Programming with Categories (DRAFT)

Brendan Fong Bartosz Milewski David I. Spivak

(Last updated: January 29, 2020)

Contents

Preface **v**
Learning programming . . . vii
Installing Haskell . . . viii
Haskell points . . . ix
Acknowledgments . . . xi

1 Is Haskell a category? **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 The category of sets . . . 9
1.3.2 Definition of category . . . 11
1.3.3 Some elementary examples . . . 13
1.3.4 Generalized elements: a first peek at the Yoneda perspective . . . 19
1.4 Thinking of Haskell as a category . . . 22
1.4.1 The lambda calculus . . . 22
1.4.2 Types . . . 24
1.4.3 Haskell functions . . . 26
1.4.4 Composing functions . . . 27
1.4.5 So is Haskell a category? . . . 31
1.4.6 Many objections . . . 34

2 Functors and type constructors **37**
2.1 Introduction . . . 37
2.2 Functors . . . 37
2.2.1 Definition of functor, and first examples . . . 38
2.2.2 The category of categories . . . 39

2.2.3 Functors and shapes . . . 39
2.3 Functors in Haskell . . . 42
2.3.1 Interlude: polymorphic lambda calculus . . . 43
2.3.2 Type constructors . . . 44
2.3.3 Back to functors . . . 45
2.3.4 Type classes . . . 47
2.3.5 More interesting functors . . . 49
2.3.6 Strength and enrichment . . . 53
2.4 Natural transformations . . . 54
2.4.1 Definition . . . 54
2.4.2 Natural transformations in Haskell . . . 55

3 Universal constructions and the algebra of types **59**
3.1 Constructing datatypes . . . 59
3.2 The Yoneda embedding . . . 61
3.3 Zero and one: some first universal constructions . . . 64
3.3.1 Terminal objects . . . 64
3.3.2 Initial objects . . . 67
3.4 Products and pairs . . . 68
3.4.1 The pair type . . . 71
3.4.2 Using the universal property . . . 73
3.4.3 Functoriality of the product . . . 74
3.4.4 Record syntax . . . 76
3.5 Coproducts and sum types . . . 77
3.5.1 Sum types . . . 78
3.5.2 Properties of the coproduct . . . 80
3.5.3 Distributivity . . . 81
3.6 Exponentials and currying . . . 82
3.6.1 Currying in Haskell . . . 84
3.6.2 Functoriality of the exponential . . . 85
3.6.3 Functoriality in Haskell . . . 87
3.6.4 Distributivity revisited . . . 87

4 Algebras and recursive data structures **89**
4.1 Algebras . . . 93
4.2 Initial algebras . . . 94
4.2.1 Lambek's lemma . . . 95
4.3 Recursive data structures . . . 96
4.3.1 The essence of recursion . . . 99
4.3.2 Algebras, catamorphsims, and folds . . . 100
4.3.3 Lists . . . 101
4.4 Coalgebras, anamorphsims, and unfolds . . . 103

4.5 Fixed points in Haskell 106

5 Monads **111**

5.1 A teaser 111

5.2 Monads in Haskell 112

5.2.1 Monad and Kleisli composition 112

5.2.2 Monad and join 113

5.2.3 Monad and bind 113

5.2.4 `do` notation 114

5.2.5 Monads and effects 115

5.2.6 The list monad and nondeterminism 116

5.2.7 The writer monad 116

5.2.8 The reader monad 117

5.2.9 The state monad 117

5.2.10 The continuation monad 117

5.2.11 The `IO` monad 118

5.2.12 `Monad` as `Applicative` 119

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.

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` $\to$ `()`. 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` $\to$ `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.

Installing Haskell

Download and install Haskell Platform on your computer. (On the Mac or Linux, you'll have to close and reopen the terminal after installation). To start an interactive session from the command line, type `ghci` (GHC stands for the Glasgow Haskell Compiler or, more affectionately, among Haskell aficionados, 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 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. 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).

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.

Useful functions: `abs`, `floor`, `ceiling`, `round`; integral division `div` and `mod` satisfying the equality:

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

Square root , `sqrt`, trigonometric functions and the constant `pi`, natural exponential `exp` and logartithm `log`.

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

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

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:

```
module Test where
```

Module names must start with 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`. 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 follows 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.

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 pramgas 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.

Chapter 1

Is Haskell a category?

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 remove all the spaces. This capability is not provided to us in the base library of a language like Haskell. 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 `concat`, which takes a list of strings and concatenates them to return a single string. For example, we might run:

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

```
"Ilikecats"
```

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

```
Prelude> let de-space = concat . words
```

This produces a solution to our problem.

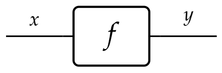
```
Prelude> de-space "Yay composition"
"Yaycomposition"
```

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.

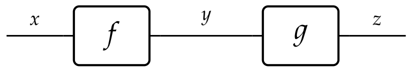
As coauthor-Bartosz once wrote: a category is an embarrassingly simple concept. A category is a bunch of objects, and some arrows that go between them. We assume nothing about what these objects or arrows are, all we have are names for them. We might call our objects very abstract names, let x, y, and z, or more evocative ones, like 42 or `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 $f\colon x \to y$ and an arrow $g\colon y \to z$, we may compose them to get an arrow with source x and target z. We denote this arrow $f \mathbin{⨾} g\colon x \to z$; we might also draw it as piping together two boxes:



This should remind you of our de-spacing example above, stripped down to its bare essence.

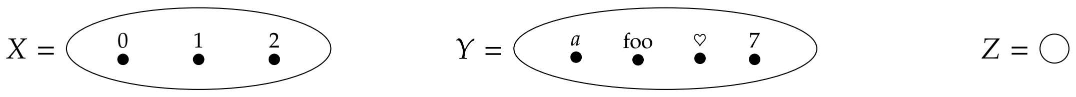
A category is a network of relationships, or slightly more precisely, a bunch of objects, some arrows that go between them, and a formula for composing arrows. A programming language generally has a bunch of types and some programs that go between them (i.e. take input of one type, and turn it into output 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. But first, 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 a bag of dots.



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* because cardinals were the first birds to recognize the importance of this concept. We denote the cardinality of X as $|X|$. 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	$\mathbb{N}$	$\{0, 1, 2, 3, \ldots, 42^{2048}+17, \ldots\}$
The nth ordinal	$\underline{n}$	$\{1, \ldots, n\}$
The empty set	$\varnothing$	$\{\}$
The integers	$\mathbb{Z}$	$\{\ldots, -59, -58, \ldots -1, 0, 1, 2, \ldots\}$
The booleans	$\mathbb{B}$	$\{\texttt{true}, \texttt{false}\}$

Exercise 1.2.

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

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.

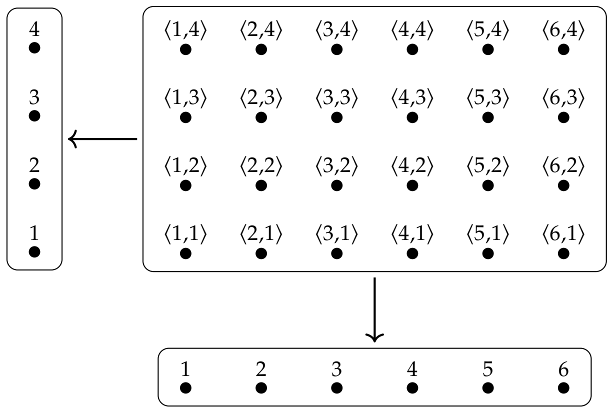
Exercise 1.4.

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 $\langle x, y\rangle$ 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 its projections:



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. ◊

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 $\mathbb{N} \times \underline{4}$. ◊

1.2.2 Functions

A function is 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 an answer, and you'll get exactly the same answer as if you put in 5 tomorrow.

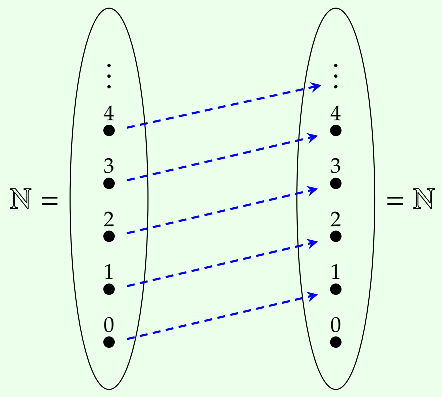
Definition 1.8 (Function)**.** Let X and Y be sets. A *function f from X to Y*, denoted $f\colon X \to Y$, is a subset of $f \subseteq X \times Y$ with the following properties.
1. For any $x \in X$ there is at least one $y \in Y$ for which $(x, y) \in f$.
2. For any $x \in X$ there is at most one $y \in Y$ 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 $(x, y) \in f$.

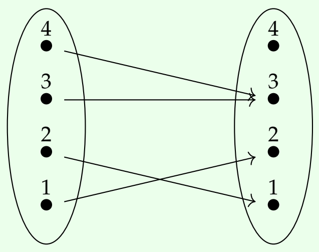
This is a rather abstract definition; perhaps some examples will help. One way of denoting a function $f : X \to Y$ is by drawing "maps-to" arrows $\mapsto$ that emanate from some particular $x \in X$ and point to some particular $y \in Y$. Every $x \in X$ gets exactly one arrow emanating from it, but no such rule for y's.

Example 1.9. The *successor* function $s : \mathbb{N} \to \mathbb{N}$ 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 1.10. Here's a picture of a function $\underline{4} \to \underline{4}$:



Exercise 1.11.

1. Suppose someone says "$n \mapsto n-1$ is also a function $\mathbb{N} \to \mathbb{N}$. Are they right?
2. Suppose someone says "$n \mapsto 42$ is also a function $\mathbb{N} \to \mathbb{N}$. 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.

$\lozenge$

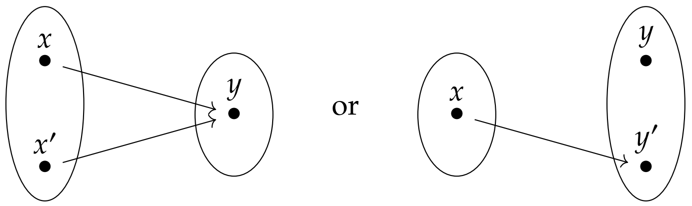
Exercise 1.12.

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

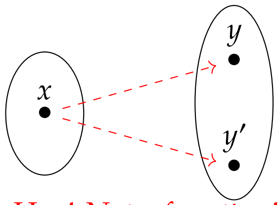
In general, for any $a, b \in \mathbb{N}$ there are b^a functions $\underline{a} \to \underline{b}$. You can check your answers above using this formula. The one case that you might be confused is when $a = b = 0$. In this case, a calculus teacher would say "the expression 0^0 is undefined", but we're not in calculus class. It may be true that as a functions of real numbers, there is no smooth way to define 0^0, but for natural numbers, the formula "count the number of functions $a \to b$" works so well, that 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 source into one element of the target or to miss elements of the target:



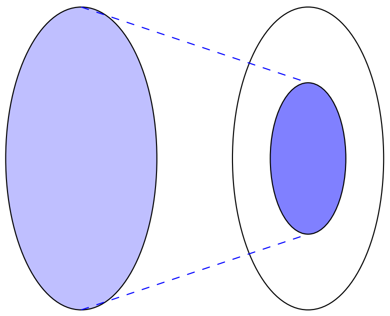
On the other hand, a function is forbidden from splitting a source element into multiple target elements.



There is another source of asymmetry: functions are defined for all elements in the source set. This condition is not symmetric: not every element in the target set has to be covered. The subset of elements in the target that are in a functional relation with the source is called the *image* of a function:

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

"The image of f is the set of y's in Y such that there exists an x in X where $f(x) = y$."



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.

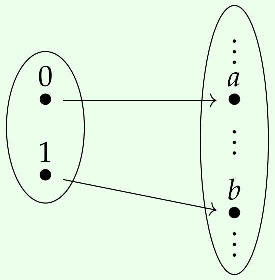
You may think of a function that maps many things to one as discarding some information. The function $\mathbb{N} \to \mathbb{B}$ that takes a natural number and returns `true` if it's even and `false` otherwise doesn't care about the precise value of a number, it only cares about it being even or odd. 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 *embedding* its source in a larger environment. It's creating a model of its source in a larger context, especially when it additionally collapses it by discarding some details. A helpful intuition is to think of the source set as defining a shape that is projected into a bigger set and forms a pattern there. 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 1.13. A singleton set 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 1.14. A two-element set can be used to pick pairs of elements in the target set. It embodies the idea of a pair.



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 1.15. 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 $i\colon \underline{2} \to N$, 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? ◊

Exercise 1.16. 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.) ◊

Remark 1.17. Category theory has a love/hate relationship with set theory. Secretly, category theorists dream of overthrowing the rule of set theory as the foundation of mathematics (more recently, homotopy type theorists have restarted this quest). But so far, even most category theorists continue to work within set-theoretic foundations. You'll see that we might skirt the issue by saying that category is a "bunch" of objects, because they don't really have to form a set-theoretical set, but then we can't say the same about morphisms between objects. They do form sets. Every time we draw a commuting diagram, we are asserting that two (or more) elements in some set of morphisms are equal. Granted, there are ways of postponing this fallback to sets by introducing enriched categories, but even these eventually force you to draw diagrams that, at some level, assert the equality of set elements. The bottom line is that we have to have some knowledge of set theory before we tackle category theory. We can think of sets as these primordial, structureless categories, but in order to build more interesting structures we have to start from the structureless.

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.

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. Instead of beginning directly with a definition, we'll motivate the definition by discussing the ur-example: the category of sets and functions.

1.3.1 The category of sets

The identity function on a set X is the function $\mathrm{id}_X \colon X \to X$ given by $\mathrm{id}_X(x) = x$. 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 1.18. Let $f \colon X \to Y$ and $g \colon Y \to Z$ be functions. Then their *composite*, denoted either $g \circ f$ or $f \mathbin{⨟} g$, is the function $X \to Z$ sending each $x \in X$ to $g(f(x)) \in Z$.

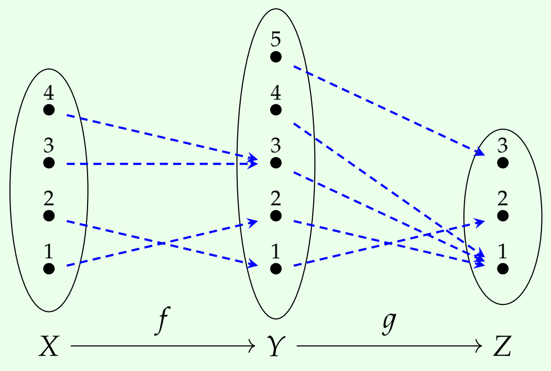
The above definition makes it look like $g \circ f$ is better notation, because $(g \circ f)(x) = g(f(x))$ looks than the backwards-seeming formula $(f \mathbin{⨟} g)(x) = g(f(x))$, which has a sort of switcheroo built in. But there are good reasons for using $f \mathbin{⨟} g$, e.g. this diagram

$$\begin{tikzcd}
 & Y \arrow[dr, "g"] & \\
X \arrow[ur, "f"] \arrow[rr, "f ⨟ g"'] & & Z
\end{tikzcd}$$

Another is that—as we saw in Example 1.13—an element of X is just a function $\underline{1} \to X$. From this point of view, function application is just composition. That is, if $f \colon X \to Y$ is a function then what's normally denoted $f(x)$ could instead be denoted $x \mathbin{⨟} f$

$$\begin{tikzcd}
 & X \arrow[dr, "f"] & \\
1 \arrow[ur, "x"] \arrow[rr, "x ⨟ f"'] & & Y
\end{tikzcd}$$

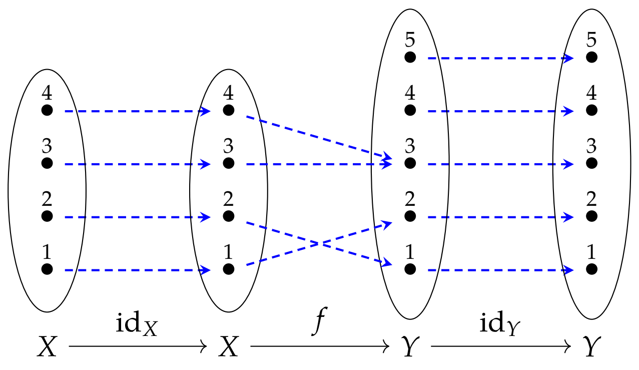
Example 1.19. A great way to visualize function composition is by path following.



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

Now the above backwards-seeming formula becomes $x \mathbin{⨾} (f \mathbin{⨾} g) = (x \mathbin{⨾} f) \mathbin{⨾} g$, and the switcheroo is gone. That is, we see that the preference for the $\circ$ notation is built in to the $f(x)$ notation, which is already backwards. In the $⨾$ direction, you start with an x, then you apply f, then you apply g, etc.

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



4
3
2
1
4
3
2
1
5
4
3
2
1
5
4
3
2
1
$X \xrightarrow{\mathrm{id}_X} X \xrightarrow{f} Y \xrightarrow{\mathrm{id}_Y} Y$

Composing with the identity doesn't do anything.

The second property we want to highlight 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 $V \xrightarrow{e} W \xrightarrow{f} X \xrightarrow{g} Y \xrightarrow{h} Z$ as any of the five ways represented in the pentagon below:

$$\begin{tikzcd}[column sep=small]
 & & e \fatsemi (f \fatsemi (g \fatsemi h)) \arrow[dll, dotted, no head] \arrow[drr, dotted, no head] & & \\
e \fatsemi ((f \fatsemi g) \fatsemi h) \arrow[dr, dotted, no head] & & & & (e \fatsemi f) \fatsemi (g \fatsemi h) \arrow[dl, dotted, no head] \\
 & (e \fatsemi (f \fatsemi g)) \fatsemi h \arrow[rr, dotted, no head] & & ((e \fatsemi f) \fatsemi g) \fatsemi h &
\end{tikzcd} \tag{1.20}$$

It turns out that all these different ways to collapse four functions give the same answer. You could write it simply $e \fatsemi f \fatsemi g \fatsemi h$ 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 $(f \fatsemi g) \fatsemi h = f \fatsemi (g \fatsemi h)$.

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

Exercise 1.21. Consider the pentagon (sometimes called the *associahedron*) in Eq. (1.20).

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?

$\lozenge$

1.3.2 Definition of category

We begin with a slogan:

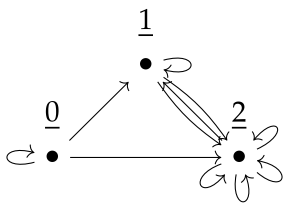
A category is an organized network of relationships.

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

[1]Actually, **Set** is the category of *small sets*, meaning sets all of whose elements come from some huge pre-chosen universe $\mathbb{U}$. The axiom of *Grothendieck universes*, which says that there's an infinite ascending hierarchy of these $\mathbb{U}$'s uses some pretty heavy set theory to make sure you don't have to worry about weird paradoxes that come about when one allows themselves to speak of the infamous 'set of all sets.' For a given universe $\mathbb{U}$ the purportedly 'small sets' can include uncountably infinite sets, as long as they're in $\mathbb{U}$. This way, rather than talking about the set of all sets, we talk about the (larger) set of all (small) sets, and everything is just a set, albeit in different universes.

The point is, you don't need to worry about all these "size issues"; just focus on the category theory for now. If you want to get deep into the technical set-theoretic issues, see [**].

network of arrows pointing from one set to another.



(and if you toss in $\underline{3}$, 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) (1.22)

But **Set** not just any old network: it's organized in the sense that we know how to compose the functions. This imposes a tight constraint: if pretty much any function was somehow left out of the network, it would cause a huge catastrophe of missing composites.

Let's see the precise definition.

Definition 1.23. A *category* $\mathcal{C}$ consists of four constituents:

1. a set $\mathrm{Ob}(\mathcal{C})$, elements of which are called *objects of* $\mathcal{C}$;
2. for every pair of objects $c, d \in \mathrm{Ob}(\mathcal{C})$ a set $\mathcal{C}(c,d)$, elements of which are called *morphisms from* c *to* d and often denoted $f \colon c \to d$;
3. for every object c, a specified morphism $\mathrm{id}_c \in \mathcal{C}(c,c)$ called the *identity morphism for* c; and
4. for every three objects b, c, d and morphisms $f \colon b \to c$ and $g \colon c \to d$, a specified morphism $(f \mathbin{;} g) \colon b \to d$ called the *composite of* f *and* g, sometimes denoted $g \circ f$.

These constituents are subject to three constraints:

Left unital: for any $f \colon c \to d$, the equation $\mathrm{id}_c \mathbin{;} f = f$ holds

Right unital: for any $f \colon c \to d$, the equation $f \mathbin{;} \mathrm{id}_d = f$ holds

Associative: for any $f_1 \colon c_1 \to c_2$, $f_2 \colon c_2 \to c_3$, and $f_3 \colon c_3 \to c_4$, the equation $(f_1 \mathbin{;} f_2) \mathbin{;} f_3 = f_1 \mathbin{;} (f_2 \mathbin{;} f_3)$ holds

Example 1.24 (The category of sets). The category of sets, denoted **Set** has all the sets[a] as its objects the sets. Given two sets $A, B \in \mathrm{Ob}(\mathbf{Set})$, we have $\mathbf{Set}(A,B) := \{f \colon A \to B \mid f \text{ is a function}\}$. There's sets everywhere: objects are sets, for every two objects $\mathbf{Set}(A,B)$ is also a set. This situation where the collection of morphisms $A \to B$ itself forms an object is a distinctive feature of certain categories that we'll see a lot of later, called "closed categories." For example, just as there is a *set* of functions from one set to another, a student of linear algebra might know that there is a *vector space* of linear transformations from one vector space to another.

But you don't have to worry about closed categories for now; we'll get to it.

[a] In some universe $\mathbb{U}$; don't worry about this for now.

Exercise 1.25. Let's return to a remark we made earlier about the category **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? ◊

1.3.3 Some elementary examples

Even though the notion of category is somehow modeled on **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 perhaps even more important than the number 9^9.

Example 1.26. There is a single-object category **1** with no other arrows but the identity arrow.

$$\boxed{\begin{tikzcd} 1 \arrow[loop above, "\mathrm{id}_1"] \end{tikzcd}}$$

Example 1.27 (Discrete categories). You can also have a category with two objects 1 and 2 and two identity morphisms id_1 and id_2.

$$\mathbf{Disc}(2) = \boxed{\begin{tikzcd} 1 \arrow[loop above, "\mathrm{id}_1"] & 2 \arrow[loop above, "\mathrm{id}_2"] \end{tikzcd}}$$

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

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

In particular, the empty category is given by $\mathbf{0} = \mathbf{Disc}(\underline{0})$.

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 $ar\colon 1 \to 2$ to the two-object category.

Example 1.28. This tiny category is sometimes called the *walking arrow category* **2**.

$$\mathbf{2} := \boxed{\begin{tikzcd} \overset{1}{\bullet} \arrow[loop left, "\mathrm{id}_1"] \arrow[r, "f"] & \overset{2}{\bullet} \arrow[loop right, "\mathrm{id}_2"] \end{tikzcd}}$$

Exercise 1.29. Show that the walking arrow category **2** satisfies all the laws of a category. ◊

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 $f : a \to b$ and $g : b \to c$, we often leave these out of our pictures.

Example 1.30 (Not drawing all morphisms). In the picture $\boxed{\bullet \to \bullet \to \bullet}$, 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 1.28 as

$$\mathbf{2} := \boxed{\begin{tikzcd} \overset{1}{\bullet} \arrow[r, "f"] & \overset{2}{\bullet} \end{tikzcd}}$$

Example 1.31 (Ordinal categories). There is a progression of *ordinal categories* that look like this:

$$\begin{array}{ccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \\ \boxed{} & \boxed{\overset{1}{\bullet}} & \boxed{\begin{tikzcd} \overset{1}{\bullet} \arrow[r, "f_{12}"] & \overset{2}{\bullet} \end{tikzcd}} & \boxed{\begin{tikzcd} \overset{1}{\bullet} \arrow[r, "f_{12}"] & \overset{2}{\bullet} \arrow[r, "f_{23}"] & \overset{3}{\bullet} \end{tikzcd}} & \cdots \end{array}$$

Adherents of the religion of Twodeism—called Twoish people—believe that **2** is the categorical manifestation of the creator of all things. (Twoish people also worship the set $\underline{2}$, its "higher-categorical" analogues, and various products of these. Among other pastimes, they enjoy working in *cubical type theory*.)

Exercise 1.32. Being sure to take into account Example 1.30,

1. How many morphisms are there in **3**?
2. How many morphisms are there in **n**?
3. Are **0** and **Disc**(0) the same?
4. Are **1** and **Disc**(1) the same? ◊

Example 1.33 (The walking isomorphism). In the following category, which we will call

I, there are two objects and four morphisms:

$$\begin{tikzcd} \overset{1}{\bullet} \arrow[r, bend left, "f"] & \overset{2}{\bullet} \arrow[l, bend left, "g"] \end{tikzcd}$$

$$g \circ f = \mathrm{id}_1$$
$$f \circ g = \mathrm{id}_2$$

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. We see that f and g are inverses; each is an isomorphism in the sense of Definition 1.52. 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 1.34. Suppose that someone tells you that their category $\mathcal{C}$ has two objects c, d and two non-identity morphisms, $f\colon c \to d$ and $g\colon d \to c$, but no other morphisms. Does f have to be the inverse of g, i.e. is it forced by the category axioms that $g \circ f = \mathrm{id}_c$ and $f \circ g = \mathrm{id}_d$? ◊

Exercise 1.35. Show that a category with only two nontrivial arrows going in the opposite direction satisfies all the laws of a category. In particular, show that the two morphisms must be the inverse of each other. ◊

Example 1.36 (Monoids). A monoid is like a video game controller: it's a set of "things you can do in sequence". More precisely, we have the following definition.

Definition 1.37. A *monoid* $(M, e, \diamond)$ consists of

1. a set M, called the *carrier set*;
2. an element $e \in M$, called the *unit*; and
3. a function $\diamond\colon M \times M \to M$, called the *operation*.

These are subject to two conditions:

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

For example, $(\mathbb{N}, 0, +)$ is called 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. $5 + 6 + 2 + 2$.

What's the point? It turns out that a monoid can always be regarded as a *category* *with one object*.[a] If a category has one object, say

[a]The term monoid comes from mono+id, where *mono* means one and *id* means the Freudian self.

Example 1.38. We don't expect you to know Haskell at this point, but here's a preview of how one might define categories and explain to the compiler that monoids are categories:[a]

```
class Category obj mor | mor -> obj where
  dom :: mor -> obj
  cod :: mor -> obj
  idy :: obj -> mor
  cmp :: mor -> mor -> Maybe mor
```

The "Maybe" part is saying that two morphisms may not compose (e.g. $f\colon a \to b$ and $g\colon b' \to c$ only compose if $b = b'$); see Example 2.30. Note also that there are no laws—associative or unital—so the user of this class has to just certify that these laws really do hold in their specific case. Finally, the weird `| mor -> obj` thing at the top is called a "functional dependency" and is just there to soothe the compiler.

In Haskell, there is already a `monoid` class; what we call the unit it calls `mempty` (like monoid-empty), and it denotes the operation we write as $\diamond$ by <>. So now we explain to the compiler that a monoid is a category with one (denoted `()` in Haskell) object:

```
instance Monoid mor => (Category () mor) where
  dom _   = ()
  cod _   = ()
  idy _   = mempty
  cmp m n = Just (m <> n)
```

[a]The code above doesn't quite compile; you need to enable some language extensions first:

```
{-# language FlexibleInstances #-}
{-# language MultiParamTypeClasses #-}
{-# language FunctionalDependencies #-}
```

Exercise 1.39. Implement the category $\mathbf{2} = \boxed{\bullet \to \bullet}$ from Example 1.31 as an instance of the `Category` class from Example 1.38. $\lozenge$

Monoids can be regarded as categories with one object. They're categories that are tiny in terms of objects—just one! Not quite analogously, but similarly, we'll next discuss categories which are tiny in terms of morphisms.

Example 1.40 (Preorders). A *preorder* is a category such that, for every two objects a, b, there is at most one morphism $a \to b$. 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 $a \to b$, we write $a \leq b$; if there is not a morphism $a \to b$, we don't.

For example, there is a preorder $\mathcal{P}$ whose objects are the positive integers $\text{Ob}(\mathbf{P}) = \mathbb{N}_{\geq 1}$ 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 dividible by a) or contains exactly one element.

Exercise 1.41. Consider the category $\mathcal{P}$.

1. What is the identity on 12?
2. Show that if $x : a \to b$ and $y : b \to c$ 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? ◊

Example 1.42. In Haskell there are various type classes. A type class is basically a template for named functions of certain types: it's a bunch of key words—each referring to a function of some type—that any member of the class needs to give meanings for.

For example, the type class `Show` is the template {`show`}: any type `T` that wants to be in `Show` needs to say what the word `show` means to them; it needs to say what function `show:: T -> String` should mean. The type class `Eq` is the template for the words `==` and `/=`; any member `T` of that class needs to say what functions `==:: T -> Bool` and `/=:: T -> Bool` should mean. We'll see more on type classes in Section 2.3.4.

There is a hierarchy of type classes in Haskell, and a hierarchy is another name for preorder; there is a preorder of type classes in Haskell. Here's a picture of part of it:

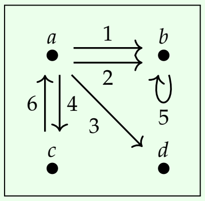
$$\begin{tikzcd}
\boxed{\text{Eq}} & \\
\boxed{\text{Ord}} \arrow[u] & \boxed{\text{Num}} \\
\boxed{\text{Real}} \arrow[u] & \boxed{\text{Fractional}} \arrow[u] \arrow[ul]
\end{tikzcd}$$

Our arrows point in the opposite direction from the way most people write it. Our arrows $A \to B$ point in the direction "if something is of type A then it is also of type B".

If you prefer the other order, please apply $^{\mathrm{op}}$, defined in Example 1.45, thus answering Exercise 1.46.

Example 1.43 (Free category on a graph). A graph consists of two sets V, E and two functions $\mathrm{src}\colon E \to V$ and $\mathrm{tgt}\colon E \to V$. The elements of V are called vertices, the elements of E are called edges, and for each edge $e \in E$, the vertex $\mathrm{src}(e)$ is called its source and the vertex $\mathrm{tgt}(e)$ 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**
1	a	b
2	a	b
3	b	b
4	a	c
5	b	b
6	c	a

Vertex
a
b
c
d

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.

Exercise 1.44. Is the ordinal category $\mathbf{3}$ (see Example 1.31) the free category on a graph? If so, which graph; if not, why not? ◊

Example 1.45 (Opposite category). For any category $\mathcal{C}$, there is a category $\mathcal{C}^{\mathrm{op}}$ 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:

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

Exercise 1.46. If $\mathbf{TC}$ is the category shown in Example 1.42, how would you draw $\mathbf{TC}^{\mathrm{op}}$? ◊

Example 1.47 (Product category). Suppose $\mathcal{C}$ and $\mathcal{D}$ are categories. We can form a new category $\mathcal{C} \times \mathcal{D}$ as follows:

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

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

$$\begin{tikzcd}
\langle 1,1\rangle \arrow[r, "\langle \mathrm{id}_1, f\rangle"] \arrow[d, "\langle f, \mathrm{id}_1\rangle"'] \arrow[dr, "\langle f, f\rangle" description] & \langle 1,2\rangle \arrow[d, "\langle f, \mathrm{id}_2\rangle"] \\
\langle 2,1\rangle \arrow[r, "\langle \mathrm{id}_2, f\rangle"'] & \langle 2,2\rangle
\end{tikzcd}$$

$$\langle \mathrm{id}_2, f\rangle \circ \langle f, \mathrm{id}_1\rangle = \langle f, f\rangle$$
$$\langle f, \mathrm{id}_2\rangle \circ \langle \mathrm{id}_1, f\rangle = \langle f, f\rangle$$

Example 1.48 (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 $D \subseteq \mathrm{Ob}(\mathcal{C})$, 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 $\mathcal{C}_{\mathrm{Ob}=D}$. It's defined by

$$\mathrm{Ob}(\mathcal{C}_{\mathrm{Ob}=D}) \coloneqq D \qquad \text{and} \qquad \mathcal{C}_{\mathrm{Ob}=D}(d_1, d_2) = \mathcal{C}(d_1, d_2).$$

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

Exercise 1.49. Does the picture shown in Eq. (1.22) look like the full subcategory of **Set** spanned by $\{\underline{0}, \underline{1}, \underline{2}\} \subseteq \mathrm{Ob}(\mathbf{Set})$? Why or why not? ◊

Exercise 1.50. Make an instance of the class `Category` from Example 1.38 that defines the full subcategory of Hask spanned by two objects, `Bool` and `Int`. ◊

1.3.4 Generalized elements: a first peek at the Yoneda perspective

A category is a web of relationships. 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 1.13 that elements of a set X are in one-to-one correspondence with functions $1 \to X$. Inspired by this, we often abuse our language to say that 'an element of X' *is* a function $x\colon 1 \to X$. 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 1.51. 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 $e\colon c \to a$.

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 $f\colon a \to b$. 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*

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

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 next chapter, with a few more tools under our belt. First though, one more lesson in the philosophy of category theory.

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

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 1.52. Let a and b be objects in a category $\mathcal{C}$. We say that a morphism $f\colon a \to b$ is an *isomorphism* if there exists $g\colon b \to a$ such that $f \mathbin{;} g = \mathrm{id}_a$ and $g \mathbin{;} f = \mathrm{id}_b$. We will call g the *inverse* of f, and sometimes use the notation f^{-1}.

If an isomorphism $f\colon a \to b$ 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 1.53. 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 $f\colon a \to b$ 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 1.54. Let $f\colon a \to b$ be an isomorphism. Then for all objects c, we have a bijection

$$\begin{tikzcd} \mathcal{C}(c,a) \arrow[r, shift left, "- \fatsemi f"] & \mathcal{C}(c,b) \arrow[l, shift left, "- \fatsemi f^{-1}"] \end{tikzcd}$$

To see this, consider the following diagram.

$$\begin{tikzcd} & c \arrow[dl, "h"'] \arrow[dr, "h'"] & \\ a \arrow[loop left, "\mathrm{id}_a"] \arrow[rr, bend left, "f"] & & b \arrow[ll, bend left, "f^{-1}"] \arrow[loop right, "\mathrm{id}_b"] \end{tikzcd}$$

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

Exercise 1.55.

1. What is the composite $- \fatsemi f$ followed by $- \fatsemi f^{-1}$?
2. Prove Proposition 1.54.

$\Diamond$

In other words, Proposition 1.54 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 $- \fatsemi f$ it may collapse two generalized elements of a, or may not cover the entire set of generalized elements of b.

1.4 Thinking of Haskell as a category

Enough 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, [2] such as `concat` and `words`, and compose them using the `.` syntax to get a new function. Now equipped with a definition of category, we can see that these functions might be considered as morphisms in some category. But there are questions to be answered. First, what is a Haskell function? And second, if Haskell functions are the morphisms, what are the objects?

A function is just a special kind of relationship between the elements of two sets. In programming, a function is defined by a formula. Such formulas are constructed using rules, defined by the syntax of a language. This brings us to the lambda calculus.

1.4.1 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 $\lambda x.A$. 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 some, a 'function' of x.

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 substituting in B wherever we see x. 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:

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

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. So, for example, we have

$$(\lambda x.x)y = y \qquad (\lambda x.x)(xy) = xy \qquad \text{and} \qquad (\lambda x.(xy)x)(zz) = ((zz)y)(zz).$$

[2]Note that we've defined the word function as a special sort of relationship between sets. Programming also often refers to functions, like `cat` or `words`. These two notions are indeed related, but distinct ideas, and care should be taken not to get them confused. If you're ever confused, it's a good idea to ask what sort of function is being referred to: mathematical or programming?

Notice that the lambda term $\lambda x.x$ behaves like an identity function: given some x, it just returns x. Haskell lets us almost directly employ this same notation; we would write it:

```
id = \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.

Identity is part of the standard Haskell library called **Prelude**. 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
```

Haskell Note 1.56 (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.

Exercise 1.57 (Church Booleans). Boolean logic may be encoded in the lambda calculus using the following definitions:

$$\begin{aligned}\text{True} &= \lambda x.(\lambda y.x)\\ \text{False} &= \lambda x.(\lambda y.y)\\ \text{AND} &= \lambda p.(\lambda q.(pq)p)\\ \text{OR} &= \lambda p.(\lambda q.(pp)q)\end{aligned}$$

Evaluate the lambda terms True AND False and False OR True. ◊

Exercise 1.58 (The Y combinator). The Y combinator is an iconic lambda term whose reduction does not terminate. It is defined as follows:

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

Compute Yg. ◊

1.4.2 Types

Untyped 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 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. And because everything is a function in lambda calculus, it's okay to apply a number to a function and get meaningless result. Imagine debugging programs written in lambda calculus! [3]

Adding types to lambda calculus means that functions can't be composed willy-nilly: the target type of one must match the source type of the next one. This immediately suggests a category in which objects are types and morphisms are functions.

More properly then, we should say Haskell's syntax is based on the *simply-typed lambda calculus*. Every Haskell term is required to have a type. If a term `x` has type `A`, we write:

```
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]
```

Exercise 1.59. Fire up GHCi. What are the types of the following terms:

1. `42`
2. `"cat"`
3. `True`

◊

Exercise 1.60. Recall the Y combinator from Exercise 1.58. Try to assign the Y combinator a type. What goes wrong? ◊

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

In Haskell we have some built-in types, like **Integer**, which is an arbitrary precision integer, 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 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 have a variable `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 within it, and will be able to work with a bare minimum of type declarations.

Haskell Note 1.61. In Haskell, the names of concrete types start with an upper case letter, e.g. `Int`. The names type variables, or type parameters, like `a` here, start with lowercase letters.

Notice that 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 2.4.2. For now, it's enough to know that a polymorphic function is defined for all types. (This is the troubling self-referential aspect of polymorphism: "all types" includes the type we are just defining. For instance, `id` can be instantiated for the type `a = b -> b` and so on.)

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]

[a]Note that the period after the `forall` clause is just a separator and has nothing to do with function composition, which we'll meet shortly.

Remark 1.62. 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 – that's exactly what A is, a bag of dots. Types and terms are the other way around: terms have types. So you can always ask what the type of a term is, but you can't ask for all the terms of a given type.

1.4.3 Haskell functions

In category theory, we write $f: a \longrightarrow b$ 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:

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

In Haskell, we can also declare that terms are (Haskell) functions, using an almost identical syntax. Given two Haskell terms `A` and `B`, there is a *type of functions from* `A` *to* `B` `A -> B` whose terms are (Haskell) functions accepting an `A` and producing a `B`. So for example, we have

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

We call the type `A -> B` the *type of functions*. If you're worried that we are mixing types, which are objects in our category, with functions between them, which are morphisms, this will become clear when we talk about exponentials.

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 a natural number, so we write $f: \mathbb{N} \to \mathbb{N}$. We might then define the function itself: $f(n) = n^2$. 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 :: Int -> Int
```

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
```

Remark 1.63 (A note on Haskell functions vs mathematical functions). "Functions" in programming are not just simple functions between sets. A "function" may loop forever, be it because it enters an infinite loop or because of recursion. So maybe we don't need recursion? Maybe we could define a programming language in which the execution of every function is guaranteed to terminate? One way to do this would be to ban recursion altogether: no function could call itself (or, more precisely, the call graph cannot contain cycles). Unfortunately, most algorithms and data structures used in programming are defined recursively (or, equivalently, using looping constructs).

In computer science we say that a language without recursion or some kind of looping construct is not Turing complete. We could ask if it's possible to somehow limit recursion and allow only recursive definitions that are bound to terminate. Often it's possible to show that a recursive step reduces the problem to a simpler one, until it reaches some kind of a bottom (e.g., counting down from n to zero is bound to terminate for any n). Unfortunately, it's also known that termination is undecidable. There is no algorithm that could decide whether an arbitrary program will terminate or not.

For these reasons, general-purpose programming languages allow unlimited recursion and non-termination. There are ways of dealing with this problem within set theory. Recursion can be described in domain theory, which is based on partially ordered sets. Later, we'll talk about polymorphism, which also stretches the boundaries of set theory by introducing self-referential types. Again, there are ways of dealing with it in set theory, but they are not easy. Category theory provides a much simpler high-level setting for interpreting programs.

1.4.4 Composing functions

What do we do with functions? Compose them! As function composition is such a basic operation, in Haskell we give it almost the simplest name:

```
.
```

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

Haskell Note 1.64. 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? The Haskell definition takes advantage of *currying*. We'll talk much more about this in Chapter 3, 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 `f :: b -> c` and `g :: 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:

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

The result of calling `g` with the argument `x` is being passed to `f`. Notice the parentheses around `g x`. Without them, `f g x` would be parsed as: `f` is a function of two arguments being called with `g` 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:

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

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

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

Exercise 1.65. Suppose $f\colon \texttt{Int} \to \texttt{Int}$ sends an integer to its square, $f(x) := x^2$ and that $g\colon \texttt{Int} \to \texttt{Int}$ sends an integer to its successor, $g(x) := x + 1$.

1. Write f and g in Haskell, including their type signature and their implementation.
2. Let $h := f \circ g$. What is $h(2)$?
3. Let $i := f \mathbin{⨟} g$. What is $i(2)$? ◊

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

Example 1.66. 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 `length` to calculate the length of the list of strings from the previous call:[a]

```
Prelude> length ["Hello","world"]
2
```

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> (length . words) "Hello world"
2
```

[a]Note that writing "`concat it`" also works in GHCI.

Haskell Note 1.67 (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.

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

We can define new functions by composing existing functions:

```
Prelude> let wordCount = length . words
Prelude> wordCount "Hello Haskell!"
2
```

Haskell Note 1.68 (let). In the interactive environment, all definitions must be prefixed with `let`. In a standalone file, `let` is only used in local definitions, inside a function body. The difference in syntax is there because all input and output must be executed in the `IO` monad (which we'll discuss later). The command line interface must therefore use monadic syntax, which is normally seen in `do` blocks.

Exercise 1.69. Recall the notion of an isomorphism from Definition 1.52. Here's an example of an isomorphism in Haskell (don't worry about the syntax, it will become clear later) between two types `Coin` and `Bool`. It consists of two function, one being the inverse of the other.

```
data Coin = Head | Tails

decide :: Coin -> Bool
decide Tail = False
decide Head = True

unDecide :: Bool -> Coin
unDecide True = Head
unDecide False = Tail
```

You can easily convince yourself by case analysis that `unDecide` is the inverse of `decide`, but this fact is not expressible in Haskell. There are programming languages like Idris or Agda, which use dependent type systems, and which treat proofs of equality as first order constructs that can be passed between functions or stored in data structures.

Identity is also an example of an isomorphism. Here's an example of a non-identity isomorphism that maps Boolean to Boolean by inverting the meaning of `True` and `False`:

```
not :: Bool -> Bool
not True = False
not False = True
```

Explain why these two functions are isomorphisms, and why `Coin` and `Bool` are isomorphic. ◊

1.4.5 So is Haskell a category?

Haskell is not a category, it's a programming language. But it is similar enough to a category that, when working with Haskell, it is productive to think of it as being category-like.

Perhaps a useful analogy is the Haskell type `Int`. This is usually 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` 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 our central metaphor, the objects of the metaphorical category, which we might refer to as **Hask**, are the types of Haskell: `String` and `Int` and `(Int, Bool)`, etc. Given two types `A` and `B`, the morphisms from `A` to `B` are the terms of type `A -> B`, e.g. the term `length :: [Bool] -> Int` once it's been defined. Composition is given by the polymorphic function `(.)`, meaning that for any terms `f :: A -> B` and `g :: B -> C`, where $A, B, C \in$ **Hask** are concrete types (like `String` and `(Bool, Int)`, etc.), we can form the term `g . f` and it will have type `A -> C`. Similarly the identities are given by the polymorphic function `id`, meaning that for any type `A` there is a term `id` of type `A`. Associativity of composition means that we can write:

```
h . g . f
```

without ambiguity, while the unit law means `f . id` and `id . f` both return the same value as `f`.

This looks very much like a category. So why is it not? One issue is that there really isn't a set of Haskell data types. For example, in one program we may have `data Foo = Baz Int`, and in another we may have `data Foo = Baz String` or maybe `data Foo = Qux String` or `data Bar = Baz String`. Should any two of these be considered the same, or are they all different? And what about the fact that we can only use the type constructor `Foo` once in any given program and similarly we can only use the data constructor `Baz` once in any given program? Again: what exactly should be the set of objects in our category?

If we consider them all the same, then we need to carefully define what it means for two data types to be the same. If we consider them all different, then we might want to say that there are isomorphisms between them, but these isomorphisms can't be written in Haskell! If you try the following, the compiler will complain:

```
data Foo = Baz String
data Foo = Qux String --oops, can't do that! Ok, let's go with it..
```

```
i :: Foo -> Foo          --oh dear? which Foo is which?
i (Baz s) = Qux s        --ok, I see, from the first to the second..
j :: Foo -> Foo          --and back from the second to the first..
j (Qux s) = Baz s        --fine, I see what you mean, but Haskell won't!
```

That program—an attempt to show that these two data types are isomorphic—cannot be written.

Exercise 1.70. Explain why you can't write a Haskell program that gives maps between the data types `data Foo = Baz String` and `data Bar = Baz String`, and therefore that Haskell has no idea they are the same or isomorphic in any sense. ◊

Let's get around this issue by saying that two types are the same if they differ only by a renaming of the data constructors and/or type constructors. With this, we have defined our set of objects. The next question is what are the morphisms `A` to `B`. One could say "all Haskell functions from `A` to `B` that compile", or perhaps "all well-formed terms of type `A -> B`", which we consider the same thing. But this definition has problems because it's hard to say when two such functions should be considered equal.

One possibility is to say that two functions `f,g :: A -> B` should be considered equal when they consist of *exactly the same text*. Composition is given by simply writing the text of one, then a `.`, then the text of the other. This almost works, but it has two problems: it both fails the test and is useless. It's useless because the whole point of using category to understand *what programs do*, not *how programs are written*. Even if it were a category, none of the neat universal properties Chapter 3, which makes category theory a useful lens for programming will hold in this category. But worse, it's not even a category.

Exercise 1.71. For any Haskell type `A`, let `idA :: A -> A` denote the function `idA a = a`. Show that with the "text-based" definition of morphisms and composition above:

1. `idA` does not satisfy the criterion of being an identity.
2. no Haskell function of type `A -> A` will satisfy the definition of being an identity.

◊

So we really should not use the text of Haskell programs to define the morphisms, but instead "what that text does." That is, we should say that two Haskell functions `f,g :: A -> B` are considered to be the same morphism if, for every closed term `a :: A`, one has that `f a` equals `g a`. This is not something you can ask Haskell: even for a single `a` you cannot always ask `f a == g a`; that requires that `B` is in the type class `Eq`. But much worse, you certainly cannot ask it to check equality on every closed term `a :: A`, because it does not keep a copy of that set.

We thus want to say that two morphisms `f` and `g` are equal if we can prove (to ourselves on paper or in a proof assistant) that they do the same thing: for every input they give the same output. In this case, we say that `f` and `g` are *extensionally equivalent*.

The next problem comes with non-termination, i.e. when a program never finishes running. This happens with recursive functions, which we'll get to soon. For example consider the function

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = factorial (n - 1) * n
```

Type in `factorial 8000`, and it'll tell you the answer in less than 1 second. But type in `factorial (-1)` and it runs forever. We say it's nonterminating.

So you have two options: include nonterminating functions or not. If you want to include nonterminating functions, you need to say when two are equal. We could just say that they are equal when they give the same answer for any input, where nontermination counts as an answer. That is, functions `f,g :: A -> B` are the same if, for every `a :: A`, either neither `f a` nor `g a` terminates, or they both terminate and give the same answer.[4] If you don't want to include nonterminating functions as morphisms in your category, that's fine but note: there is no computer program that can decide whether functions terminate or not. You might write a certain Haskell program and never know if it is in the category or not.

Which is better? In this book, we will work with the latter: only consider terminating functions up to extensional equivalence, defined above. This choice will make `()` a terminal object and `Void` an initial object and have all sorts of nice properties. The other choice (allowing nontermination) is fine, but not our choice here.

Definition 1.72. We define the category **Hask** to be the category defined as follows.

Ob **Hask** is the set of Haskell data types `A,B,...`, where two are considered equivalent if one can be obtained from the other by renaming either type constructors or data constructors.

For any types `A` and `B`, the hom-set **Hask**(`A`,`B`) is the set of all compiling Haskell functions `A -> B` which terminate for every input (an input is a compiling Haskell function `() -> A` that terminates on `()`), where two are considered the same if they give the same value for every input.

For any `A`, the identity on it is `identityA a = a`, and composition is given by `(.)`.

[4]It may also be that a function runs out of space or memory for its computation. You again have a choice: either say that we only want functions that don't run out of space or memory, or we say that we are considering the functions abstractly, and disregard memory. We will do the latter; it seems the former may also work, though you need to define a special compose operator that forces evaluation, using `{-## LANGUAGE BangPatterns ##-}` and then `(!.) !f !g = f . g`. Otherwise, it may be that neither `f` and `g` run out of memory but `f . g` does. We first saw this issue raised in [**DanPiponi**].

Exercise 1.73. Which of the following pairs of Haskell data types do we consider to be equal as objects in **Hask**?

1. `data Foo = Foo Int` vs. `data Foo = Bar Int`?
2. `data Foo = Foo Int` vs. `data Bar = Bar Int`?
3. `data Foo = Foo Int | Bar String` vs. `Foo = Bar String | Foo Int`

Which of the following pairs `f, g :: Int -> Int` of Haskell functions do we consider to be equal as morphisms in **Hask**?

4. `f a = a` vs. `g a = a + 0`?
5. `f a = (a + 1)^2` vs. `g a = a * a + a + a + 1`?
6. `f a = f a` vs. `g a = g a`

◊

Example 1.74. Suppose we define `data Foo = MkFoo Foo`. We will show that there is a morphism of type `Foo -> Void`, and that there is no morphism of type `() -> Foo`. Here are two candidates, but we'll see that the latter is not a morphism:

```
f :: Foo -> Void
f (MkFoo x) = f x

g :: () -> Foo       -- well-defined Haskell, non-terminating
g () = MkFoo (g ())
```

As in the comment, the function `g` is well-defined Haskell, but it is non-terminating; it keeps calling itself forever. We believe there is no terminating morphism `() -> Foo`. On the other hand, `f` is also well-defined Haskell, and it is terminating for all inputs of type `Foo` since there are none! Thus `f` is a morphism in **Hask**.

We will eventually talk about Kleisli categories for monads. The reader is not expected to know that material now, but this is not in contradiction to what we're saying above. All of these morphisms and their composition laws can be written in terms of morphisms in **Hask**, and the `do` blocks or bind (>>=) syntax can be understood as just syntactic sugar.

1.4.6 Many objections

In a very nice blog article `http://blog.sigfpe.com/2009/10/what-category-do-haskell-types-and.html`, Dan Piponi discusses this issue and says that what we have called **Hask** "really isn't that interesting". It is interesting enough for us, and we'll continue to use it, but Piponi makes a good point. He says that Haskell can stand as the internal language of cartesian closed category, or perhaps a bicartesian closed category, or perhaps a bicartesian closed category with initial algebras for every finitary endofunctor. You don't have to know what that means, but basically he's saying that we should think of Haskell as telling us about the objects and morphisms in general (bi)cartesian closed categories.

Andre Baujer also wrote a very helpful article http://math.andrej.com/2016/08/06/hask-is-not-a-category/ where he says that there is no category **Hask**. We certainly have not proven that it is, though we believe we have circumvented his objections. Please let us know if we've missed anything!

We should also mention that much of what Haskell can do, e.g. work with infinite streams of natural numbers (final coalgebras for the endofunctor $S \mapsto \mathbb{N} \times S$), isn't captured in the above framework. We can certainly work with Haskell functions into and out of such types, we can compose them etc., but there are no closed terminating terms of such types, and so we are losing a lot of interesting ideas. This is the cost of working with a concrete categorical model.

So the point is that there is a category that one might mean when they say **Hask**—that of Haskell data types and terminating Haskell functions up to extensional equivalence—and it has the right initial and terminal objects, products, coproducts, etc. However it doesn't do everything you expect in Haskell, e.g. the dynamics of infinite streams.

So in the end, category theory is mainly just a useful metaphor for Haskell. We can look at a concrete model if we need to, but we miss lots of the interesting stuff. We can use category theoretic ideas to good end, without needing to make them 100% precise. This is something that we just have to get used to in applied category theory.

Chapter 2

Functors and type constructors

2.1 Introduction

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. A category is thus a network of relationships, and categories themselves form such a network: the category of categories. In fact, in a beautiful example of categorical thinking, 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 categories are known as *functors*, and 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.

These three structures will form the cornerstones of our toolbox for thinking about programming. Through this chapter we'll introduce more of the programming language Haskell, and hint at how functors and natural transformations find manifestations in type constructors and polymorphic functions.

2.2 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.

2.2.1 Definition of functor, and first examples

We begin with a slogan

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

A functor $F\colon \mathcal{C} \to \mathcal{D}$ takes each object $A \in \mathcal{C}$ to an object that we denote $F\,A$ in $\mathcal{D}$. Similarly, it takes each morphism $f\colon A \to B$ in $\mathcal{C}$ to a morphism $F\,f\colon F\,A \to F\,B$ in $\mathcal{D}$.

What does it mean that a functor is supposed to preserve 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

$$F\,\mathrm{id}_A = \mathrm{id}_{F\,A}$$

and composition to composition

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

Definition 2.1. A *functor* $F\colon \mathcal{C} \to \mathcal{D}$ consists of two constituents:

1. a function $F\colon\ \mathrm{Ob}\,\mathcal{C} \to \mathrm{Ob}\,\mathcal{D}$, and
2. for every A, B in $\mathrm{Ob}\,\mathcal{C}$, a function $F_{A,B}\colon \mathcal{C}(A,B) \to \mathcal{D}(F\,A, F\,B)$.

These constituents are subject to two constraints:

Preserves composition: for any $f\colon A \to B$, $g\colon B \to C$, the equation $F\,g \circ F\,f = F\,(g \circ f)$ holds.

Preserves identity: for any $A \in \mathrm{Ob}\,\mathcal{C}$, the equation $F\,\mathrm{id}_A = \mathrm{id}_{F\,A}$ holds.

Example 2.2. Given any category $\mathcal{C}$, we may define the *identity functor* $\mathrm{id}_{\mathcal{C}}\colon \mathcal{C} \to \mathcal{C}$. For the functor $\mathrm{id}_{\mathcal{C}}\colon\ \mathrm{Ob}\,\mathcal{C} \to \mathrm{Ob}\,\mathcal{C}$, we simply pick the identity function; so on objects, this functor sends each object A to itself. The same is true for morphisms: the identity functor $\mathrm{id}_{\mathcal{C}}$ sends each object to itself. In particular, this choice of mappings on objects and functors preserves composition and identity morphisms in $\mathcal{C}$; see Exercise 2.3.

Exercise 2.3. Check that $\mathrm{id}_{\mathcal{C}}$ really does preserve identities and composition. ◊

Example 2.4. Let $\mathcal{C}$ and $\mathcal{D}$ be categories. Given any object d in $\mathcal{D}$, we can define the *constant functor* $K_d\colon \mathcal{C} \to \mathcal{D}$ on d. This functor sends *every* of object $\mathcal{C}$ to $d \in \mathrm{Ob}\,\mathcal{D}$, and *every* morphism of $\mathcal{C}$ to the identity morphism on d.

Exercise 2.5. Show that the constant functor $K_d\colon \mathcal{C} \to \mathcal{D}$ obeys the two functor laws: preservation of composition and preservation of identities. ◊

Exercise 2.6. List all functors between the two-object categories $\mathbf{2}$, $\mathbf{Disc}(\underline{2})$, and $\mathbf{I}$ (as defined in Examples 1.27, 1.28, and 1.33). ◊

Definition 2.7 (Endofunctor)**.** Let $\mathcal{C}$ be a category. An *endofunctor* on $\mathcal{C}$ is a functor $F\colon \mathcal{C} \to \mathcal{C}$.

2.2.2 The category of categories

We have seen that for every category $\mathcal{C}$, there is an identity functor $\mathrm{id}_{\mathcal{C}}$. 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.

Suppose we have functors $F\colon \mathcal{C} \to \mathcal{D}$, and $G\colon \mathcal{D} \to \mathcal{E}$. We can define a new functor $G \circ F\colon \mathcal{C} \to \mathcal{E}$, as follows. First, we need to give a function $\mathrm{Ob}\,\mathcal{C} \to \mathrm{Ob}\,\mathcal{E}$. This is given by composing our functions on objects for F and G. That is, given $c \in \mathrm{Ob}\,\mathcal{C}$, we send it to $G(F(c))$. Similarly, given a morphism $f\colon c \to c'$ in $\mathcal{C}$, we send it to $G(F(f))$, which is a morphism $G(F(c)) \to G(F(c'))$ in $\mathcal{E}$. This preserves composition and identities because both F and G do.

Exercise 2.8. Show that if $F\colon \mathcal{C} \to \mathcal{D}$ and $G\colon \mathcal{D} \to \mathcal{E}$ are functors, then so is $G \circ F$, i.e. that it preserves identities and compositions. ◊

Definition 2.9. The *category of categories*, denoted **Cat**, has categories as objects, and functors between them as morphisms. Composition is given by composition of functors, and identities are the identity functors.

Exercise 2.10. Show that the category of categories is indeed a category, i.e. that it satisfies the unital law and the associative law. ◊

2.2.3 Functors and shapes

Remember our discussion about interpreting functions $f\colon a \to b$ 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.

The categorical analogue of a singleton set is the discrete category $\mathbf{1}$ on a one element set: we met this category in Example 1.26, and saw it was part of the family of discrete categories in Example 1.27. **1** has one object and no arrows except for the identity:

$$\mathbf{1} = \boxed{\begin{tikzcd} 1 \arrow[loop above, "\mathrm{id}_1"] \end{tikzcd}}$$

In analogy with the singleton set, a functor from **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 2.11. Let $\mathcal{C}$ be an arbitrary category.

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

Rather than interpreting a functor $F\colon \mathcal{C} \to \mathcal{D}$ as a model of $\mathcal{C}$ in $\mathcal{D}$, we could interpret it as a grouping of $\mathcal{C}$ 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 2.12. Let $F\colon \mathcal{C} \to \mathcal{D}$ be a functor and let $d \in \mathrm{Ob}\,\mathcal{D}$ be an object. Show that the objects and morphisms in $F^{-1}(d)$ form a category.[a] ◊

[a]One might imagine that since every object in $\mathcal{D}$ gives rise to a subcategory $F^{-1}(d)$ of $\mathcal{C}$, perhaps also every morphism $f\colon d \to d'$ in $\mathcal{D}$ somehow gives rise to a functor between the categories $F^{-1}(d)$ and $F^{-1}(d')$. However, this is not true; instead, it induces something called a *profunctor*.

Continuing with discrete categories, there is also a category $\mathbf{Disc}(\underline{2})$ with two objects $\mathrm{Ob}\,\mathbf{Disc}(\underline{2}) = \{1, 2\}$ and two identity morphisms id_1 and id_2. This is the categorical equivalent of a two-element set. A functor from $\mathbf{Disc}(\underline{2})$ to any category **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 is also interesting. It partitions objects in the source category into two groups, the ones mapped to 1 and the ones mapped to 2. In this respect $\mathbf{Disc}(\underline{2})$ is the ying and yang of category theory. 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 **C** to $\mathbf{Disc}(\underline{2})$, if it exists, must partition **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 ar: $1 \to 2$ to the two-object category, to arrive at the category **2** (see Example 1.28).

$$\mathbf{2} = \boxed{\begin{tikzcd} \mathrm{id}_1 \arrow[r, phantom] & 1 \arrow[loop left] \arrow[r, "\mathrm{ar}"'] & 2 \arrow[loop right, "\mathrm{id}_2"] \end{tikzcd}}$$

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

Mappings out of **C** into **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 **2**.

Exercise 2.13. How many functors are there from **Set** to **2**? Write them down. ◊

Going further, we can add another arrow to our two-object category, going in the opposite direction. The result is the *walking isomorphism* **I** (Example 1.33):

$$\mathbf{I} = \boxed{\begin{tikzcd} 1 \arrow[loop left, "\mathrm{id}_1"] \arrow[r, bend left, "f"] & 2 \arrow[l, bend left, "f^{-1}"] \arrow[loop right, "\mathrm{id}_2"] \end{tikzcd}}$$

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

Exercise 2.14. Given a category $\mathcal{C}$, show that isomorphisms in $\mathcal{C}$ are in one-to-one correspondence with functors $\mathbf{I} \to \mathcal{C}$. Hint: A functor preserves composition and identity. ◊

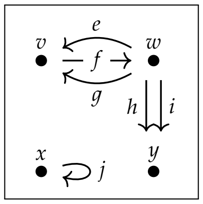
Exercise 2.15. What kind of sorting does the mapping out of $\mathcal{C}$ to the walking isomorphism category define? ◊

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.

$$\mathbf{Gr} = \boxed{\begin{tikzcd} \overset{\mathrm{Edge}}{\bullet} \arrow[r, shift left, "\mathrm{src}"] \arrow[r, shift right, "\mathrm{tgt}"'] & \overset{\mathrm{Vert}}{\bullet} \end{tikzcd}}$$

Consider a functor $G\colon \mathbf{Gr} \to \mathbf{Set}$; we will see that G represents a graph. First, it chooses two sets, $G(\mathrm{Edge})$ and $G(\mathrm{Vert})$ 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 $e \in G(\mathrm{Edge})$ to a vertex: the source vertex $G(\mathrm{src})(e)$ and the target vertex $G(\mathrm{tgt})(e)$ of e.

Here we depict an example of such a functor G



G(Edge)	G(src)	G(tgt)
e	w	v
f	v	w
g	w	v
h	w	y
i	w	y
j	x	x

G(Vert)
v
w
x
y

Here, elements of the sets G(Edge) and G(Vert) are written in the first columns of the respective tables. The results of applying functions $G(\text{src}), G(\text{tgt})\colon G(\text{Edge}) \to G(\text{Vert})$ to each edge $e \in G(\text{Edge})$ are written in the respective columns. The whole data structure that $G\colon \mathbf{Gr} \to \mathbf{Set}$ is thus represented in these two tables, but can be drawn as a graph, as shown. Elements of G(Vert) are drawn as vertices and elements of G(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 $f\colon a \to b$ 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 **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 receptacles". This viewpoint will not be as useful to us, but it can still be interesting to consider.

2.3 Functors in Haskell

Let's return to our central metaphor: the types and functions of Haskell form a category **Hask**. Since this category is the only category at our disposal, under our metaphor, we are limited to defining endofunctors on this category: functors **Hask** $\to$ **Hask**. From the point of view of working with functors in Haskell then, a Haskell functor maps types to types and functions to functions.

When dealing with endofunctors, it's important to keep in mind the difference between functors and morphisms. A functor *acts* on objects—given one object it returns another—while a morphism is an arrow going *between* objects. An endofunctor in **Set**, for instance, could assign a set of oranges to a set of apples. It would not map individual apples to individual oranges. That's what a function between those sets would do.

Example 2.16. There is a functor **Set** $\to$ **Set** sending every set X to $\varnothing$; it sends every morphism to the identity morphism on $\varnothing$. But there is almost never a *function* from an arbitrary set X to $\varnothing$.

2.3.1 Interlude: polymorphic lambda calculus

Let's start with a bit of theory. It's possible to extend lambda calculus to define functions on types. Formally, this is called polymorphic or second-order lambda calculus, or system F. Lambda abstractions over types are written using uppercase lambdas.

For instance, the implementation of our polymorphic identity function can be written as a double lambda abstraction:

$$\mathrm{id} = \Lambda a\,.\,(\lambda(x : a)\,.\,x)$$

The outer capital-Lambda takes any type a and returns a function, namely the inner lambda, $\lambda x\colon a\,.\,x$, which takes a value x of type a and returns it back. A function is a value itself, so the outer (capital) lambda turns types, here represented by the type variable a, to values of some other type that depends on a; here, the type of the inner lambda is $a \to a$.

The type of the whole expression is:

$$\mathrm{id}\colon \forall a.\, a \to a$$

The combination of the two translates to Haskell as:

```
id :: forall a. a -> a
id = \x -> x
```

Notice that you'll have to put the pragma

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

at the top of the file to allow the syntax with explicit (and, in this case, redundant) `forall`.

Despite similarities, Haskell syntax doesn't support explicit type-level abstraction: there is no Λ in Haskell, because type inference becomes undecidable in the presence of type-level lambdas. This makes working with types somewhat harder than working with values: you can't actually "plug a type" into a Λ. As a result there have been many ad-hoc extensions to Haskell, like type families, functional dependencies, and defunctionalization, to mention just a few.

Until recently, there was no syntax for the application of type-level functions to concrete types. The type-level arguments are normally inferred by the compiler and passed silently. This is why it's okay to call

```
Prelude> id "Hi, it's me"
"Hi, it's me"
```

without explicitly informing the compiler that the type argument to be passed to `id` is `String`. However, a recent addition to Haskell allows you to pass type arguments explicitly. All you need to do is turn on the language pragma `TypeApplications`. Here's the invocation of `id` that picks the specific version that only works for integers:

```haskell
{-# language TypeApplications #-}
idInt = id @Int
```

Think of this as explicitly passing the argument `Int` to a function that operates on types. Now if you write `id @int 3` you get `3`, but if you write `id @Int "hi"` you get an error.

2.3.2 Type constructors

In Haskell, functions on types are defined using a special syntax that extends the Haskell syntax for operating on terms of a given type. For instance, the identity type-function looks like data definition that is parameterized by a type parameter `a`:

```haskell
data Id a = MkId a
```

Here, `Id` is called a *type constructor* since, for any type `a`, it constructs a new type called `Id` a. `MkId` is called a *data constructor* because it is used to construct data of the type `Id` a for any `a`. Thus for any type `a`, its type is:

```haskell
MkId :: a -> Id 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 `MkId`) as keywords that indicate the type of the term to follow: "whenever I see `MkId x`, I check what type `x` has, say `Int`, and I'll know that `MkId a` has type `Id Int`."

This is how the constructor may be used to construct a value of the type `Id Int`:

```haskell
i42 :: Id Int
i42 =  MkId 42
```

The first line declares the type of the variable[1] `i42` as the type obtained by applying the type constructor `Id` to `Int`. The second line declares `i42` to have a specific value, `42`.

[1]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.

Once a value of the type `Id a` is constructed, it is immutable. This makes Haskell a purely functional language, as opposed to imperative languages in which mutation is allowed. 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 `Id a` can always be deconstructed. This deconstruction is called *pattern matching*. Here's how it's done:

```
unId :: Id a -> a
unId (MkId x) = x
```

The pattern that is matched here is `(MkId x)` (patterns are parenthesized). It names the data constructor `MkId` and the value `x`. This is the value that was passed to it during construction. When applied to the above example,

```
unId id42
```

will produce the integer `42`.

Example 2.17. Here are some other type constructors:

```
data Double a  = MkDoub a a
data WString a = WStrng a String
data Unit a    = U
```

In Exercise 2.23 we'll see that these can all be made into functors.

2.3.3 Back to functors

To define a functor in category theory, we need to say what it does on objects and what it does on morphisms (and then check that certain laws hold, but this doesn't show up in programming). We have just talked about polymorphic type constructors. 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 $F\colon \mathcal{C} \to \mathcal{C}$ does on morphisms, we might say that it "lifts f to $F(f)$".

Example 2.18 (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 2.19 (Constant functor). We can repeat the same procedure for the constant functor:

$$C_{\mathrm{Int}} = \Lambda a.\, \mathrm{Int}$$

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 2.20; in fact they are "naturally isomorphic" in the sense of natural transformations, which we'll come to later.

Exercise 2.20. 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 2.21. 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 2.22 (Reader functor). Here we specify a functor `FromInt` that takes a type `a` and returns the function type `Int -> a`.

```
data FromInt a = MkFromInt (Int -> a)
```

Here's how the functor lifts functions:

```
mapFromInt :: (a -> b) -> (FromInt a -> FromInt b)
mapFromInt f (MkFromInt g) =  MkFromInt (f . g)
```

In Haskell, this is called a `Reader` functor.

Exercise 2.23. Give each of the type constructors in Example 2.17 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) ◊

2.3.4 Type classes

You might have noticed—especially in **??**—that there is a strong similarity in the signature for how functors lift morphisms. If we replace the polymorphic type constructors with a type constructor variable `f`, they all look like this:

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

However, if you try to define `fmap` for two different type constructors, say `fmap :: (a -> b) -> (Id a -> I` and `fmap :: (a -> b) -> (FromInt a -> FromInt b)`, the compiler will protest that you are trying to implement the same function twice.

Using the same name for two different functions is called *overloading* and is a very useful feature. 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, overloading is implemented using *type classes*.

A type class defines the names and type signatures for overloaded functions. The compiler then figures out which implementations to use by figuring out what types are involved, i.e. using *type inference*. We'll talk more about type classes later. For now, we'll just show you how to use type classes to work with functors.

A polymorphic type constructor really becomes an instance of what Haskell calls a `Functor` when it is equipped with the additional function that tells us how to lift functions. The name of that function is `fmap`, and the following Haskell code declares that in order to say that something is of type `Functor`, you must say how to implement `fmap`:

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

Example 2.24. Here we show that `Id`, `CInt`, and `FromInt` from Examples 2.18, 2.19, and 2.22 are all instances of type `Functor` by instantiating `fmap`.

```
instance Functor Id where
  fmap f (MkId x) = MkId (f x)

instance Functor CInt where
  fmap f (MkC x) =  MkC x

instance Functor FromInt where
  fmap f (MkFromInt g) =  MkFromInt (f . g)
```

Haskell Note 2.25 (Double usage of constructor names). It's worth noting that it is common practice to reuse the names of type constructors as data constructors, since they live in different name spaces. For instance, we could define:

```
data Id a = Id a
```

without the risk of confusing the compiler.

Haskell Note 2.26 (Record syntax). Often the function for extracting the contents is baked into the definition of the data structure by using *record syntax* (see product types).

```
data Id a = Id { unId :: a }
```

On page 45 we wrote `unId` as a standalone function, whereas here we bake it in. Here's how you construct and deconstruct terms of this type

```
Prelude> x = Id {unId = 'c'}
Prelude> :t x
x :: Id Char
Prelude> unId x
'c'
```

2.3.5 More interesting functors

The examples of functors seen so far are not so useful in practical programming. This is because we haven't developed enough Haskell syntax. To give you a better taste of functors in Haskell, let's introduce the most common and useful example: the list. Since lists are ubiquitous in Haskell, they have a very terse syntax. The type constructor is an *outfix* operator `[]`. It means that, if you want to define the type for a list of `a`, you write `[a]`. A list of integers has the type `[Int]`.

The list has two data constructors. One needs no input and creates an empty list and is called `[]` (not to be confused with the type constructor of the same name). The other needs a value and a list and creates a list: the value becomes the new list's head and the old list becomes the new list's tail; it is called `:` (the colon) and is used in infix notation. When you want to define a function on lists, you usually pattern match on both constructors.

Haskell Note 2.27. What happens when you don't pattern match on all constructors? This is not an error, unless you tell the compiler to treat it as such. But the resulting function will not be total. It will be a partial function which may fail with a fatal error (sudden program termination).

The only possible reason to use partial functions—other than carelessness—is to improve performance. Runtime tests take precious time, so if the programmer knows with 100% certainty that the test will always be positive (for instance, because it has already been performed before), it may be better to avoid them. There are even partial functions in the `Prelude`: `head` and `tail` are common examples.

If it weren't already defined in the `Prelude`, here's how you would show that list is an instance of type `Functor`:

```
instance Functor [] where                --[] is the type constructor
  fmap f [] = []                         --These []'s are empty lists
  fmap f (x : xs) = f x : fmap f xs      --Recursion!
```

Let's analyze it line by line. The first line

```
instance Functor [] where
```

specifies that the type constructor `[]` can be made into a `Functor`. The second line

```
  fmap f [] = []
```

provides the implementation of `fmap` for the empty list `[]`. The third line is more interesting:

```
    fmap f (x : xs) = f x : fmap f xs
```

It pattern-matches on the second constructor by splitting the list into its head `x` and tail `xs`. Notice that this pattern is matched only when the list is non-empty. It then applies the function `f` to both, the head and the tail of the list, and constructs a new list using the `:` constructor. The application of `f` to the head is straightforward, because the head is of the right type for `f` to act on it. But the tail is a list, so we have to invoke `fmap` again to apply `f` to it.

We are calling `fmap` inside the definition of `fmap`. Such recursive definitions are very common in Haskell, which doesn't have any looping constructs (loops require the mutation of the iteration index). This recursion is well founded, meaning that its arguments gets smaller (the tail is shorter that the list) on every iteration, and eventually we hit the empty list case. But, in general, there is no guarantee that an arbitrary recursive function will terminate.

We could create values of the list type by applying the two constructors. Let's do it in GHCi:

```
Prelude> let ls0 = []
Prelude> :t ls0
ls0 :: [a]
Prelude> let ls1 = '!' : ls0
Prelude> :t ls1
ls1 :: [Char]
Prelude> let ls2 = 'i' : ls1
Prelude> let ls3 = 'H' : ls2
Prelude> ls3
"Hi!"
```

Here's what happened, in order:

- Construct an empty list `ls0`
- Ask the compiler what type it is using the `GHCi` command `:t`. Since the list is empty, all the compiler can tell is that it's a list of some unspecified `a`
- Construct a list using the infix constructor `:`, passing it a character literal `'!'` and an (empty) list for tail. A character literal is enclosed in single quotes
- This time the compiler correctly deduces that `ls1` is a list of characters, `[Char]`
- Construct a list from the character `'i'` and the tail `ls1`
- Prepend `'H'` to `ls2`
- Display the list `ls3`. The list is displayed as a string `"Hi!"` because, in Haskell, strings are lists of characters.

This is rather tedious, so Haskell provides syntactic sugar for both lists and strings. Here's the same thing using list literals:

```
    let ls3 = ['H', 'i', '!']
```

or string literals

```
    let ls3 = "Hi!"
```

Example 2.28 (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 $f\colon A \to B$, 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 something that Haskell will *think is a functor* but is actually not a functor.

```
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 2.29. For each of the following type constructors, define two versions of `fmap`, one of which defines a functor 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)` ◊

Example 2.30 (Maybe). When one knows that a certain function $f\colon a \to b$ 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.

```
data Maybe a = Nothing | Just a
instance Functor Maybe where
 fmap f Nothing  = Nothing
 fmap f (Just a) = Just (f a)
```

Exercise 2.31.

1. Create a list of four integers 0, 1, 2, 3 using explicit constructors and list literals.
2. Create a never-ending list of 5's as a function `fives :: () -> [Int]`.

◊

There is additional syntactic sugar for creating lists of consecutive integers. The list `[0,1,2,3]` from Exercise 2.31 can be created with

```
    let lst = [0..3]
```

Exercise 2.32. Implement a function `consec :: Int -> [Int]` that sends any $n \geq 0$ to `[0,1,...,n]` and which can do anything you want (even not terminate) on $n < 0$.

◊

Example 2.33 (Using functoriality of list). Recall that we implemented `fmap` for lists on page 49. We are now in a position to apply it. For instance, if we want to add one to every element of a list of integers, we write:

```
Prelude> fmap (\n -> n + 1) [0..5]
[1,2,3,4,5,6]
```

The lambda expression defining a function (`\n -> n + 1`) must be enclosed in parentheses when it's passed to another function (`fmap`, in this case).

Exercise 2.34. In each of the following use `fmap` for the list functor.

1. Give a function `[String] -> [Int]` that applies `length` to every string in a list.
2. Give a function `[Int] -> [Bool]` that asks if each element of a list is even or not.
3. Is it possible to use `fmap` in a way that changes the length of a list? Why or why not?

◊

2.3.6 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 2.28 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 -> (c -> d)` and those of the type `(b, c) -> d`.

Functors $F\colon \mathcal{C} \to \mathcal{D}$ 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 $F_{c_1,c_2}\colon \mathcal{C}(c_1, c_2) \to \mathcal{D}(Fc_1, Fc_2)$. 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 $F_{c_1,c_2}\colon c_2^{c_1} \to (Fc_2)^{Fc_1}$.

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 -> b` to a term of type `f a -> 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 $F\colon \mathcal{C} \to \mathcal{C}$ is called *strong* if, for every c, c' there is a natural morphism $c \times F(c') \to F(c \times c')$, called the *strength*. By uncurrying, the strength is equivalent to a map $c \to F(c \times c')^{F(c')}$, and so if F is enriched then we just need a function $c \to (c' \times c)^{c'}$, and we get that by currying the identity. Conversely, given a strength we need a function $c_2^{c_1} \to (Fc_2)^{Fc_1}$. This is equivalent to a function $c_2^{c_1} \times Fc_1 \to Fc_2$; to obtain it we just use the composite $c_2^{c_1} \times Fc_1 \to F(c_2^{c_1} \times c_1) \to F(c_2)$, where the first map is the strength and the second is application of F on the evaluation morphism $c_2^{c_1} \times c_1 \to c_2$.

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)`.

2.4 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.

2.4.1 Definition

Natural transformations are maps between parallel functors; you can draw one like this:

$$\begin{tikzcd} \mathcal{C} \arrow[r, bend left, "F"{name=U}] \arrow[r, bend right, "G"'{name=D}] & \mathcal{D} \arrow[Rightarrow, from=U, to=D, "\alpha"] \end{tikzcd}$$

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 $\mathcal{C}$. Here's a play.

(Francine and Grace meet at a party.)
Francine: Nice to meet you Grace, what do you do?
Grace: I'm a functor from $\mathcal{C}$ to $\mathcal{D}$; you tell me an object or morphism in $\mathcal{C}$ and I'll give you one in $\mathcal{D}$. And of course I'm licensed and bonded, meaning I respect identities and stuff.
Francine: Whoa, neat, I'm a functor too! Respecting the laws is key. But tell me, what do you do with the object c_5? I send it to $F(c_5) = d_2$.
Grace: Oh weird, I send c_5 to d_{12}. I can see we do things pretty differently. I bet there's someone at this party that can translate between us.
Alfreda: Hi, I can help! I've seen you both work, and I've noticed a great way of connecting your outputs. Let's see; we've got $F(c_5) = d_2$ and $G(c_5) = d_{12}$; that's easy, I'll use the map $g_8 \colon d_2 \to d_{12}$ in $\mathcal{D}$. What else you got?
(And the three talked through the night and got it all mapped out, naturally.)

Definition 2.35 (Natural transformation)**.** Let $\mathcal{C}$ and $\mathcal{D}$ be categories, and let $F, G \colon \mathcal{C} \to \mathcal{D}$ be functors. A *natural transformation α from $\mathcal{C}$ to $\mathcal{D}$*, denoted $\alpha \colon F \Rightarrow G$, consists of a morphism $\alpha_c \colon F(c) \to G(c)$ for each $c \in \mathrm{Ob}(\mathcal{C})$, called the *c-component*, and these components are subject to a condition called *naturality*. Namely, for each $f \colon c_1 \to c_2$ in

$\mathcal{C}$ the following square needs to commute in $\mathcal{D}$:

$$\begin{tikzcd}
F(c_1) \arrow[r, "F(f)"] \arrow[d, "\alpha_{c_1}"'] & F(c_2) \arrow[d, "\alpha_{c_2}"] \\
G(c_1) \arrow[r, "G(f)"'] & G(c_2)
\end{tikzcd} \tag{2.36}$$

2.4.2 Natural transformations in Haskell

A natural transformation is a family of morphisms, one per object. Haskell will never check the naturality condition Eq. (2.36), 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 and 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 2.18 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:

```haskell
{-# 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:

```haskell
  type (~>) f g = forall a. f a -> g a
```

which says that `(~>)` 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 2.37. Here is our infix operator `~>` in action. We first define the type of all natural transformations between `Id` and `CInt`:

```haskell
type IdToConst = Id ~> CInt
```

Then we can give an example term of this type:

```haskell
seven :: IdToConst String
seven _ = MkC 7
```

In fact all examples of `IdToConst` will be like this; see Exercise 2.38 for a set-theoretic analogue.

Exercise 2.38. Consider the identity endofunctor $\mathrm{id}\colon \mathbf{Set} \to \mathbf{Set}$ and the constant endofunctor $C_{\mathbb{N}}\colon \mathbf{Set} \to \mathbf{Set}$ where $C_{\mathbb{N}}(X) = \mathbb{N}$ for all $X \in \mathrm{Ob}(\mathbf{Set})$. Show that every natural transformation

$$\begin{tikzcd}\mathbf{Set} \arrow[r, bend left, "\mathrm{id}"{name=U}] \arrow[r, bend right, "C_{\mathbb{N}}"'{name=D}] & \mathbf{Set} \arrow[from=U, to=D, phantom, "\Downarrow\alpha"]\end{tikzcd}$$

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 $\alpha_X\colon X \to \mathbb{N}$ is the constant function at n. ◊

Haskell Note 2.39. 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. (2.36)) 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 $\alpha\colon F \Rightarrow G$ in Definition 2.35. In principle, one could pick a completely different morphism $\alpha_a\colon Fa \to Ga$ 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 2.40 (Lists to Maybes). In Example 2.30 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 2.41. Implement a natural transformation

```
  uncons :: [a] -> Maybe (a, [a])
```

◊

Haskell Note 2.42. 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
```

Chapter 3

Universal constructions and the algebra of types

3.1 Constructing datatypes

We have seen that types play a fundamental role in thinking about programming from a categorical point of view. Often a programming language 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 our language more expressive, we will introduce ways of making new types from old, such as the operation of product, sum, and exponential for 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 $\diamondsuit$, 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 ($2\diamondsuit$), or the ace of spades ($A\spadesuit$). A type constructed from others in the way `Card` was 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: `Card` = `Rank` $\times$ `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 `(Rank, 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 `(String, Int)` of (name, age) pairs, the type `(Int, Int)` representing locations on a two-dimensional grid, or the type `(Int,Int,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, one might want to construct a type `TVChannel` of TV-on-demand channels, which channels drawn from either Netflix or Hulu. Using the plus sign from arithmetic to represent a sum type, we might then write `TVChannel` = `Netflix` + `Hulu`.

Sum types also come with associated functions. For example, one might want to consider a Netflix channel just as some TV channel (and not specifically a Netflix channel); for this, we want a function `netflixAsChannel:: Netflix -> TVChannel`.

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 1.12 for why. Passing functions as data can be very useful in programming.

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.

But in order to explain universal properties, we'll begin with a more abstract lesson on categorical thinking. Namely, we'll a beautiful theorem, known as the Yoneda lemma, which sits at the core of the subject; roughly, it 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.

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 the unit type, 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.

These universal constructions can be further generalized in category theory using limits and colimits, and we'll introduce this general concept, together with a few more examples. An important example is that of a pullback, which can be modeled using dependent types. As we'll see, these aren't implemented in Haskell, but are in other languages like Idris. We'll also discuss directed limits and directed colimits, which do come up in Haskell in the form of reucrsion, as we'll see when we get to initial algebras and terminal coalgebras in Chapter 4.

Another viewpoint on universal constructions is given by the notion of adjunction. We'll discuss adjunctions, e.g. in the case of the *currying adjunction*, which will give us function types. This leads us to the notion of a cartesian closed category, which we briefly discusse din Section 2.3.6. It provides models of the simply-typed lambda calculus, an important way of thinking about computable functions.

3.2 The Yoneda embedding

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 3.1. Let a be an object in a category $\mathcal{C}$. We may define a functor

$$y_a \colon \mathcal{C}^{\text{op}} \to \textbf{Set}$$

sending each object x of $\mathcal{C}$ to the set $\mathcal{C}(x, a)$, and sending each morphism $m \colon x \to y$ to the function $y_a(m) \colon \mathcal{C}(y, a) \to \mathcal{C}(x, a)$ given by sending the element $f \colon y \to a$ to the element $(m \fatsemi f) \colon x \to a$.

We call y_a the *functor represented by* a.

Example 3.2 (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 a?" for every x. For example, consider the poset

$$\mathcal{TC} := \begin{tikzcd} \boxed{\text{Eq}} & \\ \boxed{\text{Ord}} \arrow[u, "\text{extends}"] & \boxed{\text{Num}} \\ \boxed{\text{Real}} \arrow[u, "\text{extends}"] & \boxed{\text{Fractional}} \arrow[u, "\text{extends}"'] \arrow[ul, "\text{extends}"'] \end{tikzcd}$$

$$\begin{aligned} y_{\text{Ord}}(\text{Eq}) &= \varnothing \\ y_{\text{Ord}}(\text{Ord}) &= \{\text{extends}\} \\ y_{\text{Ord}}(\text{Num}) &= \varnothing \\ y_{\text{Ord}}(\text{Real}) &= \{\text{extends}\} \\ y_{\text{Ord}}(\text{Fractional}) &= \{\text{extends}\} \end{aligned}$$

from Example 1.42. The functor $y_{\text{Ord}} \colon \mathcal{TC}^{\text{op}} \to \textbf{Set}$ is the table shown right above.

Example 3.3. Let's think about the category **Set**. We'll write $\underline{1} = \{*\}$ for some set with one element. What is the functor represented by 1? First, it is a functor $y_1 \colon \mathbf{Set}^{\mathrm{op}} \to \mathbf{Set}$. Where does it send a set X? Well, what are the functions from $X \to 1$? Let's suppose we have a function $f \colon X \to 1$. We have to send every element of $x \in X$ to some element $f(x)$ of 1. But there is only one element of 1; its name is $*$. So $f(x)$ must equal $*$. Thus there is only one function from X to 1, and hence y_1 sends every set to a one element set. We recognize it as the constant functor.

Exercise 3.4. Consider the functor $y_{\mathbb{B}} \colon \mathbf{Set}^{\mathrm{op}} \to \mathbf{Set}$, where $\mathbb{B} = \{\mathtt{true}, \mathtt{false}\}$.

1. What is $y_{\mathbb{B}}(\{1,2,3\})$?
2. Consider the function $f \colon \{a,b,c\} \to \{1,2,3\}$ given by $f(a) = f(b) = 1$ and $f(c) = 2$. Write out $y_{\mathbb{B}}(f)$ as a function $y_{\mathbb{B}}(\{1,2,3\}) \to y_{\mathbb{B}}(\{a,b,c\})$.
3. Say how you can think of an element $y_{\mathbb{B}}(X)$ as a subset of X for each set X.
4. Given a function $f \colon W \to X$, what does $y_{\mathbb{B}}(f)$ do to subsets, given the perspective you came up with in part 3. ◊

Exercise 3.5. We said that the functor represented by a is a functor $y_a \colon \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$. Prove that it does indeed obey the conditions in the definition of functor, Definition 2.1. ◊

So for any category $\mathcal{C}$ and 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 3.6 (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 $y_b \colon \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$, then a is isomorphic to b.

Exercise 3.7. Suppose we have a natural transformation

$$\begin{tikzcd} \mathcal{C} \arrow[r, bend left, "y_a"{name=U}] \arrow[r, bend right, "y_b"'{name=D}] & \mathbf{Set} \arrow[from=U, to=D, phantom, "\Downarrow\alpha"] \end{tikzcd}$$

Use it to find a morphism $a \to b$. Hint: find a special element of $y_a(a)$ and plug it in to the component α_a. ◊

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 $\mathcal{C}$ together 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 $m\colon x \to x'$, I have to tell me the function $y_a(m)\colon \mathcal{C}(x', a) \to \mathcal{C}(x, a)$ between those abstract sets.

The Yoneda lemma says that you can always win this game; see Exercise 3.8.

Let's think about how I might win in a poset. Suppose you pick some element a of a poset P. Then by naming elements x of P, I can find out if x was less than a or not (see Example 3.2). But $a \le a$, and in fact a is a maximal element of P less than a. So if I name enough elements until I am confident I know a maximal element of P less than a, I can win the game.

Another helpful example is in the category **Set**. In fact, in this setting I can win the game just by naming one object. Which object? The object $\underline{1} \coloneqq \{1\}$. 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 $f\colon \underline{1} \to a$ are the same as elements of X. 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 **Set**, we only have sets (objects) and functions (morphisms); we can't say the word element. But the above strategy shows that we don't need to, as long as we know the object $\underline{1}$: an element of X is the same thing as a function $\underline{1} \to X$. We call a function $\underline{1} \to X$ 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 $f\colon X \to Y$, and want to figure out $f(x) \in Y$. The global element corresponding to $f(x)$ is simply the composite of $f\colon X \to Y$ with the global element $x\colon \underline{1} \to X$ corresponding to x.

$$\begin{tikzcd} & \underline{1} \arrow[dl, "x"'] \arrow[dr, "y"] & \\ X \arrow[rr, "f"'] & & Y \end{tikzcd}$$

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 $C \to X$. 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) defined by its generalized elements.

Exercise 3.8 (Challenge). Give a strategy for winning the above "Yoneda" game in a general category $\mathcal{C}$. (You can assume $\mathcal{C}$ has finitely many objects and morphisms if you want.) ◊

We've used the symbol C for the shape of a generalized element in a nod to how it is useful in programming: the shape describes the "context". It's from this perspective (known as a point-free perspective) that we'll discuss constructing algebraic data types.

3.3 Zero and one: some first universal constructions

3.3.1 Terminal objects

Let's begin with one of the simplest examples of a universal object in a category: the notion of terminal object.

Definition 3.9. 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 $!^a : a \to 1$.

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.

Exercise 3.10. What does the universality of 1 tell us about morphisms from 1 to 1? ◊

Exercise 3.11. Which of the following categories has a terminal object?

1.
$$\begin{tikzcd} \overset{1}{\bullet} \arrow[r] & \overset{2}{\bullet} \arrow[r] & \overset{3}{\bullet} \end{tikzcd}$$

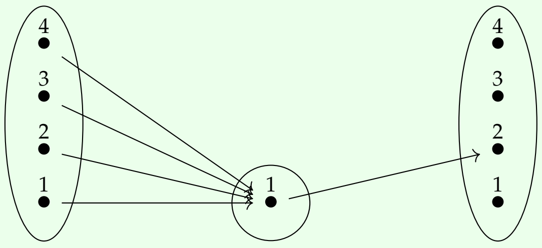
2.
$$\begin{tikzcd} \overset{1}{\bullet} \arrow[r, bend left, "f"] & \overset{2}{\bullet} \arrow[l, bend left, "g"] \end{tikzcd}$$
$$f \fatsemi g = \mathrm{id}_1$$
$$g \fatsemi f = \mathrm{id}_2$$

3.
$$\begin{tikzcd} & & \overset{3}{\bullet} \\ \overset{1}{\bullet} \arrow[r] & \overset{2}{\bullet} \arrow[ur] \arrow[dr] & \\ & & \overset{3'}{\bullet} \end{tikzcd}$$

◊

Example 3.12 (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 $f: X \to \{57\}$ is to define $f(x) = 57$ for all $x \in X$.

Example 3.13 (Constant morphisms). Suppose that $\mathcal{C}$ has a terminal object 1, and for any object C, let $!^C : C \to 1$ denote the unique morphism. We say a function $f: C \to D$ is *constant* if there exists a morphism $d: 1 \to D$ such that $f = !^C \fatsemi d$.



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

Exercise 3.14.
1. Name a poset with a terminal element.
2. Name a poset without a terminal element. ◊

Example 3.15. 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 3.16. 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 3.17 (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 $x \to y$? How about $y \to x$? Can you name a morphism $x \to x$? How many other morphisms are there $x \to x$?) ◊

Definition 3.18. Let $\mathcal{C}$ be a category with a terminal object 1. A *global element* of an object a is a morphism $1 \to a$.

Exercise 3.19. 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 3.20. Recall the category **Cat** from Definition 2.9. Show that the category **1** is the terminal category in **Cat**. ◊

The unit type In Haskell, we model the terminal object using the unit type, defined in the Haskell prelude. Its definition is:

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

This definition contains a pun. The symbol `()` (empty tuple) is being used in two, distinct ways: first as the name of a type, and then 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).

Exercise 3.21. For all types `a`, define a function

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

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

3.3.2 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 3.22. 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 $!_a : 0 \to a$.

Example 3.23. In a poset, an initial object is a least element.

Exercise 3.24. In **Set**, the initial object is the empty set, $\varnothing$. Why? (Since the empty set has zero elements, we often write 0 for an initial object in a category.) ◊

Exercise 3.25. Someone tells you an initial object in $\mathcal{C}$ is just a terminal object in $\mathcal{C}^{\mathrm{op}}$. What do they mean? Are they correct? ◊

Exercise 3.26. 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 3.27. 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.) ◊

The void type In Haskell, we model the empty set using the void type. The syntax for defining the type void is simple.

```
data Void
```

There is a correspondence, called the Curry-Howard correspondence, saying that we can think of each type T as being like the logical statement "T has a universal element", i.e. T is *inhabited*. Then universal elements of type T are then the proofs of the corresponding logical statement, that T is inhabited. A morphism $f\colon T \to U$ takes every proof that T is inhabited to a proof that U is inhabited.

Now in logic, the Latin phrase *ex falso quodlibet* ("from falsehood, anything follows") refers to the principle that if a contradiction, or *false*, can be proved, 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: if you have a proof of `Void`, you have a proof of anything. For fun, we'll refer the unique morphism from `Void` to any type `a` as `exFalso :: Void -> a`.

The type `Void` has a unique function to any other type `a` as follows:

```
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 `exFalso` will be executed. For that, one would have to provide it with an argument: a term of type `Void`, and no such term exists!

Another syntax is the case syntax.

```
absurd a = case a of {}
```

3.4 Products and pairs

So far we've considered terminal objects, which a mapping-in property, and initial objects, which have a mappingout 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 $f\colon A \to X \times Y$ from some set A to a product, it is enough to define a function $f_X\colon A \to X$ and $f_Y\colon A \to Y$. Then we can let $f(a)$ be the pair $(f_X(a), f_Y(a)) \in X \times Y$. That is, a morphism $A \to X \times Y$ in **Set** is the same as a pair of morphisms $A \to X$ and $A \to Y$. So products are about pairs, but they're about pairs of *morphisms*, rather than elements.

Definition 3.28. 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 $\pi_1\colon x \times y \to x$ and $\pi_2\colon x \times y \to y$, with the following *universal property*: For any other such three-things, i.e. for any object a and morphisms $f\colon a \to x$ and $g\colon a \to y$, there is a unique morphism $h\colon a \to x \times y$ such that the following diagram commutes:

$$\begin{tikzcd}
 & a \arrow[d, dashed, "h"] \arrow[ddl, bend right, "f"'] \arrow[ddr, bend left, "g"] & \\
 & x \times y \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
x & & y
\end{tikzcd}$$

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 3.29. 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 3.30.

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. ◊

Example 3.31. The product of two sets X and Y in **Set** is exactly the cartesian product $X \times Y$, with the projection maps, well, the projection functions $\pi_1\colon X \times Y \to X$ and $\pi_2\colon X \times Y \to Y$ defined respectively by $\pi_1(x, y) = x$ and $\pi_2(x, y) = y$.

To prove this, given functions $f\colon A \to X$ and $g\colon A \to Y$, we must show there is a unique function $h\colon A \to X \times Y$ such that $h \mathbin{⨟} \pi_1 = f$ and $h \mathbin{⨟} \pi_2 = g$. 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 $h(a) = (f(a), g(a))$.

Exercise 3.32. Let's work out an explicit example in the category **Set**. Choose sets X, Y, Z and functions $f\colon X \to Y$ and $g\colon X \to Z$ What is $h = \langle f, g\rangle$? Do you think this is a good notation? ◊

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 3.17.

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 $h\colon a \to x \times y$. The product can be used to *decompose* this problem into two simpler problems: one function $a \to x$ and another $a \to y$. Indeed, the universal property of the product implies that we have a one-to-one correspondence (i.e. an isomorphism of sets)

$$\mathcal{C}(a, x \times y) \cong \mathcal{C}(a, x) \times \mathcal{C}(a, y). \tag{3.33}$$

This means that in order to specify any morphism in the set on the left (i.e. one morphism $a \to x \times y$), its enough to name its corresponding element in the set on the right (i.e. a pair consisting of a morphism $a \to x$ and a morphism $a \to y$). 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 3.34. 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. (3.33):

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

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.

$$\begin{tikzcd}
 & 1 \arrow[ldd, bend right, "x"'] \arrow[d, dashed, "{(x,y)}"] \arrow[rdd, bend left, "y"] & \\
 & a \times b \arrow[ld, "\pi_1"] \arrow[rd, "\pi_2"'] & \\
a & & b
\end{tikzcd}$$

Exercise 3.35. 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. ◊

Exercise 3.36. Given three objects x, y, z, one can define their three-ary 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 $\pi_1 \colon P(x, y, z) \to x$ and $\pi_2 \colon P(x, y, z) \to y$ and $\pi_3 \colon (x, y, z) \to z$, and for any other P' equipped with morphisms $p_1 \colon P' \to x$, $p_2 \colon P' \to y$, and $p_3 \colon P' \to z$, there is a unique morphism $\langle p_1, p_2, p_3 \rangle \colon P' \to P(x, y, z)$ such that $p_i = \pi_i \circ \langle p_1, p_2, p_3 \rangle$ for each $i \in \{1, 2, 3\}$.

1. Suppose that $\mathcal{C}$ has a terminal object. Show that if it has three-ary products then it also has (2-ary, i.e. ordinary) products.
2. Show that if $\mathcal{C}$ has 2-ary products then it also has 3-ary products. That is, show that $P(x, y, z)$ is isomorphic to $(x \times y) \times z$. ◊

If a category $\mathcal{C}$ has a terminal object and (2-ary) 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 3.37. We say that a category is a *cartesian category* if it has all finite products.

3.4.1 The pair type

Let's talk about how to implement products 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
```

For this to behave as a product, we also need to be able to extract the components of pair.

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

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

Exercise 3.38. 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 3.39. In fact, 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:

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

Here we have 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`. Note that we could not have defined this type constructor ourselves, since our names for type constructors must begin with an upper case letter. Instead, this is hard-coded into the language.

The built-in pair type comes with projection maps

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

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`.

Example 3.40 (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 the next section.

3.4.2 Using the universal property

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 in the factors! It's helpful to explicitly have available to 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)   -- pattern match on the pair of functions
```

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 3.41. Show that `tuple` and `untuple` are inverses, and hence the types `(c -> a, c -> b)` and `c -> (a,b)` are isomorphic. ◊

Haskell Note 3.42. 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 this chapter.

3.4.3 Functoriality of the product

If every pair of objects in a category $\mathcal{C}$ has a product, we can define a functor $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$. 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 require two arguments, and is an analogue of `fmap` that asks for two functions rather than one, but still returns one function (between pairs).

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 `(,) :: a -> b -> (a, b)`. To instantiate this type constructor as a bifunctor, we provide `bimap` as follows:

```
instance Bifunctor (,) where
  bimap f g = \ab -> (f (fst ab), g (snd ab))
```

or, using pattern matching:

```
bimap f g ab = (f (fst ab), g (snd ab))
```

$$\begin{tikzcd}
 & a \times b \arrow[dl, bend right, "\pi_1"'] \arrow[d, dashed, "h"] \arrow[dr, bend left, "\pi_2"] & \\
a \arrow[d, "f"'] & a' \times b' \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & b \arrow[d, "g"] \\
a' & & b'
\end{tikzcd}$$

Figure 3.1: Functoriality of the product: $h = \langle (f \circ \pi_1), (g \circ \pi_2) \rangle$

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

$$\begin{tikzcd}
& a \times b \arrow[d, dashed, "\gamma"] \arrow[ddl, bend right, "\pi_2"'] \arrow[ddr, bend left, "\pi_1"] & \\
& b \times a \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
b & & a
\end{tikzcd}$$

2. Associativity

$$\begin{tikzcd}
& a \times (b \times c) \arrow[d, dashed, "\alpha"] \arrow[ddl, bend right, "\mathrm{id}_a \times \pi_1"'] \arrow[dr, "\pi_2"] & \\
& (a \times b) \times c \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & b \times c \arrow[d, "\pi_2"] \\
a \times b & & c
\end{tikzcd}$$

3, 4. Two unit isomorphisms

$$\begin{tikzcd}
& a \arrow[d, dashed, "\lambda^{-1}"] \arrow[ddl, bend right, "!"'] \arrow[ddr, bend left, "\mathrm{id}_a"] & \\
& 1 \times a \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
1 & & a
\end{tikzcd}
\qquad
\begin{tikzcd}
& a \arrow[d, dashed, "\rho^{-1}"] \arrow[ddl, bend right, "\mathrm{id}_a"'] \arrow[ddr, bend left, "!"] & \\
& a \times 1 \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] & \\
a & & 1
\end{tikzcd}$$

Exercise 3.43. Implement functions of the following type signatures:

1. `swap :: (a,b) -> (b,a)`
2. `assoc :: (a,(b,c)) -> ((a,b),c)`
3. `unitl :: a -> (1,a)`
4. `unitr :: a -> (a,1)`
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 3.36 we could also define a triple products (or any finite product, for that matter) using a universal construction:

$$\begin{tikzcd}
& d \arrow[d, dashed, "h"] \arrow[ddl, bend right, "f"'] \arrow[ddr, bend left, "g"] \arrow[ddrr, bend left, "h"] & & \\
& a \times b \times c \arrow[dl, "\pi_1"] \arrow[dr, "\pi_2"'] \arrow[drr, "\pi_3"] & & \\
a & & b & c
\end{tikzcd}$$

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 `()`.

3.4.4 Record syntax

Product types are ubiquitous in programming so a lot of syntactic sugar is provided 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, `fst` and `snd`. But as you keep adding 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 selectors. A pair, for instance, can be defined as:

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

The names given to components also serve as accessor functions; here, these are exactly the two projections. 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

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

Here, we are creating a new version of `p`, called `incrFst p`, whose `fst` field is set to `fst p + 1` (the previous value of `fst p` plus one.

Example 3.44. 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

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

The fields can be accessed through their selectors, as in

```haskell
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

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

You can construct a record either using the record syntax with named fields, or by providing values for all fields in correct order

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

[a]One might also do this 'setting" and "getting" using the Haskell lens library.

Example 3.45 (Single field records). Record syntax is also used in cases when there is only one field, just to have the accessor function named and implemented in one place. Here is a record whose only field is a list of pairs. It can be accessed using `unLp`

```haskell
newtype ListPair a b = LP { unLp :: [(a, b)]}
```

as you can see in this implementation of `fmap`

```haskell
instance Functor (ListPair a) where
  fmap f = LP . fmap (bimap id f) . unLp
```

3.5 Coproducts and sum types

We may dualize the definition of product to obtain the 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 3.46. 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 $i_1 \colon x \to x + y$ and $i_2 \colon y \to x + y$, such that for any object a and morphisms $f \colon x \to a$ and $g \colon y \to a$, there is a unique morphism $h\colon x + y \to a$ such that the following diagram commutes:

$$\begin{tikzcd}
x \arrow[dr, "i_1"] \arrow[ddr, bend right, "f"'] & & y \arrow[dl, "i_2"'] \arrow[ddl, bend left, "g"] \\
& x + y \arrow[d, dashed, "h"] & \\
& a &
\end{tikzcd}$$

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

Remark 3.47. 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.

The computational content of coproducts are as follows. Suppose we want to define a function $h\colon a + b \to c$. The coproduct *decomposes* the task into two simpler problems of defining two functions $a \to c$ and $b \to c$.

3.5.1 Sum types

One may construct generalized elements of $a + b$ 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. 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. We thus obtain the following *sum type*.

```
data Coproduct a b = I1 a | I2 b
```

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
```

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 3.48. What are the types of the functions `foo` and `bar` above? ◊

Example 3.49. The type `Bool` as $1 + 1$ (or 2).

We could define it in one of two ways. The first is to say

```
type Bool = Either () ()
```

with

```
true :: Bool
true = Left ()

false :: Bool
false = Right ()
```

The second is to do it "by hand":

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

Two inclusions (constructors) are maps from unit, `True` and `False`.

Example 3.50. `Maybe` as $1 + a$

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

Two injections, `Nothing` from unit `()` and

```
Just :: a -> Maybe a
```

Here is an example of how you might use a type `Maybe a`.

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

Example 3.51. Cards:

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

3.5.2 Properties 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:

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

It has inverse

```
unEither h = (h . Left, h . Right)
```

This is called (point free) notation. It's equivalent to:

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

Indeed, all the properties of the product have corresponding analogues for the coproduct. This includes the functoriality. 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')
```

This is done as follows:

```
instance Bifunctor Either where
  bimap f g = \eab -> case eab of
                        Left  a -> Left (f a)
                        Right b -> Right (g b)
```

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)
```

As for products, it is also possible to define symmetry, associator, and unitor isomorphisms for the coproduct.

Definition 3.52. 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.

3.5.3 Distributivity

This is something from high school, but this time we are doing arithmetic on types. We'd like to show that there is an isomorphism between the two types.

$$a \times c + b \times c \longrightarrow (a + b) \times c$$

The morphism from left to right is relatively easy to prove from the universal construction. It is, after all, a mapping out of a coproduct and a mapping into a product. We'll see that the other direction is not easy at all, and it doesn't hold in an arbitrary bicartesian category.

The construction of the left to right map can be given by tupling:

$$f : a \times c \longrightarrow (a + b) \times c$$

$$g : b \times c \longrightarrow (a + b) \times c$$

$$\begin{tikzcd}
a \times c \arrow[dr, "i_1"] \arrow[ddr, bend right, "f"'] & & b \times c \arrow[dl, "i_2"'] \arrow[ddl, bend left, "g"] \\
& a \times c + b \times c \arrow[d, dashed, "h"] & \\
& (a + b) \times c &
\end{tikzcd}$$

We can easily find the two morphisms that will uniquely determine the sought for mapping:

$$f = i_1 \times id_c$$

$$g = i_2 \times id_c$$

How do we prove the other direction? In Haskell it works:

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

The reason for this is not obvious, though. This is because Haskell data types go beyond a bicartesian category. They add to it one more universal construction.

3.6 Exponentials and currying

So far we've been treating types as opaque objects of some category. Functions, on the other hand were supposed to be morphisms in that category. This picture breaks down as soon as we try to talk about higher order functions. How can we explain a function that takes a function as an argument or produces a function? It's a morphism between types, so what type is its argument or its return value? We need an object in our category that represent function type. This is not a problem in the category of sets. Functions between two sets form a set, the hom set, and a set is automatically an object in this category. In an arbitrary category, morphisms between two object also form a set, the hom-set, but a set is not an object in that category. What we need is a way of describing a set of functions between two sets without referring to its elements as functions, and then generalize this characterization to an arbitrary category.

There is one property of the function set that can be expressed this way. For any element f of the function set from b to c, there is a function that maps the pair (f, b) to c. It's called the evaluation. It just applies f to b.

Whatever object we select to represent morphism from b to c there must be an evaluation morphism from the product of it with b that goes to c. This is pretty much all that we can say about this object in terms of morphisms. We just have to make sure that this is the *best* such object, because there may be many other contenders.

This is what universal constructions are all about. We have to make sure that any other object a that pretends to be a function object by providing a morphism $f : a \times b \to c$ (which pretends to be an evaluation) is inferior. This inferiority can be tested by requiring that there be a unique morphism from a to the actual function object, which we'll call the exponential c^b, that factors through the actual evaluator, which we'll call ε.

$$\begin{tikzcd}
a \times b \arrow[d, "h\times \mathrm{id}_b"'] \arrow[dr, "f"] & \\
c^b \times b \arrow[r, "\varepsilon_{c,b}"'] & c
\end{tikzcd}$$

Figure 3.2: Universality of the exponential

This factoring out requires some elaboration. We say that there exists a unique morphism $h\colon a \to c^b$, but the morphism to be factored out goes from $a \times b$ to c. But we know from the previous section that the product is a bifunctor, so we can map $a \times b$ to $c^b \times b$ by lifting a pair of morphisms h and id_b.

Remark 3.53. Notice that the definition of the exponential is tightly coupled with the definition of a product. You can't have an exponential if you don't have some kind of a product. On the other hand, the only property of the product that we are using here is that it's a bifunctor. You may ask the question, are there other bifunctors that can be substituted for product in this definition? The answer is, yes. But if you are thinking of trying a coproduct (which is also a bifunctor), this won't work.

Remark 3.54. A category with a terminal object and both a product and an exponential defined for every pair of objects is called *cartesian closed*. Catesian closed categories are important in modeling programming languages and the lambda calculus in particular.

Exercise 3.55. Try to define the exponential object by replacing a product with a coproduct. What goes wrong? Try implementing it in Haskell. ◊

The idea that the exponential object c^b somehow represents the set of morphisms $b \to c$ is substantiated by considering the unviersal construction for the terminal object 1. The terminal object is a unit of product–we have previously constructed the isomorphism $1 \times b \cong b$. Therefore the set of morphisms $1 \times b \to c$ is isomorphic to the set of morphisms $b \to c$. Universality tells us that these morphisms are in turn in one to one correspondence with the morphisms $1 \to c^b$, which are global elements of the exponential object, as seen in the following diagram

$$\begin{tikzcd}
1 \times b \arrow[d, "h\times \mathrm{id}_b"'] \arrow[dr, "f"] & \\
c^b \times b \arrow[r, "\varepsilon_{c,b}"'] & c
\end{tikzcd}$$

Figure 3.3: Global elements of an exponential object

3.6.1 Currying in Haskell

Universality of the exponential tells us that for every $f: a \times b \to c$ there is a unique $h: a \to c^b$. The converse is also true, you can recover f from h by composing

$$\varepsilon \circ (h \times id_b)$$

In other words, there is a one to one correspondence (a bijection) between morphisms that take a product as an argument and morphisms that produce a function object. This is the essence of currying. The slogan is: a function of a pair of arguments is equivalent to a function returning a function. This equivalence is baked into Haskell from the very beginning. This is why we write the signature of a function of two arguments as

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

This is also the reason why we can apply this function to one argument of type `a`, e.g., `f x` to create a function that expects `b` and produces a `c`. This is called *partial application*. Even when we apply `f` to two arguments, conceptually we are performing a two-step application. First we apply it to the first argument, get a function in return, then apply this function to the second argument. This is even true for binary operators like the plus sign. A partial application of a binary operator is called a *section*. Here's a section of plus: `(2+)`. It produces a function of one argument that adds 2 to it.

The exponential object is defined by its mapping in property. On the other hand, its universal construction can be used to produce a mapping out of a product. Suppose that you want to define a function $f: a \times b \to c$. It may sometimes be easier to define a function $h: a \to c^b$ instead. Or the other way around: you have to define a function $h: a \to c^b$. But maybe you have at our disposal a function $f: a \times b \to c$. We'll use these properties to finish our proof of distributivity.

But first, let's express the universal property of the exponential in Haskell:

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

Given a function from a pair, we can create a function that returns a function (an exponential)

Conversely, given a function of two arguments, we can create a function that takes a pair

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

The evaluation function can be defined as

```
epsilon :: (b -> c, b) -> c
epsilon (g, b) = g b
```

We usually see the curried version of the evaluator as an infix operator

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

Because of its low precedence, this operator is often used between a function name and an expression that evaluates to its arguments. For instance, instead of writing

```
f (a + b)
```

we can write

```
f $ a + b
```

Exercise 3.56. What do you think this is?

```
uncurry (+)
```

◊

3.6.2 Functoriality of the exponential

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 $g\colon c \to c'$ and try to lift it to the morphism $c^b \to c'^b$. We are looking for a mapping into an exponential c'^b, so we should use the universal construction to find it.

$$\begin{tikzcd}
c^b \times b \arrow[d, "h' \times \mathrm{id}_b"'] \arrow[dr, "f'"] & \\
c'^b \times b \arrow[r, "\varepsilon_{c',b}"'] & c'
\end{tikzcd}$$

Figure 3.4: Functoriality of the exponential

We are searching for an h' in this diagram. We can get it if we can find a morphism

$$f'\colon c^b \times b \to c'$$

We have at our disposal the evaluator for c^b, which is a morphism $\varepsilon_{c,b}\colon c^b \times b \to c$, so we can simply postcompose it with g.

$$\begin{tikzcd}
a \times b \arrow[d, "h \times \mathrm{id}_b"'] \arrow[dr, "f"] & & \\
c^b \times b \arrow[r, "\varepsilon_{c,b}"'] \arrow[d, "h' \times id_b"'] & c \arrow[dr, "g"] & \\
c'^b \times b \arrow[rr, "\varepsilon_{c',b}"'] & & c'
\end{tikzcd}$$

Figure 3.5: Composition of two universal diagrams

Altogether we get

$$h' = curry(g \circ \varepsilon_{c,b})$$

However, if we tried to do the same trick to establish fuctoriality of c^b in b, we would fail. The reason is that the exponential is contravariant in b. It cannot lift a morphism $b \to b'$ but it can lift a morphism going in the opposite direction: it can lift $g \colon b' \to b$ to the morphism $c^b \to c^{b'}$. As before, we'll use the universal construction to find it.

$$\begin{tikzcd}
c^b \times b' \arrow[d, "h' \times \mathrm{id}_{b'}"'] \arrow[dr, "f'"] & \\
c^{b'} \times b' \arrow[r, "\varepsilon_{c,b'}"'] & c
\end{tikzcd}$$

Figure 3.6: Functoriality of the exponential

This time we need a morphism

$$f' \colon c^b \times b' \to c$$

We can obtain it by this composition

$$f' = \varepsilon_{c,b} \circ (id_{c^b} \times g)$$

Notice that this time we are *precomposing* ε with g. This is why we needed a g that was going in the opposite direction.

Categorically, we can interpret the lifting of a pair of morphism, one of them going in the opposite direction as defining a functor from a product category, in which the first category is the opposite of **C**. Recall that the opposite category has the same objects as the original one, but all the arrows are reversed. If the exponential object is defined for every pair of objects in **C** then it defines a functor

$$C^{op} \times C \to C$$

This functor lifts a single morphism in $\mathbf{C}^{op} \times \mathbf{C}$, which is a pair of morphisms from **C**, the first going in the opposite direction.

3.6.3 Functoriality in Haskell

These two constructions can be translated to Haskell as

```haskell
rmap :: (c -> c') -> (a -> c) -> a -> c'
rmap g = curry (g . epsilon)
```

```haskell
lmap :: (b' -> b) -> (b -> c) -> b' -> c
lmap g = curry (epsilon . bimap id g)
```

Since `epsilon` is really function application and currying is built into Haskell, these definitions can be substantially simplified. They can also be combined into one function

```haskell
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, there is a typeclass for functors from $\mathbf{C}^{op} \times \mathbf{C}$ in Haskell

```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`

```haskell
instance Profunctor (->) where
  dimap g' g h = g . h . g'
```

3.6.4 Distributivity revisited

The other side of distributivity

$$(a + b) \times c \to a \times c + b \times c$$

involves mapping out of a product. We are looking for a function

$$f : (a + b) \times c \to a \times c + b \times c$$

The universal construction of the exponential object contains a recipe for such mappings. It tells us that we need to first find

$$h : (a + b) \to (a \times c + b \times c)^c$$

and then uncurry it to get the desired result.

$$\begin{tikzcd}
(a+b)\times c \arrow[d, dashed, "h\times id_c"'] \arrow[dr, "f"] & \\
(a\times c+b\times c)^c \times c \arrow[r, "eval"'] & a\times c+b\times c
\end{tikzcd}$$

Figure 3.7: Uncurrying

As you can see, the h we are looking for is a mapping out of a coproduct, so it is uniquely determined by two morphisms:

$$h_1 : a \longrightarrow (a \times c + b \times c)^c$$

$$h_2 : b \longrightarrow (a \times c + b \times c)^c$$

These are mapping into exponential objects, we we can get them from the universal construction. The first one is given by currying the first injection into the coproduct

$$\begin{tikzcd}
a\times c \arrow[d, dashed, "h_1\times id_c"'] \arrow[dr, "i_1"] & \\
(a\times c+b\times c)^c \times c \arrow[r, "\varepsilon"'] & a\times c+b\times c
\end{tikzcd}$$

Figure 3.8: Defining h_1

This translates to Haskell as `h1 = curry Left`. Similarly for h_2 we get `curry Right`. Combining all the steps we end up with

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

This kind of point-free implementations are notoriously difficult to obtain and, in general, are difficult to analyze, so they are avoided in practice (unless we want to impress other Haskellers). We were able to derive it with the help of category theory and its diagrammatical language. In the process, we gained important insights about distributivity. We learned that, in one direction, it works in any bicartesian category, but to prove it in the opposite direction, one needs a cartesian closed category.

Chapter 4

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 expression and the other is the evaluation.

From a point of view of a programmer, an expression is a tree whose nodes are operators, like $+$ or $*$, and whose leaves are either constants or placeholders for values. The latter correspond to variables. For instance, the expression

$$2x^2 + 3x + 4$$

corresponds to a tree

$$\begin{tikzcd}
 & & + \arrow[dl] \arrow[dr] & & \\
 & * \arrow[dl] \arrow[d] & & + \arrow[d] \arrow[dr] & \\
2 & * \arrow[dl] \arrow[dr] & & * \arrow[d] \arrow[dr] & 4 \\
x & & x & 3 & x
\end{tikzcd}$$

This is how a parser would see this expression: as a parsing tree. More precisely, it would see it as an instance of a tree-like data structure, whose nodes are operators `Plus` and `Times` and leaves are `Const` and `Var`.

```
data Expr = Plus  Expr Expr
          | Times Expr Expr
          | Const Double
          | Var   String
```

This is a recursive data structure: its definition refers to itself. When constructing an expression, you can bootstrap yourself by first constructing the non-recursive expressions using the `Const` or `Var` constructors. These in turn could be passed to the recursive constructors `Plus` and `Times`. The expression tree from our example would be written as

```
expr :: Expr
expr = Plus (Times (Const 2)
                   (Times (Var "x") (Var "x")))
            (Plus  (Times (Const 3) (Var "x"))
                   (Const 4))
```

The recursive process of constructing expressions can be decomposed into a series of smaller non-recursive steps. First we define a non-recursive type constructor with a placeholder `a` replacing the recursive branches.

```
data ExprF a = PlusF  a a
             | TimesF a a
             | ConstF Double
             | VarF   String
```

One thing we can do with it is to apply it 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"
```

We are stuck then, because if we wanted to use the other two constructors, we would have to provide them with terms of type `Void`, and there aren't any. If we want to construct depth-2 trees, we have to apply `ExprF` to leaves. We define a new type

```
type Expr2 = ExprF Leaf
```

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

```
e3xp4, e2x2 :: Expr3
e3xp4 = PlusF e3x e4  -- 3 * x + 4
e2x2  = TimesF e2 ex2 -- 2 * x * x
```

Finally, we can recreate our original expression tree.

```
expr' :: Expr4
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: If we apply `ExprF` infinitely many times, we should get a data type that can deal with any depth tree. Amazingly enough, as we will soon see, this procedure can be made rigorous, and yield the original recursive definition of `Expr` (or, at least, an isomorphic type). This will work under one important condition: the type constructor that we are iterating must be a functor. It so happens that `ExprF` is indeed a functor.

Exercise 4.1. Implement the functor instance for `ExprF` ◊

This example illustrates how we can create expressions starting from a functor. Algebraic expressions don't make much sense, though, unless we can evaluate them. What does it mean "to evaluate"? It means replacing the whole tree with a single value of some specified type. The obvious choice would be to pick `Double` as the target type, since the `Const` leaf contains it. But there are other options as well. We could, for instance, evaluate the whole expression to a single string, say `"2 * x * x + 3 * x + 4"`. In fact, we could come up with a way to evaluate the expression to any type we could think of, if we abandon our preconceptions about the meaning and properties of the operators in question.

Let's look at an example of evaluating to `Double`, as it makes most sense to us

```
eval1 :: ExprF a -> Double
```

We start with the leaves. The `ConstF` leaf is obvious:

```
eval1 (ConstF x) = x
```

To evaluate the variable leaf, we have to decide what value to assign to `x`. Let's say, our evaluator will make `x` equal to 2 (we can create separate evaluators for each interesting value of `x`, or make a function that takes `x` and returns the appropriate evaluator)

```
eval1 (VarF "x") = 2
```

We are pattern matching the constructor `VarF` and, inside it, pattern matching the string `"x"`. This is not an exhaustive match, so we'll provide the default pattern as well

```
eval1 (VarF _) = undefined
```

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. For the time being, we'll make the program fail when the variable is not `"x"`. 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'". We have to tell the compiler that we have made our choice: the type of `a` has to 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. They would contain 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 x)   = x
eval1 (VarF "x")   = 2
eval1 (VarF _)     = undefined
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 rather than a functor between categories, because we have to be able to recursively apply it to itself. Then we define evaluators for this functor–we'll call them F-algebras. Here, we did this by picking a type a (`Double`, in our case) and a morphism $Fa \to a$. Notice that, have we picked a different type, say `String`, we would have to implement a different evaluator. The evaluator is *not* a parametrically polymorphic function–it's not "one formula for all types."

Exercise 4.2. Implement an evaluator `showEx :: ExprF String -> String` that produces a text version of the expression. ◊

Exercise 4.3. Implement a recursive function that evaluates an `Expr`. When working with a node, it should call itself to evaluate the children and then combine the results. ◊

4.1 Algebras

Definition 4.4. Let $F\colon \mathcal{C} \to \mathcal{C}$ be a functor. An F-algebra (c, φ) is

(a) An object $c \in \mathcal{C}$, called the *carrier*

(b) A morphism $\varphi\colon Fc \to c$, called the *structure map*

Given two F-algebras (c, φ) and (d, ψ), a morphism $f\colon (c, \varphi) \to (d, \psi)$ is a morphism

$f\colon c \to d$ in $\mathcal{C}$ such that

$$\begin{tikzcd} Fc \arrow[r, "\varphi"] \arrow[d, "Ff"'] & c \arrow[d, "f"] \\ Fd \arrow[r, "\psi"'] & d \end{tikzcd}$$

commutes.

Proposition 4.5. There is a category $F\text{−}\mathbf{Alg}$, whose objects are F-algebras, morphisms are morphisms of F-algebras, and whose composition and identities are given by that in $\mathcal{C}$.

Exercise 4.6. 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. ◊

Example 4.7. Consider the functor $F\colon \mathbf{Set} \to \mathbf{Set}$ that maps X to $1 + \mathbb{Z} \times X$, and sends $f\colon X \to Y$ to $\mathrm{id}_1 + \mathrm{id}_{\mathbb{Z}} \times f\colon 1 + \mathbb{Z} \times X \to 1 + \mathbb{Z} \times Y$.

An algebra for this functor is given by taking $\mathbb{Z}$ to be the carrier and $[0, +]\colon 1+\mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}$ to be the structure map.

4.2 Initial algebras

Definition 4.8. An *initial algebra* for a functor F is an initial object in the category of F-algebras.

The above definition is a little terse. It unpacks as follows. Given a functor $F\colon \mathcal{C} \to \mathcal{C}$, an initial algebra (i, j) for F is an object i and morphism $j\colon Fi \to i$ such that for every F-algebra $f\colon Fa \to a$ there is a unique morphism $h\colon i \to a$ in $\mathcal{C}$ such that the diagram

$$\begin{tikzcd} Fi \arrow[r, "Fh"] \arrow[d, "j"] & Fa \arrow[d, "f"] \\ i \arrow[r, "h"] & a \end{tikzcd}$$

commutes.

The unique morphism $h\colon i \to a$ from the carrier of the initial algebra is called the *catamorphism* for $f\colon Fa \to a$.

4.2.1 Lambek's lemma

Theorem 4.9 (Lambek's lemma)**.** The structure map of the initial algebra is an isomorphism.

Proof. Notice that if (i, j) is an initial algebra than (Fi, Fj) is an algebra as well. Initiality tells us that there is a unique algebra morphism $h\colon i \to Fi$.

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

We want to show that h is the inverse of j. It's easy to show that j itself is an algebra morphism, because the following diagram trivially commutes

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

Pasting the two diagrams along the common arrow, we arrive at the conclusion that the outer rectangle in the following diagram commutes

$$\begin{tikzcd}
Fi \arrow[r, "Fh"] \arrow[d, "j"] & F(Fi) \arrow[r, "Fj"] \arrow[d, "Fj"] & Fi \arrow[d, "j"] \\
i \arrow[r, "h"] \arrow[rr, bend right, "j\circ h"'] & Fi \arrow[r, "j"] & i
\end{tikzcd}$$

Therefore, the composition $j \circ h$ is an algebra morphism. Moreover, this is an algebra morphism between (i, j) and itself. Initiality tells us that there can only be one such morphism, and it so happens that the identity morphism is also an algebra morphism. Therefore $j \circ h = id$. We can now go back to the original diagram and notice that the commuting condition

$$h \circ j = Fj \circ Fh$$

can be rewritten using functoriality of F as

$$h \circ j = F(j \circ h)$$

But since $j \circ h = id$, and a functor maps identity to identity, we must have $h \circ j = id$. The following diagram illustrates this

$$\begin{tikzcd}
Fi \arrow[r, "Fh"] \arrow[d, "j"] \arrow[dr, bend left, "F(j\circ h)"] \arrow[dr, bend right, "h\circ j"'] & F(Fi) \arrow[d, "Fj"] \\
i \arrow[r, "h"'] & Fi
\end{tikzcd}$$

□

We have shown that h is the inverse of j, therefore Fi is indeed isomorphic to i. Informally, this can be explained as: once you reach the fixed point of F, further application of the functor doesn't change anything.

4.3 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 a tree. The functor in question generated leaves and nodes. The nodes contained placeholders of arbitrary type. The recursive version of the tree replaced these placeholders with the recursive versions of the tree. For instance, the node

```
PlusF a a
```

became

```
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 has the property that one more application of the function doesn't change it. 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 X of a functor F can be defined the same way. It's a solution to the equation

$$X = FX$$

This equation can in fact be used as a Haskell definition of a data type. 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, which we'll rewrite as

$$X(F) = F(X(F))$$

to emphasize that the fixed point depends on the functor in question:

```
data X f = MakeX (f (X 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.

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.

Here's the actual definition of the fixed point type from `Data.Fix`

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

This definition 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. We use `newtype` in place of `data`. The only difference between the two is performance, so every time we are allowed to use `newtype`, we'll use it[1].

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
```

It can be shown that the *least fixed point* of a functor is also the carrier of its initial algebra. 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

```
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`)

[1] `data` can be replaced by `newtype` if there is exactly one data constructor with exactly one field.

```
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` is automatically establishing an isomorphism.

Let's go back to our initial example that was based on the following functor

```
data ExprF a = PlusF  a a
             | TimesF a a
             | ConstF Double
             | VarF   String
    derive Functor
```

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 this functor

```
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. 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

```
-- 2 x^2 + 3 x + 4
ex'' = add (mul (num 2)
                (mul (var "x")(var "x")))
           (add (mul (num 3) (var "x")) (num 4))
```

4.3.1 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 in the main body of 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. 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 plug them into the holes in our functor. This is the meaning of `f (Fix f)` in the definition of the fixed point.

We can go even further. We can apply 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 `Double`s. We have a functor-full of `Double`s. In our case that would be 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 an algebra `Algebra ExprF Double`. The algebra is the recursive step that we looked for. We also have a machine that accepts an algebra and cranks up the recursion. It's called the catamorphism.

4.3.2 Algebras, catamorphsims, 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
```

It should be stressed 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`.

An algebra defines a single recursive step. What we need is a way to apply this algebra recursively to the recursive data structure that is a fixed point of the functor. In other words, we are looking for an algebra morphism from the initial algebra to a given algebra. This morphism is called a catamorphism and, thanks to Lambek's lemma, we can provide a closed formula for it. 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`.

$$\begin{tikzcd}
f(Fix\, f) \arrow[r, "fmap\,(cata\, alg)"] \arrow[d, shift left, "Fix"] & f\, a \arrow[d, "alg"] \\
Fix\, f \arrow[u, shift left, "unFix"] \arrow[r, "cata\, alg"] & a
\end{tikzcd}$$

Because the diagram commutes, we can read it as

$$cata\, alg = alg \circ fmap\, (cata\, alg) \circ unFix$$

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 _) = undefined
eval1 (PlusF x y) = x + y
eval1 (TimesF x y) = x * y
```

We can use the catamorphism to evaluate the expression ex''

```
> cata eval1 ex''
> 18.0
```

A 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.

4.3.3 Lists

A list is a workhorse data structure in Haskell. Its definition is simple and there is a large library of functions that operate on lists. One should however keep in mind that a list is not a very efficient vehicle for storing large amounts of data. It should be treated more like a control structure for algorithms (especially ones that process data in a linear fashion) than as a storage device. If you are writing a program whose main purpose in processing large amounts of textual information, there are other ways of storing it.

Since lists are so ubiquitous, Haskell provides special syntax for them. The list type constructor is a pair of square brackets, so a list of Int is simply [Int]. There are two data constructors, one is a pair of empty square brackets [] to construct an empty list, and the other is a colon usually used in infix notation. It's type is

```
(:) :: a -> [a] -> [a]
```

It takes a value of type a as the head of the new list, and a list [a] as its tail.

We can also define our own list type, using more verbose syntax

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

Here, `Nil` and `Cons` play the role of `[]` and `:`. This is a recursive definition, so there must be a version of it that is a fixed point of an endofunctor. This endofunctor is

```
data ListF a x = NilF | ConsF a x
  deriving Functor
```

The type variable `a` describes the payload of the list.

If follows then that the recursive definition is equivalent to the fixed point of this functor

```
type List a = Fix (ListF a)
```

A list algebra for a carrier `b` has the type

```
type ListAlg a b = Algebra (ListF a) b
```

It is determined by a function that accumulates the values in the accumulator of type `b` and a single value of type `b` to initialize the accumulator

```
mkAlgebra :: (a -> b -> b) -> b -> ListAlg a b
mkAlgebra f b NilF = b
mkAlgebra f b (ConsF a x) = f a x
```

To evaluate, or *fold*, a list we use a catamorphism

```
fold :: (a -> b -> b) -> b -> List a -> b
fold f b = cata (mkAlgebra b f)
```

This function is defined in the Haskell Prelude. There are actually two versions of it, one accumulating from the right

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

and one accumulating from the left

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

They have different performance characteristics.

Exercise 4.10. Explain what this code does when applied to a list of integers

```
foldr (+) 0
```

Try applying it to the list `[1..100]` (this is syntax for a list from 1 to 100) ◊

4.4 Coalgebras, anamorphsims, and unfolds

As usual, every construction in category theory has its dual. If you reverse the arrows, an algebra becomes a coalgebra.

```
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 is the type of the seed. The idea is that you are given a seed and you use it to create a single level of a recursive data structure that is described by a functor. 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. An anamorphism takes a colagebra and turns a seed into a recursive data structure

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

In a general categorical setting, we are talking about a terminal coalgebra. The Lambek's lemma still holds and it makes a terminal coalgebra a fixed point of a functor. This time it's the greatest fixed point. In Haskell, however, if the fixed point is unique, then the initial algebra coincides with the terminal coalgebra. This means, we can use the same `Fix` data structure to evaluate terminal coalgebras.

Coalgebras' greatest application is in defining infinite data structures. Because of Haskell's laziness, it's perfectly okay to define a data structure that stretches forever. The data stored in such a structure is only evaluated when the program tries to access it. The trick of designing infinite data structures can help us transform the way we think about algorithms. Instead of coding a system of recursive functions, we can instead work on generating and traversing a data structure. Let's see how it works in practice.

The simplest non-trivial infinite data structure is a generalization of a list called a `Stream`. It is generated by a functor that drops one of the list constructors, the one that terminates the recursion.

```
data StreamF a x = StreamF a x
  deriving Functor
```

Another way of looking at it is that it's a product of `a` and `x`, so it's isomorphic to a pair. The type of an infinite stream is the fixed point of this functor

```
type Stream a = Fix (StreamF a)
```

We can expand this definition by tying the recursive knot—replacing `x` with the result of the recursion

```haskell
data Stream a = MkStream a (Stream a)
```

This shows us how we can map out from a `Stream` by pattern matching on its constructor `MkStream`. We get the head of the stream and its tail, which is another `Stream`, we can extract the head of this stream, in turn, and so on, ad infinitum. The stream never ends.

But how do we create terms of such type? We can't possibly pre-fill it with the infinity of values. This is where an anamorphism shows its usefulness. An anamorphism can (lazily) generate an infinite data structure from a seed.

In Haskell, the same list data type we've seen before can be used for infinite streams. A stream is simply a list that, when pattern matched, will never match the empty list pattern `[]`. A few anamorphisms are even built into the language. For instance, the infinite list of integers starting with 2 can be created using the syntax

```haskell
intsFrom2 :: [Int]
intsFrom2 = [2..]
```

Exercise 4.11. Implement a function `intFrom :: Int -> Stream Int` that uses an anamorphism to generate a `Stream` of integers starting from `n`. ◊

Here's an interesting example. We are going to generate a stream of prime numbers using a version of the sieve of Eratorsthenes. We start with a seed that is the list of all integers greater than one. It starts with a prime number, so we extract it. Then we eliminate all multiples of this number from the tail. This decimated list becomes our new seed. We package this idea in a stream `Coalgebra` whose carrier is our seed—a list of integers.

```haskell
era :: Coalgebra (StreamF Int) [Int]
era (p : ns) = StreamF 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 from the head of the list. The second is a filtered tail. `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`. The predicate is implemented in the **`where`** clause. It performs division modulo and compares the result to zero using the inequality[2] 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 we can generate an infinite stream of prime numbers from the seed `[2..]`

```
primes = ana era [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.

Exercise 4.12. Implement a function that converts a **`Stream`** to an (infinite) list

```
toList :: Stream a -> [a]
```

Hint: Implement an algebra

```
alg :: Algebra (StreamF a) [a]
```

and apply a catamorphism to it. In order to display the result, use the function `take` which truncates the (possibly infinite) list down to size.

```
take 100 (toList primes)
```

◊

This pattern of applying a catamorphism immediately after an anamorphism is common enough to deserve its own function called a *hylomorpshism*.

```
hylo :: Functor f => Algebra f a -> Coalgebra f b -> b -> a
hylo f g = f . fmap (hylo f g) . g
```

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

[2] In many programming languages this operator is encoded as `!=`.

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 4.13 (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 know how to deal 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). ◊

4.5 Fixed points in Haskell

In Haskell, least fixed point and greatest fixed point, when uniquely defined, coincide. We can still define them separately by directly encoding their universal property. The initial algebra can be defined by its mapping out property.

```
newtype Mu f = Mu (forall a. (f a -> a) -> a)
```

Notice that this definition requires the following language pragma

```
{-# language RankNTypes #-}
```

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)
```

(`flip` is a function that reverses the order of arguments of its (function) argument.) What the definition of `Mu` is saying is that it's an object that, for all algebras, has a mapping out to a catamorphism.

It's easy to define a catamorphism in terms of `Mu`, since `Mu` *is* a catamorphism

```
cataMu :: Functor f => Algebra f a -> Mu f -> a
cataMu alg (Mu cata) = cata alg
```

The challenge is to construct terms of type `Mu f`. For instance, let's convert a list of `a` 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))
```

Notice that we 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 #-}
```

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
```

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))
```

Existential types can be encoded in Haskell using the so called Generalized Algebraic Data Types or GADTs

```
data Nu f where
  Nu :: (a -> f a) -> a -> Nu f
```

The use of GADTs requires the language pragma

```
{-# language GADTs #-}
```

The argument 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` and obtain the term of the type `f a`. Since `f` is a functor, we can lift our function and apply it again, to 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 we are no longer guaranteed that the initial algebra is the same as the terminal coalgebra for a given functor.

Chapter 5

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 of partial computation. These are computations that are not defined for all values of their arguments. We can model such computations using the `Maybe` data type. A partial computation from `a` to `b` can be implemented as a pure 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. A morphism from a to b in **Kl**(*Maybe*) is represented by a function `a->Maybe b` in Haskell.

To define a category, we have to define composition. But two composable morphisms in **Kl**(*Maybe*) don't correspond to composable functions in **Hask**. We need to construct a representative of the composition of two morphisms from their two representatives. We need a higher order function in Haskell that has the following signature

```
(<=<) :: (b -> Maybe c) -> (a -> Maybe b) -> (a -> Maybe c)
```

Compare this with regular function composition

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

There is an obvious implementation: 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 **Kl**(*Maybe*) then `g <=< f` produces a representative of their composition. This composition operator is often called the *fish* or the Kleisli arrow.

Next we need an identity morphism. It is represented by

```
idMaybe :: a -> Maybe a
idMaybe a = Just a
```

It's easy to see that this is indeed the identity morphism with respect to fish composition. The fish is also associative. Therefore **Kl**(*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.

5.2 Monads in Haskell

5.2.1 Monad and Kleisli composition

This procedure of constructing a Kleisli category can be generalized to functors other than `Maybe`, for which we are able to supply lawful implementations of the Kleisli arrow and the Kleisli identity. In fact this is one way of defining a monad 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
```

Traditionally, the representative of the identity morphism is called `return`.

The advantage of this definition is that the laws are very easy to formulate: they are just the laws of a category—associativity and unitality

```
(f <=< g) <=< h = f <=< (g <=< h)
f <=< return    = f
return <=< g    = g
```

5.2.2 Monad and join

The Kleisli arrow can be decomposed taking advantage of the fact that `m` is a functor. Indeed, we can start by trying to `fmap` the first function over the result of the second function

```
  f <=< g = \a -> let mb = g a
                      mmc = fmap f mb
                  in _ mmc
```

The result, however, is of the type `m (m c)` and we need something of the type `m c`. In other words, if we had a function `m (m a) -> m a`, we could implement the Kleisli arrow. Hence the alternative definition of a monad

```
class Functor m => Monad m where
  join :: m (m a) -> m a
  return :: a -> m a
```

One can derive the laws for this definition from the laws for the Kleisli definition. This definition mirrors closely the categorical definition of a monad, with `return` in place of η and `join` in place of μ.

5.2.3 Monad and bind

Notice that, in our alternative implementation of the Kleisli arrow

```
  f <=< g = \a -> let mb = g a
                  in join (fmap f mb)
```

we used the combination of `join` and `fmap`. This combination has the following type signature (with some renaming)

```
bind :: m a -> (a -> m b) -> m b
```

This observation leads to another definition of `Monad`

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

The bind function is represented by an infix operator. There is no `Functor` constraint because `fmap` can be constructed from bind and return. It is implemented as

```
liftM :: Monad m => (a -> b) -> m a -> m b
liftM f ma = ma >>= (\a -> return (f a))
```

This used to be the official definition of `Monad` in Haskell, until the additional constraint of `Applicative` was added. More about it later.

`join` is defined using bind in the library `Control.Monad` as

```
join :: Monad m => m (m a) -> m a
join mma = mma >>= id
```

5.2.4 `do` notation

The bind operator takes a monadic value and applies a Kleisli arrow to it. If a lot of Kleisli arrows are available, one can build programs by composing them and applying them to monadic values. This way one can essentially develop pont-free notation in the Kleisli category. But point free notation is often hard to read, especially if Kleisli arrows are created inline using lambdas. Look for instance at the definition of `liftM`.

This is why Haskell supports special syntax for composing monadic operations, the `do` notation. This is `liftM` in `do` notation

```
liftM f ma = do
               a <- ma
               return (f a)
```

Every line of a `do` block contains a monadic object (here `ma`), or a function call that generates a monadic object (here, `return` applied to `f a`), or a sub-block that results in one. This object is an argument to an implicit bind. The second argument of bind is an implicit lambda. The name of the argument to that lambda is specified in front of the arrow in the previous line. Here, the argument is called `a`. The rest of the `do` block forms the body of this lambda. Here, it's just one line, `return (f a)` but, in general, it could be a multi-line body that is further recursively desugared.

Here's another example

```
pairM :: Monad m => m a -> m b -> m (a, b)
pairM ma mb = do
                a <- ma
                b <- mb
                return (a, b)
```

This function can be rewritten using the desugaring rules as

```
pairM ma mb =
  ma >>= \a ->
    mb >>= \b ->
      return (a, b)
```

Notice the nested lambdas.

Sometimes we want the implicit lambda to ignore its argument. This can be written using a wildcard on the left hand side of the arrow or by omitting the left arrow altogether. Here's an example that uses the `IO` monad

```
main :: IO ()
main = do
  putStrLn "Tell me your name" -- no arrow
  name <- getLine
  putStrLn ("Hello " ++ name) -- no arrow
```

`do` notation has a distinctly imperative look. The left arrow notation suggests assignment

```
a <- ma
```

It looks like extracting `a` from the monadic argument `ma`. It's often read as "`a` gets `ma`." The extracted variable is usually later used in the body of the `do` block.

5.2.5 Monads and effects

The introduction of monads into programming was prompted by the need to provide semantics to functions that had side effects. 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
- Nondeterminism: Computations that may return many results
- Side effects: Computations that access/modify state
  - Read-only state, or the environment
  - Write-only state, or a log
  - Read/write state
- Exceptions: Partial functions that may fail
- Continuations: Ability to save state of the program and then restore it on demand
- Interactive Input
- Interactive Output

It turns out that all of them can be implemented as Kleisli arrows for an appropriate monad. We'll skip the discussion of the partiality monad and list Haskell implementations of the rest.

5.2.6 The list monad and nondeterminism

A computation that may return different values at different times can be represented in Haskell using fuctions returning lists of possible values. When composing two non-deterministic Kleisli arrows we want to apply the second arrow to every possible outcome of the first arrow. The identity arrow deterministically produces one output, so it should produce a singleton list. We have

```
instance Monad [] where
  as >>= k = concat (fmap k as)
  return a = [a]
```

The function `concat` concatenates its arguments

```
concat :: [[a]] -> [a]
```

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)
```

5.2.7 The writer monad

```
data Writer m a = Writer m a
  deriving Functor
```

```
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
```

5.2.8 The reader monad

```
newtype Reader e a = Reader (e -> a)
  deriving Functor

runReader :: Reader e a -> e -> a
runReader (Reader f) e = f e
```

```
instance Monad (Reader e) where
  ra >>= k = Reader f
    where f e = let a  = runReader ra e
                    rb = k a
                in runReader rb e
  return a = Reader (\_ -> a)
```

5.2.9 The state monad

```
newtype State s a = State (s -> (a, s))
  deriving Functor
```

```
runState :: State s a -> s -> (a, s)
runState (State f) s = f s
```

```
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))
```

5.2.10 The continuation monad

```
data Cont r a = Cont ((a -> r) -> r)
```

```
runCont :: Cont r a -> (a -> r) -> r
runCont (Cont k) h = k h
```

```
instance Monad (Cont r) where
    ka >>= kab = Cont (\hb -> runCont ka (\a -> runCont (kab a) hb))
    return a = Cont (\ha -> ha a)
```

5.2.11 The IO monad

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. The compiler could optimize multiple calls to getLine by memoizing its result the first time it was called, and using it over and over. Output could be optimized away completely, since it doesn't produce any value. This is clearly wrong.

The solution adopted by Haskell is to, at least conceptually, postpone 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 is encapsulated inside the IO monad. Every Haskell program contains main, which is an IO object. 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 into the language, you can create a value of the type IO () (that is IO of unit) 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 signature of putStrLn is

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

Anything more complicated can be composed, either using bind, or the **do** notation.

The IO monad is peculiar in that it provides no way of accessing its contents. For instance, to get a character of input, you would use

```
getChar :: IO Char
```

but there is no way to bring this `Char` into the open. You can manipulate this `Char` by, for instance, applying `fmap` to it (`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.

The do notation may give you the impression that you can access a string `name` in this program

```
main :: IO ()
main = do
  putStrLn "Tell me your name"
  name <- getLine
  putStrLn ("Hello " ++ name)
```

but you know that `name` is just a name of the argument to a lambda function that will produce another `IO` object.

5.2.12 `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 nuissance, 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
```