be a bridge between the two pairs. In particular, there must exist a c and a pair of morphisms $f: s \rightarrow c \otimes a$ and $g: c \otimes b \rightarrow t$. In other words, we must have

$$\int_{P \in \text{Tam}b} \text{Set}(P(a, b), P(s, t)) \cong \int^c C(s, c \otimes a) \times C(c \otimes b, t)$$

The right hand side turns out to be the existential representation of an optic. The left hand side is an end in the category of Tambara modules. What's important is that it's just a family of functions between sets. As such, they can be composed using function composition.

The whole argument can be made precise with the use of the Yoneda lemma in the profunctor category.

7.7.2 Profunctor optics in Haskell

Since in Haskell we have two monoidal structures defined by the product and the coproduct, we can define two types of Tambara modules. They are known as **Cartesian**

and **Cocartesian** or, in the **Date.Profunctor** library, as **Strong** and **Choice**. Here's one possible definition

```
class Profunctor p => Cartesian p where
  alpha :: p a b -> p (c, a) (c, b)
```

```
class Profunctor p => Cocartesian p where
  coalpha :: p a b -> p (Either c a) (Either c b)
```

We've seen that Tambara optics can be defined as an end over Tambara modules. In Haskell, an end is expressed as a universally qualified data type. Therefore, a lens can be defined as

```
type Lens s t a b = forall p. Cartesian p => p a b -> p s t
```

```
type Prism s t a b = forall p. Cocartesian p => p a b -> p s t
```

Index

[filter](#), 118

bicartesian, 44

cartesian category, 43

cartesian closed category, 58

catamorphism, 105

Church encoding, 26

comments, [xiii](#)

compositionality, 1

existential types, 123

function, 4

generalized element, 95

global element, 94

global element , `textbf39`

hylomorphism, 119

inhabited type, 47

language pragmas, [xiii](#)

partial application, 61

pattern matching, 49

point free, 55

product, 40

sieve of Eratosthenes, 117

walking, 71

 arrow, 72

 isomorphism, 72

 object, 71

walking arrow category, 15

wildcard pattern, 101

zero object, 40