A Brief Course in Modern Math for Programmers

VLAD PATRYSHEV

Contents

Introduction

The specific purpose of this book is to help practicing programmers who have gaps in their education fill these gaps without being forced to return to college. The world is changing, and mathematics itself is undergoing a great revolution. The foundations of computer science are being replaced. Notions that were officially designated as abstract nonsense even by algebraists are becoming a daily reality in our coding practice.

Let me name a few of these notions. Monoids: you need a monoid to apply map/reduce, to implement or use efficient data storage structures, like finger tree, and you need monads and a Kleisli category to explain and formalize side effects in your application.

Much of this is not included in regular math courses for engineers. Mathematics courses, except for the version studied by post-grads in the best universities, mostly consist of centuries-old ideas and notions. These ideas may be good, but the world of ideas in computing is already much further ahead, and we must either catch up or perish, mentally and professionally.

This book is not for mathematicians. They should find better sources of information or inspiration, those that are mathematically strict, contain many theorems and exercises, and are probably targeting a more distant future. For example, there is no topology in this book, no homotopy, no functional analysis, and no equations. No linear algebra, either. These days, linear algebra is extremely popular in machine learning, but machine learning is not programming.

The main purpose of this book is to provide the reader with food for thought, material for imagination, and ideas from modern mathematics that have been used in programming practice for a while now by those who know these things but which about 90% of practicing programmers would find totally alien.

Readers are expected to be, but not necessarily, practicing programmers. So, instead of set theory, on most occasions, types are used. Currently, programmers, for good reasons, know more about types than about sets. Set theory was used as a foundation for mathematics, while its axioms made it possible to prove theorems about functions and real numbers that have long been accepted.

When we write real code, these theorems are often irrelevant or just inapplicable, and state the existence of objects that cannot be observed even theoretically. Types, on the other hand, are unassuming—we only postulate properties that can be validated and are necessary in practice. We will cover set theory and its axioms in detail later on. Just keep in mind that not everything in math is based on set theory. If you are looking for an example, consider set theory, which is not based on set theory.

Code samples are mostly in Scala, and some are in JavaScript. If you are a Java or C#/++ programmer, you will understand Scala easily. If you are a Haskell programmer, please be

patient as you still may find something new in this book. If you use dynamic types or no types, this may be a good opportunity to gain some *practical* ideas with no type theory involved.

Over 50 years ago, an amazing book was published by Faure, Kaufmann, and Denis-Papin: *A New Mathematics*. This small book was a rather easy read, an adventure in math, and a version of *Harry Potter*. It covered a wide range of areas of mathematics, explaining algebra, topology, logic, and analysis, and in such a way that, as a schoolboy, I just could not stop reading it. It was like a sci-fi novel. This book has always been an ideal example for me. I am humbly attempting to follow the authors' approach.

Credits:

I am deeply thankful to my friends who reviewed the book while I was writing it. Their remarks have helped remove most of the errors and bugs, and their wise advice contributed a lot to the quality of the content: Dmitry Cheryasov, Michael Steinhaus, Alex Filippov, Alex Otenko, Georgiy Korneev, Dmitry Bisikalo, Kris Nuttycombe, and Pavel Lyutko.

This book consists of the following chapters:

Chapter 1. Functions In this chapter, I introduce the notions that are related to functions, such as domain, range, and composition; which functions are injective, surjective, and bijective; currying, etc. All these notions are probably taught in high school or at university, but often nobody remembers them. In this book, set theory is not involved in the discussions about these notions. We rely, informally, on types.

Chapter 2. Abstractions of Algebra *Monoid*: the basis of many programming constructs.

Semigroup: "monoids without neutral elements." If we fold, instead of reducing, only a semigroup is needed.

Magma: a "semigroup without associativity," like tuples and binary trees. They are everywhere in code.

Laws of associativity, commutativity, and idempotence.

Naive sets that are just a form of monoid, where union (and intersection) are commutative and idempotent.

Functions that preserve algebraic structures are defined and discussed.

Chapter 3. Partial orders, Graphs, and DAGs A *poset* is a partially ordered set and is a simple, unassuming structure. A special kind of poset is one in which for each pair of elements a minimum and a maximum exists. Add top and bottom elements, and we have two monoids.

This structure is called a *lattice*. *Graph* can be defined in many ways. In this book, a graph is defined as a directed multigraph. If the graph is acyclic (known as a *DAG*, a directed acyclic graph), it can also be considered a poset. A *tree* is a special kind of DAG.

Chapter 4. Boolean Logic Discussion of propositional logic, truth tables, conjunction, disjunction, negation, implication, Sheffer stroke, Peirce arrow, sound and valid arguments, De Morgan laws, and double negation. Additionally, *two monoids*—one with conjunction, another with disjunction—are discussed.

Chapter 5. Non-Boolean Logic Although programmers are used to binary logic, it is fairly easy to introduce a type of logic with more than two logical values.

As an example, consider SQL. A boolean value can be `true/false/null`. These three values, if treated properly, constitute an example of a ternary logic.

Such a logic does not need to support the law of double negation, but it may; a special case is intuitionistic logic. Truth tables for intuitionistic logic. Examples will be presented, and we will examine which De Morgan laws work in this case. Similar to Boolean Logic, we have two monoids in this logic.

Chapter 6. Quantifiers Universal $\forall$ and existential $\exists$, `collection.foreach`, `collection.exists`, informal checking of sentences with quantifiers; differences between $\exists\forall$ and $\forall\exists$, with code samples.

Chapter 7. Models and Theories A gentle introduction to model theory.

Chapter 8. Category is a Multi-tiered Monoid Introduction of notion. Comparison of categories with monoids and graphs. Many examples of categories, including types in programming languages and database structures.

Chapter 9. Working with Categories Kinds of arrows and how they are defined in categories. Initial and terminal objects.

Chapter 10. Manipulating Objects in a Category Sums and products of objects and their properties (each of these two operations provides a monoid structure for a category). Examples of products and sums in categories of a different nature, with code examples. In this chapter, we also introduce equalizers and pullbacks. The notion of *pullback*, which anybody familiar with

the notion of `join` in SQL would easily recognize. An *equalizer* is a generalization of the idea of fixpoint. All these notions are introduced in a "point-free" style. The dual notions, *coequalizer* and *pushout*, are also discussed, with examples.

Chapter 11. Relations between Categories Categories form a category, where categories themselves are objects and arrows are called *functors*. Sums and products of categories are special examples of sums and products and are discussed as well.

Chapter 12. Relations between Functors Functors between two given categories also form a category. Its arrows are called *natural transformations*. We establish relationships between parallel functors. Then we take a look at pairs of functors pointing in opposite directions, and see how these can be related, with examples. *Adjoint functors* are introduced in this chapter.

Chapter 13. Cartesian-Closed Categories A Cartesian-closed category is a special kind of category for which additional structures are introduced. These categories represent the typed lambda calculus, where the category's objects are types.

Chapter 14. Monads Monoid-like properties are discovered when looking at an adjoint pair of functors, and a functor is built with such properties. This is called a *monad*, and we look at several examples. The notions of algebra and free algebra are studied, with examples. Another way to view monads is via Kleisli categories.

Chapter 15. Algebras and Kleisli A monad is defined by its category of algebras, or, alternatively, by its Kleisli category. In this chapter, the category of algebras and the Kleisli category are defined, and examples that demonstrate these categories for certain popular monads are presented.

Chapter 1. Functions

General Ideas

You either know, or you think you know, what a function is. Definitions and opinions vary. One definition represents a function as a set of pairs (x, y), with certain properties. This definition is popular among traditional mathematicians, and it causes a lot of confusion when you want to apply it in practice. While this definition is still traditionally used in textbooks, we will not use it. Instead, we will be much more vague, and in this vagueness we will be much more precise.

Given two types, A and B, a function from A to B is anything that, given an instance of A, produces an instance of B. Note that if there is more than one occasion when the same function is applied to the same value, the resulting instance is always the same.

There are many ways to specify a function. One is the set-theoretical way, mentioned above, which is similar to how we define a *map* literal, by just listing key/value pairs. However, to do this in code and to be able to practically list all such pairs, we need a type A that is finite (and rather small), demonstrated in the Scala code below:

```
val myMap: Map[A,B] = new Map(a1 -> b1, a2 -> b2, a3 -> b3)
```

Above, we are defining a value named `myMap`, which has a type `Map[A,B]`, and which has three key/value pairs.

If `a1`, `a2`, and `a3` are all possible values of type `A`, this is a legitimate way of defining a function. If not, this is actually a partial function, since for some `a` the function is not defined.

A more general way to define a function would look like this:

```
val myFunction = (x: X) => {
  /* some calculations that produce a value y of type Y */
  y
}
```

Here is a simple example:

```
val s2c2 = (x: Double) => (sin(x)*sin(x) + cos(x)*cos(x))
```

This function *maps* values of type `Double` to values of type `Double`.

When defining a function using an algorithm or using other functions, a programmer would expect that the function's value is eventually provided by a calculation. In real life, we know that this is not always the case. Similarly in math, a definition may look good, and, in test cases, the function provides a value "for all reasonable arguments." Nevertheless, we may not

know whether a result can be produced for every possible argument. The length of a Collatz sequence is an example of such a function.

The sequence starts with a certain (arbitrary) natural number and continues like this: if the current number is even, the next one is the current number divided by two. If the current number is odd, the next number is the current number multiplied by three, plus one. The question is, will this sequence ever reach number 1?

Here is a JavaScript version of this calculation:

```
function collatz(n) {
  let nSteps = 0
  while (n > 1) {
    n = n % 2==0 ? n/2 : (3*n+1)
    nSteps++
  }
  return nSteps
}
```

This function counts the number of steps it takes a Collatz sequence to reach the number 1 (where it stops).

A pretty simple function, but we don't know if it produces a result for every possible integer number `n`. It is not known yet whether the sequence ever ends. If it's infinite for a given `n`, the function does not give a result—meaning, it's not even a function.

Now, we will look at some examples and *counterexamples*.

Example 1. SQL `select firstname from users where id=?;` This SQL statement can be considered as specifying a function, where `id` is a parameter. The problem with this "function" is that not every `id` gives us a result. We can only use a collection of known `id`s and expect that for an acceptable `id`, we will always have a name as a result.

Counterexample 2. SQL, Not a Function `select * from users where firstname=?` is not a function returning a `User` record. For a single first name, we can get more than one result or fewer. It could be turned into a function if its codomain is defined as a collection of users. We can always have a collection, even if the collection is empty.

Counterexample 3. Dependency but not a Function $x^2 + y^2$ is not a function in x, since it also depends on y. We can either declare it a function of two parameters, x and y, or fix a certain value for y, call it y_0, and then $x^2 + y_0^2$ becomes a function of just one parameter, x.

Counterexample 4. Not a Function: Result Varies In Java, static methods can qualify as functions. Unfortunately, a function that takes no arguments and returns different values cannot. For example, `System.currentTimeMillis()` is not a function because its value changes independently of any parameter we pass (no parameters are passed). For some obscure reason, it is customary in programming practice to treat these things as functions. Actually, this is just the name of a channel from which input originates. It is not a function but an input parameter for our code, and our code is a function that depends on this parameter.

The same argument applies to `random()`. Even if it depends on something, we don't know what that is. We just get its value as an input. *We are a function*—that means the code that calls `random()` *is* the function that depends on the values provided by `random()`.

Now let us introduce the proper terminology.

Main Definitions

Definition: Domain and Codomain

Given a function $f : A \to B$, we call A the *domain* of f and B the *codomain* (some books use the term *range*) of f.

Note that there is no requirement that f should be able to produce any given value of type B. We just know that all the values f produces belong to type B, and that's how B becomes a codomain.

Take, for example, a *constant* function $c : A \to B$ that produces the same value b_0 for any value of A. This fact, which may be unknown when the function is defined, does not make the singleton $\{b_0\}$ its codomain. If we declare that our function c has a codomain B, that is all that we know at the moment at the point of definition. The fact that it only produces one value may never be known.

Example 5. $sin(x)$ First, let me remind you of the traditional notation. $\mathbb{R}$ is the type of real numbers, which can be represented in your code in various ways.

$sin : \mathbb{R} \to \mathbb{R}$ is a regular sine function. Of course, with a real number parameter, it only yields values in the range $[-1.0, 1.0]$; but its codomain is defined as $\mathbb{R}$.

Definition: Identity Function

The *identity function* $id_A : A \to A$ is a function that always returns its argument.

Here is the identity function defined in Scala, with type parameter `A`, because all identities are different:

```
def identity[A] (a:A) = a
```

The identity function, while it does not do anything, has a special property: if you *compose* it with any other function f, you will get the same function f as a result.

But first, composition should be defined, and we should talk about what it means when two functions are the same.

Definition: Constant Function

Given a domain A, a codomain B, and a value $b \in B$, a constant function $const_b : A \to B$ is a function from A to B such that $const_b(a) = b$ for all a.

Definition: Composition

Given a function $f : A \to B$ and a function $g : B \to C$, we define their *composition*, $g \circ f$, as a function $h : A \to C$ such that $h(a) = g(f(a))$ for all possible values $a : A$.

Given two functions, we have introduced a third one, and we have a reason to believe that if f and g are functions, h exists as well and is a function. We define it using f and g.

Now, what if we already have a function $h : A \to C$ that has this property, $h(a) = g(f(a))$, for all $a : A$? Or what if we have two functions, $f_1 : A \to B$ and $f_2 : A \to B$, and we know that $f_1(a) = f_2(a)$ for all $a \in A$? In this case, we declare that these two functions are equal, although one of them might take hours to calculate and another is just a constant, similar to the function `s2c2` above.

This also means that two clearly different expressions may define the same function. It doesn't matter how we produce the result. As long as it is the same, it is considered the same function.

Now back to composition. Using the definitions, one can see that for any $f : A \to B$, $id_B \circ f = f \circ id_A = f$. One can also see that, by just comparing expressions, $(h \circ g) \circ f = h \circ (g \circ f)$, where h is some function from C to D. This property of composition is called *associativity*. We'll discuss this again in the next chapter.

Here are a couple of examples.

Example 6. Composition of salary and $sin(x)$ Define f as `select salary from Employees where id=?`, and $g = sin : \mathbb{R} \to \mathbb{R}$. The crazy thing we are going to do here is calculate the sine of an employee's salary—which is a composition of the two functions.

We can express it all in SQL: `select sin(salary) from Employees where id=?`.

Example 7. Composition in JavaScript In JavaScript we can easily implement composition of two functions like this:

```
const compose = (f, g) => (x) => g(f(x))
```

In case you are not familiar with JavaScript, the line above means the following: a constant value called compose is defined; this value is a function; this function takes two arguments (`f` and `g`) and returns a value `(x) => g(f(x))`, which is a function taking an `x` and returning `g(f(x))`.

Special Classes of Functions

Monomorphism

Definition A function $f : A \to B$ is called an *injection*, or a *monomorphism*, or just a *mono*, if from $f(a_1) = f(a_2)$ it follows that $a_1 = a_2$. A notation for mono is: $f : A \rightarrowtail B$.

There is an alternative definition, saying that for different arguments an injection should return different results: if $a_1 \neq a_2$, then $f(a_1) \neq f(a_2)$.

While the way this latter definition is expressed makes it sound more powerful, we are within the territory of negative sentences, and this is known to be less powerful, although on many occasions it is the same thing.

Informally, a mono does not merge values. So, a constant function will not be a mono, unless its domain consists of just one value. The identity function, on the other hand, is a mono.

Now take a look at a couple of examples.

A good example of a mono is an inclusion:

$f : A \to B$, where $f(x) = x$

If A is a subtype of B, in the sense that every instance $a : A$ is also an instance of B, that is, $a : B$, this is an *inclusion* of A into B. Since inclusion does not make different objects equal, this is a monomorphism.

Example 8. Make a Function Injective Even if the function is not injective, we can restrict the function to a smaller domain and make it injective: $sin : [0.1, 0.2] \to \mathbb{R}$ A regular sine function restricted to a segment of arguments between 0.1 and 0.2 is injective.

Counterexample 9. Non-Injective Function `select firstname from users where id=?` This SQL statement can be considered as specifying a function, if we pass `id` as a parameter. We cannot expect this function to be injective, since in many cultures people don't have much of a choice for the first name, and we will see names repeated frequently.

Counterexample 10. Another Non-Injective Function $sin : \mathbb{R} \to \mathbb{R}$ is definitely not injective: Any value in the range $[-1.0, 1.0]$ is repeated an infinite number of times.

Monomorphisms have a nice feature: a composition of two monos is also a mono. Why? Suppose $g(f(a_1)) = g(f(a_2))$. Then, since g is a mono, $f(a_1) = f(a_2)$. Since f is also a mono, $a_1 = a_2$. QED.

QED stands for "Quod Erat Demonstrandum"—"that which was to be proven".

The opposite is also true: if $g \circ f$ is a mono, then f is a mono, too. Proving this could be a good exercise. Note that we cannot guarantee in this situation that g is also a mono. It is an exercise for the reader to provide an example.

Dual to the notion of monomorphism is the notion of epimorphism.

The term *dual* comes from Category Theory and implies that all arrows in a definition can be reversed.

Epimorphism

Definition A function $f : A \to B$ is called a *surjection*, or an *epimorphism*, or just an *epi*, if for each $b : B$ there is an $a : A$ such that $f(a) = b$. A notation for epi is this: $f : A \twoheadrightarrow B$.

Note In this definition, epimorphism and surjection are treated as synonyms, but are not exactly synonymous. The distinction between the two will be covered later on, but for now assume they are synonyms.

Informally, an epi covers the whole range, and not necessarily once. For example, the function $sin(x)$, defined on the whole class of real numbers, with the segment $[-1, 1]$ as its codomain, covers the segment infinitely many times.

It is clear that an identity function is an epi.

You can also see that a composition of two epis is an epi. Why?

Take a function $f : A \to B$, a function $g : B \to C$, and their composition $h = g \circ f$.

If g is an epi, for any $c : C$ there is a $b : B$ such that $g(b) = c$. Fine, but f is an epi too. So there is at least one $a : A$ such that $f(a) = b$. Taking these two facts together, we conclude that $h(a) = c$. QED.

Also, if $g \circ f$ is an epi, then g is an epi too. Prove it!

Example 11. Surjection Notation: $\mathbb{N}$ is the type of Natural Numbers.

Define a function $f : \mathbb{N} \to \mathbb{N}$ like this: $f(x) = x/100$. Use integer division. Any natural number m in the codomain $\mathbb{N}$ is $f(100 * m)$, hence this function is a surjection.

Counterexample 12. Not a Surjection Define a function $f : \mathbb{N} \to \mathbb{N}$ like this: $f(x) = x + 100$. Numbers under 100 cannot be represented as $f(x)$ for any x.

For a function that is both a mono and an epi there is a special term: *bijection*.

Bijection

Definition A function $f : A \to B$ is called a *bijection*, or *one-to-one correspondence*, if it is both an injection and a surjection (i.e. mono and epi). That is, every element in B is the result of applying f to some unique element in A.

The identity function is always a bijection, and a composition of two bijections is a bijection. One may ask, how about the inverse? Does a bijection always have an inverse function (which we have not defined yet)? Not necessarily. In set theory it does, but set theory makes too many interesting assumptions regarding the world, and not all of them work in programming practice.

Example 13. Caesar Cipher Define a function on lower-case Latin letters: $f : \mathbb{N}_{26} \to \mathbb{N}_{26}$ like this:

```
f(c) = ('a'.toInt + (c.toInt - 'a'.toInt + 5) % 26).toChar
```

This function maps `"from fairest creatures we desire increase"` to `"kwtr kfn-wjxy hwjfyzwjx bj ijxnwj nshwjfxj"`. Can you find the inverse to this function?

Example 14. Bijection between $\mathbb{N}$ and $\mathbb{Z}$ Define a function $f : \mathbb{N} \to \mathbb{Z}$ like this: $f(x) = ((i + 1)/2) * (i\%2 * 2 - 1)$. If you don't see what kind of function it is, here is what it does: it maps odd numbers to positives and even numbers to non-positives, thus covering the whole $\mathbb{Z}$. As an exercise, you can verify that this is a bijection.

Isomorphism

Approximately, an isomorphism is an invertible function. *Invertible* means that the function has an inverse. Before defining an invertible function, we should define what an inverse function is.

Definition: Inverse Function Given a function $f : A \to B$, its inverse, f^{-1}, is a function $B \to A$ such that:

$f \circ f^{-1} = id_B$ and $f^{-1} \circ f = id_A$.

We can see that an inverse is unique. Suppose g and h both satisfy the condition of being an inverse of f. Then $g = g \circ id = g \circ (f \circ h) = (g \circ f) \circ h = id \circ h = h$. QED.

Also note, f is the inverse of its own inverse.

Definition: Isomorphism A function $f : A \to B$ is called an *isomorphism* if it has an inverse.

An isomorphism must be a bijection, that is, both an epi and a mono.

First, is it a mono? If we compose f with f^{-1}, we get an identity function, and an identity is a mono. We know that if a composition of two functions is a mono, then the first one is a mono, as well.

Similarly, a composition of f^{-1} followed by f is an identity function, hence it is an epi, and the second one, f, must also be an epi. We have a bijection.

Since for a bijection f, for each b there is a unique a such that $f(a) = b$, why not take this as the definition of a new function that would serve as an inverse to f? The problem is, we may know that such an a exists, but at the same time have no clue how to find it. What kind of function would it be if we cannot get its value for a given argument? It is similar to the real life situation of a suspect that may be hiding too well from the police, even though there is no doubt the suspect exists and is unique. Of course, we could apply the "axiom of choice" from set theory—but could we, really?

The axiom of choice is an axiom of some set theories that, given a set of non-empty sets, allows one to build a set of samples from each of these sets. There are many equivalent formulations of the axiom of choice, and also alternatives that make a set theory rather different from the kind of theory that is traditionally used in general-purpose mathematics. For instance, there is a version of AC that involves two gamblers. One "throws" a set and another chooses an element in the set. The axiom consists of stating which one wins or whether one wins all the time.

When you write a program, you use some libraries. Suppose one such library you use has a *sin* function. *sin* is not bijective, but if we restrict it to $sin : [-\pi/2,\ \pi/2] \to [-1,\ 1]$, we can see it

is bijective. Still, if *arcsin* is not present in our library, we may have a hard time figuring out how to represent the inverse of sin.

Another argument: While formally providing an inverse for any bijection may work for sets, for richer structures—like ordered sets, monoids, and trees—functions are defined as preserving that additional structure. And if a function has one-to-one mapping on elements, this does not guarantee that the structure is preserved. In other words, we may not have an inverse. Instead of providing an example here, we will see later how this may be the case.

Definition: Fixpoint A *fixpoint* for a function f is a value x such that $f(x) = x$.

Example 15. Half as a Fixpoint For a real number $a : \mathbb{R}$, we can define its half as a fixpoint of the function $f(x) = a - x$. It is easy to see that $f(x) = x$ means $2 * x = a$, that is, $x = a/2$.

Example 16. Square Root as a Fixpoint Again, for a positive real number $a : \mathbb{R}$, we can define its square root as a fixpoint of the function $f(x) = a \ / \ x$. It is easy to see that $f(x) = x$ means $x^2 = a$, that is, both $x = \sqrt{a}$ and $x = -\sqrt{a}$ are fixpoints.

In general, the collection of fixpoints can be empty, as is shown next.

Counterexample 17. No Fixpoint Take a function $f(x) = x + 1$. This function has no fixpoints. Therefore, the set of its fixpoints is empty.

Restricting a Function

Suppose we have a function $f : A \to B$ and a subtype $A_1 \subset A$. Our function f is defined on the whole A. We can *restrict* it to A_1 by accepting only those parameter values that are in A_1. This new function is denoted as $f_1 = f|_{A_1} : A_1 \to B$.

Example 18. Restricting a Map Given (in Scala) `Map(”one”->1, ”two”->2,”three”->3)`, restricting it to `Set(”two”, ”three”)` gives us `Map(”two”->2, ”three”->3)`.

Note that while we are changing the function's domain, we are not changing the codomain of the function.

Example 19. Making sin Injective In one of the previous examples, we restricted $sin : \mathbb{R} \to \mathbb{R}$ to the segment $[0.1, 0.2]$. This is a good example. The restricted function is injective.

Restriction may look like a non-trivial operation, but it is actually a composition. For a subtype $A_1 \subset A$, we have an inclusion function, $i : A_1 \rightarrowtail A$. Restriction consists of this inclusion followed by the original function:

$f|_{A_1} : A_1 \to B = f \circ i$

Restricting a function by its values in the codomain is more challenging. We will need to add some definitions first.

Image of a Function

Definition

Given a function $f : A \to B$, the collection of all such $b : B$ that come from A via f (this can also be defined as $\{f(a) \mid a : A\}$) is called the *image of function* f and is denoted as $Im(f)$.

Depending on whether our universe can have such types, we can call it a type, or a set, or a collection. Generally speaking, there is no way to determine whether a $b : B$ belongs to $Im(f)$, except by trying all $a : A$ and checking to see whether $b = f(a)$ for one of these a.

Note, there is no symmetry here that would define a reverse image of f in A: by definition, every element $a : A$ maps to something in B.

While the existence, or practical calculability of an image may be under question, the notion of image is important in representing a function as a composition of an epimorphism followed by a monomorphism.

Example 20. Circle as an Image Use two functions, $sin(x)$ and $cos(x)$, and define a function $(sin, cos) : \mathbb{R} \to \mathbb{R} \times \mathbb{R}$.

This function, for each given real number x, gives us a point on a plane. The collection (set) of all such points is the image of this function, and this collection is a circle of radius 1.

Disambiguation

There are three terms that are frequently confused:

- *codomain*: the object (the set) where the values of a function are defined (e.g., for sin, the codomain is $\mathbb{R}$).

- *image*: the collection of all values of a function; not every codomain value has to be in the image, but every value of the image belongs to the function's codomain.
- *range*: a term used by different sources to denote either the image of a function or the codomain of a function; to avoid confusion, this term is not used in this book.

Suppose we have a function $f : A \to B$ and $Im(f)$, a subset (or a subtype) of B. We can introduce a function $e : A \to Im(f)$ defined by this formula: $e(a) = f(a)$. These two functions differ only by their codomains. A popular misconception is that these are the same function. I'll show you why they are not.

Take a function `findEmployee` defined as `select * from Employee where Name=?;` (assuming that `Name` is an enumeration containing some valid employee names) and another function, `salary` (assuming that employees receive double salaries). `findEmployee` has the signature `Name -> Employee`, and `salary` has the signature `Employee -> Double`.

Now suppose `Employee` is a subtype of `Person`. We can introduce another function, `findPerson: NamePerson`, that returns the same result as `findEmployee`. If we think these two functions are identical, we should be able to compose either of them with the `salary` function. But `salary` is only defined on `Employee`, not on `Person`—some `Person`s may not be getting any salary at all. As you see, we can compose `findEmployee` with `salary`, but we cannot compose `findPerson` with `salary`.

Decomposing a Function

Now that we have discussed disambiguation, we can go back to the image. We had a function $f : A \to B$, and we found a function $e : A \to Im(f)$ such that $e(a) = f(a)$. There is also an inclusion function,

$$i : Im(f) \rightarrowtail B$$

and the composition of the two, $i \circ e$, is equal to f.

We have built a decomposition of an arbitrary function into an epi followed by a mono. This is a pretty generic structure. All that's missing is the ability to build an image, and this ability is, generally speaking, not guaranteed.

On the other hand, this decomposition is unique *up to an isomorphism*: we can have more than one decomposition like this, but all the solutions will be isomorphic.

Example 21. Decomposing a Circle Now we return to the previous example, a function $f = (sin,\ cos)$ that produces a circle as its image. The function's domain is $\mathbb{R}$, and its codomain is $\mathbb{R} \times \mathbb{R}$. Its image is the unit circle and is traditionally denoted as s^1. When we

decompose it into an epi followed by a mono, the epi will be from $\mathbb{R}$ to s^1, and the mono will be the embedding i of s^1 into $\mathbb{R} \times \mathbb{R}$. The whole picture looks like this: $\mathbb{R} \overset{e}{\twoheadrightarrow} s^1 \overset{i}{\rightarrowtail} \mathbb{R} \times \mathbb{R}$

Predicate

A special kind of function worth noting is one that has logical values as the codomain. It is customary to call logical values *Boolean*. We will call them that occasionally, but remember that Boolean logic is not the only kind of logic known to humans, and we need to be prepared to be more generic. A better name for the type of logical values would be *Logical*.

Definition

A *predicate* is any function that takes Boolean values, $f : A \to Boolean$.

Example 22. A Predicate (in Scala)

```
isOdd = (n:Int) => n%2 == 1
```

Several operations are defined on logical values. We can extend these operations to be applicable to predicates. For example, given two predicates, $f : A \to Boolean$ and $g : A \to Boolean$, we can define $f \vee g : A \to Boolean$, $f \wedge g : A \to Boolean$, $\neg f : A \to Boolean$, and so on.

In programming languages predicates are usually applied to any collection, e.g., to lists in Python. In set theory, predicates are used to create new sets via *set comprehension*: $\{x \in S \mid p(x)\}$. With a set and a predicate, another set can be built from them.

It's also possible to move in the opposite direction: given a set S of values of type A, we can define a predicate $p(s) = s \in S$, or, in Scala,

```
pS(x): (X => Boolean) = S contains x
```

This predicate is true only for elements of S. Will the operation of taking a set comprehension for such a predicate produce the same set S? It will if we rely on set theory axioms.

Functions Having More Than One Parameter

So far, we've reviewed functions that have only one parameter. In real life, functions may have more. For functions taking several parameters, we can define the domain as well, but to do that, the notion of Cartesian product will be necessary (discussed later). Let's continue to deal with such functions without specifying their domains just yet.

A two-parameter function, $f(a, b) : C$, where $a : A$ and $b : B$, is general enough. The notation for such a function is $f : A \times B \to C$, which demonstrates the idea that its domain is the product of two types, $A \times B$. We will discuss products later, also.

Without defining a domain for such a function, we cannot label it as an epi or a mono. But we can consider parameters one by one. If we fix $a_0 : A$, we get a regular function $f : B \to C$, which may be different for each $a : A$. Essentially, we have a function, defined on A and taking values in functions $B \to C$.

This process of converting a two (or more) parameter function into a "chain" of one parameter functions is called *currying*.

The name comes from Haskell Curry, a mathematician, who picked it up from Moses Schönfinkel's works. The original idea was first published by Frege at the end of the 19th century.

Here is how currying looks in JavaScript:

```
const curry = (f) => (a) => (b) => f(a,b)
```

Haskell doesn't have multiple-parameter functions. All functions by default are in curried form. In Scala, on the other hand, you can explicitly curry or uncurry your functions if you feel like it.

Two special kinds of two parameter functions are worth noting.

Binary Relation

A special case of a two parameter function is where the result type is Boolean: $r : A \times B \to Boolean$.

This kind of predicate is called a *binary relation*. A binary relation is informally perceived as a relation between values of two types, A and B. The value of r is true exactly for those pairs (a, b) that are in the relation.

Example 23. Relational Database Consider a database where `A=Person` and `B=Company`, and there is a relation named `worksFor`. Such relations are usually stored in relational databases as a collection of pairs, e.g. `(person, company)`. This is similar to set comprehension for the set of all possible pairs:

```
def worksFor(person:Person, company:Company): Boolean =
company.name == person.job.companyName
```

Or like this:

```
def worksFor(person:Person, company:Company): Boolean =
 company.listOfEmployees contains person
```

In both cases the binary relation is between `Person` and `Company` types.

A popular notation for binary relations is *infix*: instead of writing $R(a, b)$, we write $A\ R\ B$. For the binary relation defined above, an expression would be `Person worksFor Company`.

In many modern programming languages, standard two-argument predicates are written in infix mode. Scala allows you to define your own two-argument functions that can be written in infix mode, and this form applies to binary relations as well. For instance, the expression: `Person worksFor Company` is legitimate in Scala.

Another example of a binary relation is a partial order—for example, in integer numbers, $A < B$. It could be rewritten as $<(a, b)$, which definitely looks less intuitive.

Binary Relation on a Single Type

If both parameters of a binary relation R have the same type, we can ask whether $a\ R\ a$ is true, and whether $a\ R\ b$ yields $b\ R\ a$ or not, etc. Here are some new terms for such properties.

Definition: Reflexive Relation

A binary relation $R : A \times A \to Boolean$ is called *reflexive* if for each $x : A$ we have $x\ R\ x$. If, on the contrary, $x\ R\ x$ is false for all x, the relation is called *antireflexive* or *irreflexive*.

For example, the relation $x < y$ is antireflexive, and the relation $x = y$ is reflexive.

Definition: Symmetric Relation

A binary relation $R : A \times A \to Boolean$ is called *symmetric* if for each $x, y : A$ such that $x\ R\ y$ we have $y\ R\ x$. On the other hand, if both $x\ R\ y$ and $y\ R\ x$ are true only in the case when $x = y$, the relation is called *antisymmetric*. A relation can be both symmetric and antisymmetric. In this case, we have the equality relation.

Example 25. Order The relation $x \leqslant y$ on natural numbers, $x : \mathbb{N}$, is antisymmetric, since from having $x \leqslant y$ and $y \leqslant x$ it follows that $x = y$.

Example 26. Equality Equality is a special kind of relation. It is defined as $x = y$ and can be represented as a diagonal in the rectangle consisting of pairs (x_1, x_2) where $x_1, x_2 \in X$. This relation is obviously reflexive.

Definition: Transitive Relation

A binary relation $R : A \times A \to Boolean$ is called *transitive* if for each $x, y, z : A$ such that $x\ R\ y$ and $y\ R\ z$ we have $x\ R\ z$.

For example, the relation between numbers $x < y$ is transitive, and the relation $x = y$ is transitive, too!

Relation	Properties
Equality	symmetric, reflexive, antisymmetric
$`<`\subset \mathbb{N} \times \mathbb{N}$	irreflexive, antisymmetric, transitive
$`\leqslant`\subset \mathbb{N} \times \mathbb{N}$	reflexive, antisymmetric, transitive
`isParentOf`	irreflexive, antisymmetric
‘Has same birthday (or birth date)‘	reflexive, symmetric, transitive

Equivalence Relation

Definition An equivalence relation is a binary relation R defined on a type A such that it is symmetric, reflexive, and transitive.

A natural case of equivalence is equality, and other examples can be found. For instance, in Java (and Scala), one can define the method `equals()`, which tells whether one value can be replaced with another. To ensure that the code behaves properly, this method should be symmetric, reflexive, and transitive. This method defines an equivalence relation. There is no reason to expect that it is an exact equality, except maybe for the simplest cases.

Binary Operations

Another special case of a two parameter function is where both parameters are of the same type, and the result is the same type: $Op : A \times A \to A$.

Programmers are familiar with plenty of examples of such functions: integer addition, real numbers multiplication, list concatenation, gluing two binary trees together, Boolean operations, etc. Some of these operations have nice properties, similar to the properties of binary relations. We will talk about them in great detail when we discuss monoids.

A binary operation can have special properties. Here are some of them:

Commutative Binary Operation

Given a binary operation $Op : A \times A \to A$, we call this operation *commutative*, if the following holds: for all a and b, $a\ Op\ b = b\ Op\ a$.

Addition and multiplication of numbers is commutative. Addition of matrices is commutative, but multiplication is not. You will find more examples of such operations in Chapter 2.

Associative Binary Operation

Given a binary operation $Op : A \times A \to A$, we call this operation *associative*, if the following holds: for all a, b, and c, the following holds: $(a\ Op\ b)\ Op\ c = a\ Op\ (b\ Op\ c)$.

Addition and multiplication of numbers is associative. The same is true for addition and multiplication of matrices. More examples can be found in Chapter 2.

Idempotent Binary Operation

Given a binary operation $Op : A \times A \to A$, we call this operation *idempotent*, if the following holds: for all a, $a\ Op\ a = a$. You will find examples of such operations in Chapter 2.

The union and intersection of sets are idempotent. Again, more examples can be found in Chapter 2.

Chapter 2. Abstractions of Algebra

Introduction

You have likely encountered monoids many times in life and programming practice, but you might not have recognized them. Here are some examples.

1. Integers with addition.
   - $(a + b) + c = a + (b + c)$
   - $0 + n = n + 0 = n$
2. Integers with multiplication.
   - $(a * b) * c = a * (b * c)$
   - $1 * n = n * 1 = n$
3. Strings with concatenation.
   - $(s1 \ ++ \ s2) \ ++ \ s3 \ = \ s1 \ ++ \ (s2 \ ++ \ s3)$
   - $"" \ ++ \ s \ = \ s \ ++ \ "" \ = \ s$
4. Lists with concatenation, for example:

$List(1, 2) + +List(3, 4) = List(1, 2, 3, 4)$

5. Sets with union, for example:

$Set(1, 2, 3) \ \cup \ Set(2, 3, 4) \ = \ Set(1, 2, 3, 4)$

Do you see a common pattern? We have some type of data, a binary operation, a special instance of the type, and certain rules. This common pattern is called a monoid.

Formal Description

First, let's review the notation from the previous chapter. We write $a : A$ meaning that A is a type, and a is an instance of this type, whatever it might mean in type theory.

Now, I will provide a strict formal definition of a monoid.

Definition: Monoid

Given a type T, a binary operation $Op : T \times T \to T$, and an instance $Z : T$, with the properties specified below, the triple (T, Op, Z) is called a *monoid*. Here are the properties:

- *Neutral element*: $Z\ Op\ a = a\ Op\ Z = a$
- *Associativity*: $(a\ Op\ b)\ Op\ c = a\ Op\ (b\ Op\ c)$

Some examples were provided in the introduction above, e.g. $(String, ++, \texttt{""})$—strings, concatenation, and the empty string. Below are more examples.

Example 1 $(Boolean, \&, true)$. Boolean conjunction is associative, and $true$ is a neutral element for conjunction (which may also serve as a definition of *truth*).

Example 2 $(Boolean, \|, false)$. Boolean disjunction is associative, and $false$ is a neutral element for disjunction.

Example 3 $(\mathbb{N}, max, 0)$: for the set of natural numbers, use the operation of finding the maximum of two numbers, and 0 as a neutral element. We now have a monoid.

Example 4. Monoid of Endomorphisms Given a set A, the set of all functions $f : A \to A$ form a monoid, if we use composition as the operation, and the identity function as the neutral element. Composition is known to be associative. Functions from a set to itself are called *endomorphisms*.

Counterexample 5 $(\mathbb{N}, min, ???)$ If, on the set of natural numbers, we use as the binary operation the action of finding the minimum of two numbers, we have a structure that is almost a monoid (it is called a *semigroup*). For a monoid, we would need a neutral element. There is no such thing. It is missing (remember, infinity is not a number). So this is not a monoid.

Counterexample 6 Not every binary operation in the world is associative. For example, the operation of taking an average is not associative: $avg(10, avg(30, 50)) \neq avg(avg(10, 30), 50)$. So $(\mathbb{R}, avg, ?)$ is not a monoid. Also, there is no neutral element.

Counterexample 7 A typical case of non-associative operation is forming tuples. Whatever the type of data, A, $(a, (b, c))$ is not the same as $((a, b), c)$. One possible exception is Quine set theory, where $a = (a, a)$. And of course, there is no neutral element.

Counterexample 8 This is similar to counterexample 7, but instead of tuples we will use binary trees. Two trees, a and b, are concatenated by having a new root and turning a and b into subtrees of that new root. An empty tree would be a neutral element, if we expanded the definition of

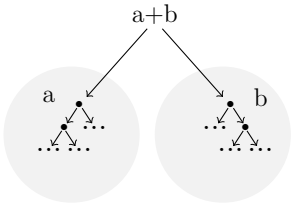


Figure 1: *Binary operation on trees.*

concatenation. In the picture below, some nodes of the tree have labels. These labels are only for reference and are not a part of the tree structure.

This operation is not associative:

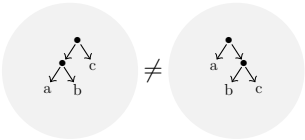


Figure 2: *No associativity.*

Is it Useful?

Software engineers love to ask these questions: "What is it good for?", "How can it be used?", "Why should I learn anything?" My positive answer to these questions is, *Yes!* With an associative operation, you can reorganize a calculation in any order. Suppose we have the following expression:

$(a1\ Op\ (a2\ Op\ (a3\ Op\ (a4 \ldots))))$

Or, the other way around,

$(\ldots (a1\ Op\ a2)\ Op\ a3)\ Op\ a4) \ldots$

In Scala, there is a *fold* notation for this expression:

```
(Z /: as) (x Op y)
```

where `as` is a list of values of type `A`, `Z` is a value that serves as a neutral element, and `Op: (A, A) => A` is the binary operation.

All calculations are done sequentially, because we rely on results from the previous calculation. In the case of associativity, we don't have to rely on a specific order of calculations. We can regroup the calculation and run it in parallel, as in the following picture:

$$\left.\begin{array}{rcl} b_1 & = & a_1\ Op\ a_2\ \ ...\ \ Op\ a_{10} \\ b_2 & = & a_{11}\ Op\ a_{12}\ \ ...\ \ Op\ a_{20} \\ & ... ... & \\ b_{10} & = & a_{91}\ Op\ a_{92}\ \ ...\ \ Op\ a_{100} \end{array}\right\}\ b_1\ Op\ b_2\ \ ...\ \ Op\ b_{10}$$

Mappings Between Monoids

We have defined monoids, but it is not enough. Next, we will study relations between monoids. If we just have types, without any additional structure, any function $f : A \rightarrow B$ is good for mapping instances of one type to instances of another. But if we have monoids, it is not enough. Any mapping from one monoid to another should also preserve the monoid features.

Now, we will formally specify what it means to preserve the structure of monoid.

Definition: Monoid Mapping

For two monoids, (A, Op_A, Z_A) and (B, Op_B, Z_B), we define a *mapping from one monoid to another* (aka *monoid homomorphism*) as a function $f : A \rightarrow B$ such that

- $f(Z_A) = Z_B$
- $f(x\ Op_A\ y) = f(x)\ Op_B\ f(y)$

Example 9 The exponent function maps sums to products:

$exp : (Double, +, 0) \rightarrow (Double, *, 1);\ exp(a + b) = exp(a) * exp(b)$

Example 10 Length of a string:

$length : (String, ++, \texttt{""}) \rightarrow (Int, +, 0)$

Example 11 Sum of a list of integers:

$sum : (List[Int], ++, Nil) \rightarrow (Int, +, 0)$

Counterexample 12 The following function is not a mapping of monoids $(Int, +, 0)$:

$f(n) = n * 5 + 1$

As you can see, neither addition nor the neutral element is preserved.

Special Kinds of Monoids

Free Monoids

If you do not see enough monoids, you can build one out of anything you have. From any A, we can build a monoid from it. In mathematics, this monoid is known as A^* and is called a *Kleene Star*. In programming languages, it is called `List`—to be more precise, `List[A]`, in Scala notation, a type of lists of elements of type `A`.

Mathematically, A^* can be defined either as $1 + A + A^2 + A^3$ ... or as a root of an equation,

$$X = 1 + A \times X$$

If X has this property, we can substitute X in the right part of the equation and see that

$$X = 1 + A \times X = 1 + A \times (1 + A \times X) = 1 + A + A \times A + A \times A \times X...$$

We can solve this equation:

$$X = 1 + A \times X \Leftrightarrow X \times (1 - A) = 1 \Leftrightarrow X = \frac{1}{(1 - A)} = 1 + A + A^2 + A^3 ...$$

If someone thinks that division or subtraction of types is not allowed, not so long ago extraction of the square root of -1 was also unthinkable. Later, imaginary numbers became a natural part of math. Similarly, operations on types may become a natural part of type theory.

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

In Haskell, the colon character, `':'`, is the operation of appending an element to a list, and `Nil` denotes an empty list.

Unfortunately, Scala is not as good at defining types via recursion yet.

What's important about lists (i.e. Kleene stars or free monoids) is their universality among other monoids. To define a monoid mapping from a monoid A^* to a monoid B, it is enough to define such a function on elements of A (that is, on single-element lists). In addition, any such function from A to B provides a monoid mapping.

Now let us see how it works, in terms of lists.

Given $f : A \to B$, we define $f' : List[A] \to B$ like this:

$f'(Nil) = Z_B$

$f'(x :: xs) = f(x) \ \ Op_B \ \ f'(xs)$

In this example, Nil denotes an empty list, and $::$ is the operation of appending an element to a list.

Did you notice that f' is a monoid mapping? Does it preserve the neutral element? Yes. Does it preserve composition? We can prove this as well, but our plans do not involve proving anything. As an example, try to find out whether the operation preserves composition and maps the neutral element to a neutral element.

Also, f' on single-element lists is "the same as" f, so we can retrieve f from f' without loss.

This operation, having an f and producing f' as above, is known as `foldMap` in functional programming languages (e.g., Haskell, Scalaz library, Scala Cats library).

Example 13 Suppose we have the following in Scala:

```
object WeekDay extends Enumeration {
  val Mon, Tue, Wed, Thu, Fri, Sat, Sun = Value
}
```

(Ignore the technicalities; a type consisting of seven possible values has been defined here).

Now we define the following function in Scala:

```
val f: (WeekDay => Int) = Map(
  Mon->1, Tue->1, Wed->1, Thu->1, Fri->1, Sat->0, Sun->0
)
```

Having a monoid $(Int, +, 0)$, we can automatically extend this function to the function $List[WeekDay] \ \rightarrow \ Int$, which counts the number of working days.

Deterministic State Machine

How is a deterministic machine usually defined? The definition involves:

- An input alphabet A;
- A collection of states S; and
- A transition function: $f : A \times S \ \rightarrow \ S$

For a transition function $f : A \times S \ \rightarrow \ S$, we can use its curried version, $f' : A \rightarrow (S \rightarrow S)$. This function maps our alphabet to a monoid described above, a monoid of endomorphisms on S.

Words over alphabet A form A^*, or $List[A]$, the free monoid already described above. We also have $f'' : A^* \to (S \to S)$, a mapping between two monoids.

Example 14 Consider an input alphabet to be $\{'a', 'b'\}$, and suppose our machine should check that there are the same number of $'a$'s and $'b$'s in the input. As a state space, we use `Int` set, all integers. The traditional representation of the transition function would look like this:

```
def f(in: Char, s: Int) = in match {
  case 'a' => s+1
  case 'b' => s-1
}
```

When we turn it into a curried version, we will have

```
def f1(in: Char) = in match {
  case 'a' => fa
  case 'b' => fb
}
```

where two functions, `fa` and `fb`, are defined as

```
val fa = (n:Int) => n+1
val fb = (n:Int) => n-1
```

In Scala, composition of functions is written as `andThen`: e.g. `fa andThen fb`.

We now have functions defined on elements of the alphabet, and we need to extend them to the whole monoid, that is, to the lists of letters. We will obviously have "$abba$" $\mapsto$ $fa \circ fb \circ fb \circ fa$.

The problem is, how do we write it? We could start with a list of characters, that is, a string `s`, and map the characters into functions, via `f1`, obtaining a list, `List[Int=>Int]`. Knowing that we have a monoid (of endomorphisms), we don't need to keep a list of functions, we can just fold them together. Since we are using Scala, we use the Scala fold.

Putting all of this together, we can properly define `f2:String => (Int=>Int)` like this:

```
def f2(input:String) = (id /: input) { (fs, c) => fs andThen f1(c) }
```

This code likely needs an explanation. We define a function that takes a parameter of type String, named `input`. The function takes an identity function, called `id`, scans the whole input, and produces a composition. E.g., for `abba`, it produces:

$f1('a') \circ f1('b') \circ f1('b') \circ f1('a') \circ id$

This resulting function is a transition function that corresponds to the input string `abba`.

Commutative Monoids

Not all monoids are created equal. We can find classes of monoids that have additional properties, reflecting the nature of their operations. In this section, we will discuss some of those properties.

You have probably noticed that $'+'$ behaves differently for strings than for numbers. For numbers, $a + b = b + a$; for strings, it almost never does. So, to be specific, we need to define what we are talking about.

Definition The definition of commutativity (see Chapter 1) does not involve any monoids and is good for any binary operation. In the specific case involving a monoid, we call this monoid *commutative* if its binary operation is commutative.

We know that some monoids are commutative and some are not: 2+3 = 3+2; "*hello*"+"*world*" ≠ "*world*" + "*hello*".

One might expect that not every monoid function would be good for commutative monoids, but this is not the case. We do not need a special tyoe of function. Commutativity is a property of the binary operation. As long we have a function from one monoid to another, $f : A \to B$, either of the two monoids can be commutative, or both. Of course, if $a_1\ Op\ a_2 = a_2\ Op\ a_1$, then:

$f(a_1)\ Op\ f(a_2) = f(a_1\ Op\ a_2) = f(a_2\ Op\ a_1) = f(a_2)\ Op\ f(a_1)$

So there is some commutativity in B, but it does not necessarily make the whole B commutative.

Can free monoids be commutative? The short answer is *no*. Instead, we can introduce a new kind of free monoid, the *free commutative monoid*.

How can we get one? Start with a free monoid (on a type A), that is, a monoid $List[A]$. To make it commutative, introduce a special kind of equality (strictly speaking, it is an equivalence, not an equality), where two lists that differ only in the order of elements are considered to be the same. The easiest way, in terms of code, is to have "sorted lists." In the Java/Scala world, this monoid is called `SortedList[A]`.

For example, for a list of characters "*abracadabra*", forget about the concatenation order. Sort the list, obtaining "*aaaaabbcdrr*".

We can build it for any type `A`. But note, concatenation will be different from list concatenation. After concatenating two lists, we will need to sort the new list, or, if these two lists were sorted, we could just merge the two sorted lists.

As you can see, lists may be not the right representation, so shouldn't we look for a better representation? All we need is the number of occurrences for each element of the alphabet. This

structure resembles a set, but we need to count how many times each element appears. It is called a *multiset*, or a *bag*.

Example 15, in JavaScript

```
{
  "partridges in a pear tree" : 1,
  "turtle doves" : 2,
  "French hens" : 3,
  "calling birds" : 4
}
```

To count all of the birds, we don't need to know on which day they were appended to the multiset.

What is the binary operation that makes multisets a monoid? It is a multiset union, where two multisets are joined, adding the counts. The empty multiset will be our neutral element.

Did you realize that this operation of joining multisets is commutative?

How About Sets?

A regular set is pretty similar to a multiset, except that we do not count occurrences; an element is either there or not. The main distinction is this: a set A joined with itself is still the same set A: $A + A = A$. This property has a name.

Definition An element x of a monoid (A, Op, Z) that has the property $x\ Op\ x = x$ is called *idempotent*.

Obviously, a monoid's neutral element is always idempotent.

A monoid is called *idempotent* if every element is idempotent. See the end of Chapter 1 where idempotence is defined for any binary operation.

In the monoid of sets with a union operation, every element (that is, every set) is idempotent.

A simpler version of idempotent monoids is Boolean logic—it has two idempotent monoids. One is $(Bool, \vee, false)$, and the other one is $(Bool, \wedge, true)$. We will focus more on logic later.

Removing Constraints

A monoid has two properties: associativity and a neutral element. What do we get if we remove these properties?

Semigroups

If we drop the requirement of having a neutral element in the definition of a monoid, we have a *semigroup*. There are plenty of examples. First, any monoid is a semigroup. Here is a non-trivial one:

Example 16 Take a set of all integers above 17 with addition as the binary operation: $(\{n : Int \mid n > 17\}, +)$. If 0 were in this set, we would have a monoid, but since it isn't, it's just a semigroup.

Notice that we still have associativity in semigroups. And what if we drop associativity, too? We get a *magma*.

Magmas

A *magma* is a binary operation. You may think that this does not make much sense, but it actually does. Associativity is good to have, but it's not always available. All semigroups, including all monoids, are magmas, but there are other kinds, too. Syntax trees and binary trees are magmas. If you represent JSON or XML as a tree, associativity is not there, but it's possible to represent them as magmas.

Example 17 A binary tree is a magma. Given two binary trees, we can build another tree out of them by attaching a new root.

Example 18 The operation of forming a pair gives us a magma. Pairing is a binary operation, and it is not associative.

Chapter 3. Partial Orders, Graphs, and DAGs

In this chapter, we continue studying some simple and useful mathematical structures that you may encounter in your programming practice.

Partial Order

A *partial order* is also known as a *poset*. Poset means *partially ordered set*. Don't let the word "set" confuse you, as we are not talking about sets in the mathematical sense. As programmers, we deal with types. But the correct term is still "partial order."

Definitions

Definition: Partial Order Given a type A, a partial order on an object ("set") A is any transitive, antisymmetric binary relation on A. There are two kinds of partial order: strict and non-strict. A *strict* partial order is a partial order that is irreflexive. A *non-strict* partial order is a partial order that is reflexive.

A strict partial order is usually denoted as $a < b$. This expression is the same as saying that the pair (a, b) belongs to this relation. A non-strict partial order is usually denoted as $a \leqslant b$.

Alternative Definition Given a type A, a *non-strict partial order* on A is any transitive binary relation R on A for which $(aRb \wedge bRa) \Rightarrow a = b$. The notation for such a relation is $a \leqslant b$. The advantage of this version of the definition is that the rule of the excluded third is not involved. Further on, we will be using this second definition. Be aware that opinions on this topic vary.

Example 1 Given an arbitrary set $\{a, b, c\}$, we turn it into a partial order just by saying that it is a partial order. No elements are comparable, except with themselves. This kind of partial order is called *discrete*.

Definition: Discrete Partial Order A partial order $(A, \leqslant)$ is discrete if the relation $\leqslant$ only consists of the diagonal: $x \leqslant x$ for all x (see chapter 1 for the role of the diagonal in the definition of reflexive relations).

Example 2 Again, take an arbitrary set $\{a, b, c\}$, and declare $a \leqslant b$. That's in addition to the diagonal $x \leqslant x$, and it is enough to define a partial order. c can only be compared to itself.

Example 3 Start with the previous example, and add one more arbitrary relation, $b \leqslant c$. Now we are in trouble. This new relation is not a partial order anymore, and we need transitivity. From $a \leqslant b$ and $b \leqslant c$, it should follow that $a \leqslant c$. In the case of this example, having $a \leqslant c$ would be enough, and we would have a legitimate partial order.

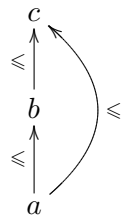


Figure 3: *A partial order with elements that are all comparable.*

If, instead of $b \leqslant c$, we added $a \leqslant c$, then we would have a different example, and there would be nothing to apply transitivity to.

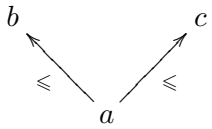


Figure 4: *A partial order where two elements are not comparable.*

Example 4 Strings, with alphabetic order. Alphabetic order in a string depends on how to compare any two characters. If we are dealing with natural languages, the order of characters depends on a) language, b) year, c) country, and d) the prevailing point of view of local linguists at a given moment in time. It also depends on how a character is defined in a particular language: "ll" or "nh" may be just one character in one language but two characters in another. The same two strings can compare differently in different languages, in different countries, and at different times.

A partial order is called *partial*, because it is not necessary for every two elements to be comparable. If either $a \leqslant b$ or $b \leqslant a$ is required, such an order is called total, or linear.

Definition: Linear Partial Order A partial order is called a *linear* or *total* order if every two elements are comparable: for all a, b, we have $a \leqslant b$ or $b \leqslant a$.

In set theory, one of the axioms is equivalent to the requirement that we should be able to define a linear order for every set. The axiom only states the existence of a linear order, but it does not tell us how to produce such an order.

In a partial order, we can define least upper bound (lub) and greatest lower bound(glb) functions. The value of lub and glb does not have to exist for all pairs.

Definition: Greater Lower Bound, Least Upper Bound In a partial order A, $glb(a, b)$ is such a value c that $c \leqslant a$ and $c \leqslant b$, and for each $x : A$, if $x \leqslant a$ and $x \leqslant b$, it follows that $x \leqslant c$.

Similarly, $lub(a, b)$ in a partial order A is such a value c that $a \leqslant c$ and $b \leqslant c$, and for each $x : A$, if $a \leqslant x$ and $b \leqslant x$, it follows that $c \leqslant x$.

The values $glb(a, b)$ and $lub(a, b)$ don't always have to exist. For example, in $\{a, b, c\}$, with the order defined as $a \leqslant b$ and $a \leqslant c$, we can see that $a = glb(b, c)$, but there is no $lub(b, c)$.

If, for instance, we are dealing with natural numbers, $lub(a, b)$ is the same as $min(a, b)$, and $glb(a, b)$ is the same as $max(a, b)$.

Caveat In Chapter 2, we learned that for a partial order $(\mathbb{N}, \leqslant)$, the function lub gives us a monoid, $(\mathbb{N}, lub, 0)$. Generally, neither glb nor lub necessarily give us a monoid for a given partial order. First, neutral elements are not guaranteed, and moreover, the operations glb and lub don't even have to be associative. Look at this picture:

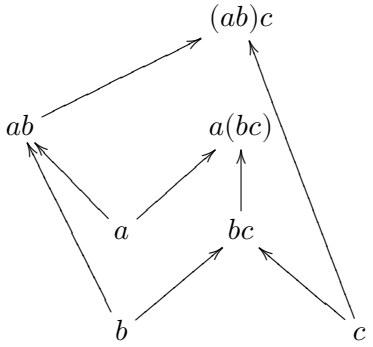


Figure 5: *A partial order where glb is not associative: $glb(glb(a, b), c) \neq glb(a, glb(b, c))$.*

A partial order may have the smallest element, usually called *bottom* and denoted as $\bot$. The largest element of a partial order, *top*, is denoted as $\top$. Neither of them need to exist for a given partial order. For natural numbers, $\mathbb{N}$, the bottom exists and is the number 0, but there is no top. If you look at Figure 5, you'll notice that $(ab)c$ is the top, but there is no bottom. Neither b nor c is below all other nodes.

Trees

Programmers are familiar with trees as data structures. There is a variety of definitions, in different languages and libraries. Most of these definitions of trees as data structures are implementation-specific, focusing on the ways of referencing one node from another, not on what a tree is as a generic data structure.

From a mathematical point of view, a tree is a special kind of partial order. Here, a tree is defined in a way that even covers the idea of continuous trees, in which there is a node between any two connected nodes. For instance, if we take $\mathbb{R}$ with numbers as nodes, there is no "parent node" for a given number, since for any two numbers, there is a number in between.

We will try to avoid the requirement of having a unique parent node for each node. One of the solutions consists of requiring that all nodes above any given node are comparable.

Definition: Tree A *tree* is a partial order with two additional properties. It has a top, called *root*, and for each x the collection of elements above this x, $\{y \mid y \geqslant x\}$, is a linear order.

Figure 6: *This is a larch tree. Root not shown.*

Example 5 A traditional binary tree from computer science, consisting of edges and nodes, can fit the definition above if we turn it into a partial order (parent $>$ child) and see that the chain of ancestors (all nodes above a given node) is a linear order.

More examples are shown in the pictures.

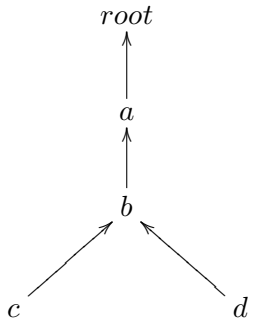


Figure 7: *A tree with four nodes.*

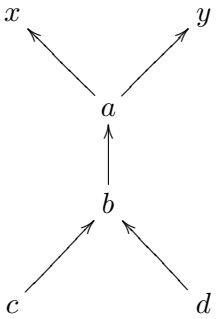


Figure 8: *Not a tree. Neither x nor y is a root.*

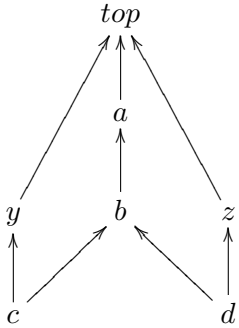


Figure 9: *Not a tree. c has incomparable ancestors, y and b. So, $\{z|z > c\}$ is not a linear order.*

Functions for Partial Orders

Definition: Monotone Function A *monotone*, or *order-preserving*, function $f : (A, \leqslant) \to (B, \leqslant)$ from partial order $(A, \leqslant)$ to partial order $(B, \leqslant)$ is such a function from A to B that preserves order, that is, if $x \leqslant y$ then $f(x) \leqslant f(y)$.

Example 10 $f : \mathbb{N} \to \mathbb{N}$ defined as $f(n) = n/10$. Does it preserve order? Yes. If $m \leqslant n$, then $m/10 \leqslant n/10$.

A monotone function does not have to map top to top or bottom to bottom. Here is an example:

Example 11 $f : [2, 4] \rightarrowtail [0, 5]$. This function inserts one segment of natural numbers into another, mapping numbers to themselves. f preserves the order, but it preserves neither the top nor the bottom. The top of $[2, 4]$ is 4, and the bottom of $[2, 4]$ is 2. Neither is a top or a bottom in $[0, 5]$.

Remember that a partial order is just a binary relation. In sets, it is customary to define or represent binary relations as sets of pairs, e.g., for the partial order $(\mathbb{N}, \leqslant)$, we can think of it as $'\leqslant' \equiv \{m, n \mid m \leqslant n\} \subset \mathbb{N}$.

Imagine you are now storing this in a relational database. The natural solution is to store such a relation in a table with two columns. That's what a mathematician would do. However, database people will tell you that a record should have a primary key—a unique "id"—as well as a constraint that a pair (x, y) for which $x \leqslant y$ must be unique.

Having such a table, we can add more details to this representation of a relation: the kind of relation, timestamps, etc. Now, it is no longer a binary relation. This turns our partial order into something different, which we will return to in Chapter 5, where non-Boolean kinds of logic are discussed.

Epi, Mono, Isomorphisms As usual, we can define a monomorphism, an epimorphism, and an isomorphism for functions between partial orders. Mono and epi are defined as they were defined before, on the elements of partial orders, and isomorphisms are those functions that have an inverse.

But surprise, surprise! Not every partial order function that is both epi and mono turns out to be an isomorphism. Here is a simple example:

Example 12 Take a partial order $X = (\{a, b\}, \{a \leqslant b\})$. It has two elements and one instance of a relation, $a \leqslant b$. Take another partial order, $Y = (\{a, b\}, \{\})$. It has the same two elements

that are not comparable. You can see that the inclusion $i : Y \rightarrowtail X$ is a mono and an epi, but it has no inverse. If $f : X \to Y$ were an inverse to i, it would have to map a to a and b to b, but in X we have $a \leqslant b$, hence in Y we should also have $a \leqslant b$. However, this is not the case.

Graphs

Every software user (at least a backend one) knows what a graph is, however, definitions vary. We will choose one for now, but be aware of the existence of others. Let's walk through three such definitions.

Undirected Graphs

Definition: Undirected Graph An *undirected graph* is a collection of *nodes* (instances of some type A), some of which are connected by links. To be more precise, each such connection (called an *edge*) is between two nodes. For every pair of nodes, there is at most one connection.

For each collection of nodes A, there are two extreme kinds of graphs on A, a *discrete graph*, where there are no edges at all, and a *complete graph*, where every two nodes are connected by an edge.

Undirected graphs are the most traditional kind of graphs. The term originates in 1878, but the whole idea is much older. While we are not studying such graphs in depth here, some interesting problems related to such graphs are listed.

Eulerian Path Leonard Euler was a great German mathematician born in Königsberg, Prussia, and he found the following problem interesting: given that there are seven bridges in Königsberg, can one take a walk crossing each bridge exactly once?

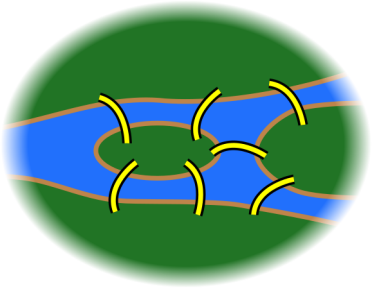


Figure 10: *Königsberg Bridges, schematic.*

Before working on that problem, consider a simpler one: draw an envelope without lifting your pen. The solution:

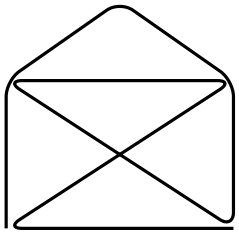


Figure 11: *An envelope drawn without lifting the pen.*

Why this solution? Notice, if a node has one edge, we have to either start or finish there. If it has two edges, we can just draw right through it. If we add two more edges, we can draw through the node again, each time using two more edges. Now, if there is just one node with an odd number of edges, we should either start or finish at that node. If there are two odd-edged nodes, we have to start and finish at these two nodes, as in the envelope picture.

This strategy works until we find ourselves in Königsberg: then we are out of luck, since every node (island) has an odd number of edges (bridges). Our path cannot have more than two ends, though. So, there is no solution, if we try to solve the problem on a flat map. On a map, or on a flat Earth, there is no way to bypass the Pregolya River. Of course, the Earth is not flat, and we can think about walking around the globe to get to the other side of the river.

Chinese Postman Problem A postman must deliver mail to all nodes on a graph, and for some reason, he or she must walk through every edge, eventually returning to his or her home node. Being smart, the postman wants to make the path as short as possible. If the graph were Eulerian, the problem would have a neat solution. If not (say, this is a Königsberg postman), something else should be done. A length is ascribed to each edge, and the problem turns into a "find the shortest path" problem.

The general solution to this problem is known to be equivalent to listing all possible paths in a graph.

Planar Graphs Let's start with a practical problem. Three cottages are standing nearby, and there are three utility companies (electricity, gas, and water). Is there a way to provide electricity, gas, and water to these cottages so that the lines don't cross each other? This picture represents the problem (this specific graph is called $K_{3,3}$):

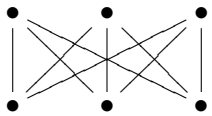


Figure 12: $K_{3,3}$.

It turns out that it is impossible to do so, unless we involve other dimensions (three dimensions are enough, and electrical wires can go in the air).

Another case of a graph that cannot be drawn on a flat surface without crossing edges is a complete graph with 5 nodes (called K_5):

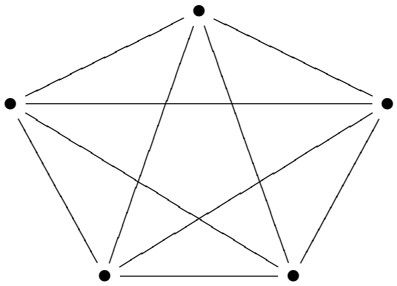


Figure 13: K_5.

For a generic graph, can we somehow decide whether it is *planar* (that is, can be drawn) or not? It turns out, these two graphs, $K_{3,3}$ and K_5, decide everything. If our graph contains any of these two, the problem cannot be solved, but if it does not contain them, the graph is planar.

Directed Graphs

Definition: Directed Graph A *directed graph* is a collection of *nodes* with a collection of *edges* that connect one node to another or to itself, so that there is not more than one edge for each pair of nodes. If a and b are nodes, an edge from a to b is denoted as $a \to b$.

Example 13 Take a partial order $(A, \leqslant)$, and introduce an edge from $a \to b$ for every pair $a \leqslant b$. Now we have a directed graph for the partial order $(A, \leqslant)$.

Example 14 A directory hierarchy in a computer is a directed graph. In a file system, the fact that one folder is a subfolder of another can be represented as an edge of a graph, from containing folder to subfolder. If links in our file system exist, we can have any graph structure. In any case, we don't generally have a tree.

Example 15 A directed graph can have cycles. Note that in this case, we do not have a partial order. If we have a cycle, containing two distinct nodes a and b, by transitivity, we will have $a < b$ and $b < a$. This breaks the requirement of antisymmetry.

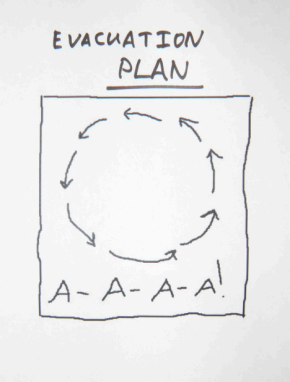


Figure 14: *Directed graph with cycles.*

Definition: Directed Acyclic Graph A *DAG*, or *directed acyclic graph*, is a directed graph that has no cycles. A cycle in a graph likely needs to be defined, but the reader can guess what it is.

The lack of cycles makes DAGs a good tool for organizing both data structures and processes in system design.

Example 16 The Java memory model has a well-defined "happened-before" relation. It is a partial order and a directed graph.

Directed Multigraphs

Definition: Directed Multigraph A *directed multigraph*, or just a *graph*, consists of a collection of nodes and a collection of edges, where each edge has a source node and a target node. Formally, we have:

$(Nodes, Edges, source : Edges \to Nodes, target : Edges \to Nodes)$.

Directed multigraphs are good for describing actions, transitions, and machines.

Example 17 Given a state machine, we can represent each state as a node of a graph and each transition as an edge. The edges are labeled by the symbols of the input alphabet. In UML, it is called a *state machine diagram*:

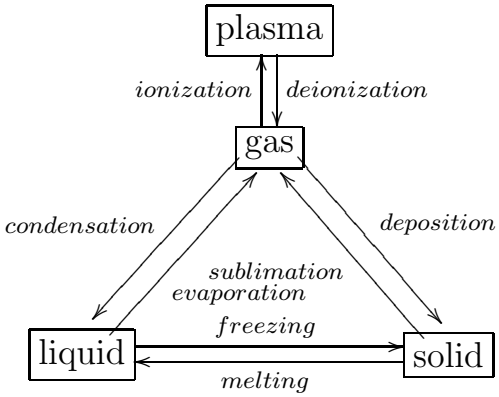


Figure 15: *Matter state diagram.*

A directed graph is a graph ("directed multigraph"), too, including no more than one edge from node to node. Hence, a partial order is a special case of graph.

Example 18 Remember that a collection of all functions $A \to A$ is a monoid. It can be represented as a graph by having just one node, A, and as many edges as there are functions. Since we can have more than one node in a graph, we can imagine, given a collection of types, e.g., $\{Int, String, Char, Bool, Double\}$, a graph with these types as nodes and all functions between these types as edges.

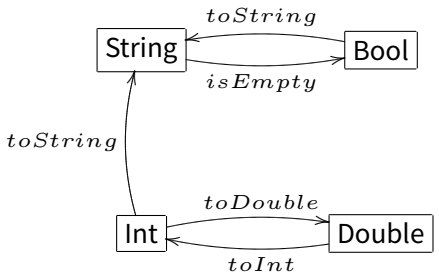


Figure 16: *Sample graph of Scala types.*

Chapter 4. Boolean Logic

"Logic" is an immense area of human knowledge. I have no intention to discuss all aspects of it but will focus instead on simple notions and in language that is reasonable enough to be used in daily practice by software developers. Lawyers, philosophers, physicists, or linguists will probably need to use very different kinds of logic than what we are going to study here. Boolean logic is the variant of logic where for each statement, either the statement or its negation is true. The logic used for most, but not all, hardware and software is assumed to be Boolean. In this chapter, we only cover Boolean logic, but in Chapter 5, we will discuss non-Boolean logic.

The Language of First-Order Logic

Before talking about logic, even vaguely, it's time to introduce a language used throughout this text. It is neither loose nor strict but somewhere in the middle.

Definition: First-Order Language

A first-order language consists of the following:

- *Names*, which are intended to denote objects in some kind of universe (e.g., "John" denotes a unique human being who exists in a small world, and 10 denotes a number, if we know what kind of numbers we are talking about).
- *Variables*, denoted by Latin letters of the alphabet, possibly with indices, are used for unknown objects within a certain domain.
- *Functions*, which are used to form terms from names, variables, and other terms and can use *infix notation*, such as $2 + 3$. A function takes arguments and has a result value.
- *Terms*, which are expressions produced using names, functions, and other terms.
- *N-ary relations*, which denote statements that some terms have certain properties (see examples below).
- *Formulas*, which are expressions built from n-ary relations applied to terms.

Functions are also known as *operations*. A one-parameter operation is called *unary*, a two-parameter operation is called *binary*, and a three-parameter operation is called *ternary*. Terms for bigger *arities* (up to "googleplexanary") can be found on Wikipedia.

We could express all this formally, e.g., in Backus Normal Form, but it's likely unnecessary.

In summary, terms are either built from names and functions, using parentheses, or are formulas that use terms. We'll leave it at that for now.

Example 1

- `sin(ln(2.718285)) < cos(exp(0.001))`. Here `2.718285` and `0.001` are *names*, `ln`, `sin`, `exp` are *functions*, `sin(ln(2.718285))` is a *term), `<` is a* binary relation*, and the whole expression is a "formula.*
- `phoneNumber(John) = "314 159 2654"`
- `Alaska`

Note that a "first order language" is not a unique universal language. It is a notion that applies to any language we can build from a collection of names, functions, and relations of our choice. There are many first order languages.

Example 2 The language of arithmetic, where integer numbers serve as values in the domain of integer numbers, arithmetic operations on integer numbers serve as functions, and comparisons and equality serve as relations.

Example 3 Planar geometry. First, we need numbers and symbols for points, lines, circles and angles. Then, we add functions and relations:

- $isLine(L_1, P_1, P_2)$: L_1 is a line that contains points P_1 and P_2.
- $circle(P, R)$: this term denotes a circle with its center at P and a radius R.
- $center(C)$: this term denotes a point that is the center of circle C.
- $liesOn(P_1, L_1)$: point P_1 lies on line L_1.
- $liesOn(P_1, C_1)$: point P_1 lies on circle C_1.
- $isBetween(P_1, P_2, P_3)$: points P_1, P_2, and P_3 are on the same line, and P_1 is between P_2 and P_3.
- $areParallel(L_1, L_2)$: lines L_1 and L_2 are parallel.

Example 4 A version of *naïve set theory* can also be expressed as a first-order language:

1. Names denote sets or elements (may not be sets).
2. $a \in b$, where a is an element, b must be a set.
3. $a = b$.
4. $\{a_1, ... a_n\}$ is a set, where $a_1 ... a_n$ are elements.
5. $s_1 \cap s_2$ is a set, where s_1 and s_2 are sets.
6. $s_1 \cup s_2$ is a set, where s_1 and s_2 are sets.

Example 5 Directed graphs. They include nodes and edges, and the only relation:

$isBetween(Node_1, Node_2, Edge_3)$.

Logical Operations

The logical formulas provided before (relations/predicates) are called *atomic formulas*. We can build more formulas by combining them.

Negation

For a formula P, its *negation* is denoted as $\neg P$ and is true if and only if P is false. It is clear from the definition that double negation of a formula P is true if and only if P is true.

Conjunction

For two formulas, P and Q, their *conjunction* is denoted as $P \wedge Q$ and is true if and only if both P and Q are true.

Disjunction

For two formulas, P and Q, their *disjunction* is denoted as $P \vee Q$ and is true if and only if at least one of P and Q is true.

Sentences: Combining Operations

Sentences are built from atomic formulas using negation, conjunction, and disjunction. To avoid ambiguity, parentheses may be required in the expressions.

Example 6 $((x < 7) \vee ((y < x) \wedge \neg isEmpty(\texttt{"}helloworld\texttt{"})))$

Properties of Operations

Describing these properties, we use the symbol $\equiv$, which roughly means that the expressions on the left and on the right are equivalent. The exact meaning of such equivalence binary relations may vary from theory to theory.

Associativity Both conjunction and disjunction are associative.

$(P \vee Q) \vee R \equiv P \vee (Q \vee R)$

$(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$

Commutativity Both conjunction and disjunction are commutative.

$P \vee Q \equiv Q \vee P$

$P \wedge Q \equiv Q \wedge P$

Idempotence Both conjunction and disjunction are idempotent.

$P \vee P \equiv P$

$P \wedge P \equiv P$

Double Negation $\neg\neg P \equiv P$

De Morgan Laws

- $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
- $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$

More Operations

We've discussed negation, conjunction, and disjunction. Can other operations be added?

Negation is a unary operation. Conjunction and disjunction are binary. How about other arities?

We can start with functions of zero arity, that is, with *constants*, and remember the two constants that are always lurking in the background: $True$ and $False$. There may be more nullary operations, but for the sake of this discussion, let us limit ourselves with just two, and both are necessary. The first one, $True$, is a neutral element for conjunction. The second one, $False$, is a neutral element for disjunction.

Now take a look at the unary operations. How many can we define on two logical constants?

Obviously, exactly four operations are possible: here is a table with all four operations:

x	**identity**	**always True**	**always False**	**negation**
$True$	$True$	$True$	$False$	$False$
$False$	$False$	$True$	$False$	$True$

For binary operations, there are more combinations (any idea how many?). It is not necessary to list them all here, so we will just mention the important ones:

		conjunction	**disjunction**	**implication**	**equivalence**	**Peirce arrow**	**Sheffer stroke**
x	y	$x \wedge y$	$x \vee y$	$x \to y$	$x \leftrightarrow y$	$x \downarrow y$	$x \uparrow y$
T	T	T	T	T	T	F	F
T	F	F	T	F	F	F	T
F	T	F	T	T	F	F	T
F	F	F	F	T	T	T	T

Some operations can be defined using other operations. For example, implication can be defined as $\neg x \vee y$, and equivalence can be defined as $(x \wedge y) \vee (\neg x \wedge \neg y)$.

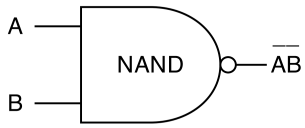
Conjunction and disjunction are traditionally used as base logical operations. Any formula can have a standard representation via negation, conjunction, and disjunction. Moreover, disjunction can be expressed via negation and conjunction: $x \vee y \equiv \neg(\neg x \wedge \neg y)$. Now we see that two operations, negation and conjunction, are sufficient to define all other operations.

Instead of negation and conjunction, we could choose negation and disjunction, and express conjunction as $x \wedge y \equiv \neg(\neg x \vee \neg y)$. Again, two operations are all we need.

Can we go further? Yes. Two operations, which can each be used to represent all other operations, are: the *Peirce Arrow*, denoted as $\uparrow$, or *NOR* (because it is equivalent to $\neg(x \vee y)$); and the *Sheffer Stroke*, denoted as $\downarrow$, or *NAND* (because it is equivalent to $\neg(x \wedge y)$).

Imagine you must use only one circuit to build a logical schema—which one would you use? You have two choices, either *NAND* or *NOR*.

Hardware professionals hold the opinion that *NAND* is the best way to save electricity.



A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

Figure 17: *NAND, electronic schema. Here,* 1 *denotes* $True$ *and* 0 *denotes* $False$.

How do we express negation, conjunction, and disjunction? Here's the solution for *NAND*:

$\neg A = A \uparrow A$

$A \vee B = \neg((\neg A) \uparrow (\neg B))$

A solution for *NOR* is very similar, and I'm leaving it to the reader as an exercise to find it.

Proving Something

Premises and Conclusions

With a first-order language, we can connect our formulas (see definition), calling some of them *premises* and some *conclusions*. Informally, you would list premises and then come up with a conclusion, or you could go in the opposite direction by presenting conclusions, because of premise 1, premise 2, etc.

Example 7 "All men are mortal. Superman is a man. *Hence*, Superman is mortal."

Some may agree, and some may disagree. Some will ask, "Which Superman?", thus invalidating the whole discussion.

Example 8 "Pavlova is a man: after all, Pavlova is mortal, and all humans are mortal." We may argue that, judging by the name, Pavlova must be a human. I will also add that Pavlova is a cat. The fact that she is dead now does not make her a man.

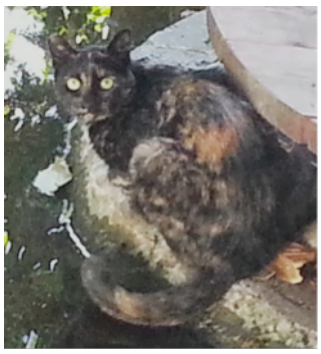


Figure 18: *Pavlova, a cat.*

Arguments

A sequence of premises followed by a conclusion is called an *argument*. Arguments are formally written in the following form, using the character $\vdash$ (called "turnstile"):

$Premise_1, Premise_2, \dots, Premise_n \vdash Conclusion$

Valid and Sound Arguments

As you can see from the examples above, some conclusions make sense, some don't. In addition, some premises make sense, and some don't. We should disambiguate these situations.

Definition An argument is *valid* if, assuming that the premises are true, the conclusion is true. An argument is *sound* if it is valid, and all the premises are true.

Note that we have just introduced the word "true" into our discourse. This is a little bit unusual, but we do not need to be as formal here.

You can check for yourself to see whether each of the examples above are valid, sound, both, or neither.

Premises are *inconsistent* if they contradict each other, that is, if you can deduce $\perp$ (*false*) from them. Remember, in this case, the argument is not sound (incorrect premises!), but it is always valid. This may be the easiest way to prove literally anything: just start with inconsistent premises.

Proofs, Formally

Definition A *proof* is a step-by-step demonstration that a conclusion follows from the premises (that is, that the argument is valid).

How do we know that our proof makes sense? There are a variety of approaches. One of these is by applying *rules*. We will walk through such rules, without questioning them (questioning them would involve proving theorems, which is beyond the scope of this book).

Below are the proof rules. Some of them are obvious, some less obvious. The rules either have a form of argument (see above) or consist of several arguments.

Negation Elimination

$$\neg\neg P \vdash P$$

Informally, it means that if we have "not not P," we can say that P is true. This may sound obvious but consider this: We could not prove P. We could only prove that assuming "not P" leads to a contradiction. Does this give us P? Probably not, generally speaking. For example, Scott Peterson could not prove he did not commit the crime and hence was found guilty. That's how classical logic works.

Conjunction Elimination

$$P \ \wedge \ Q \ \vdash \ Q$$

Informally, if we have a conjunction of P and Q, then we have Q. This is part of the definition of conjunction.

Conjunction Introduction

$$P, Q \vdash P \wedge Q$$

Informally, if we have P and Q, then we have a conjunction, $P \wedge Q$. This is part of the definition of conjunction.

Disjunction Introduction

$$P \vdash P \vee Q$$

Informally, if we have P, a disjunction of P and Q also holds. This is part of the definition of disjunction.

Disjunction Elimination

$$\frac{P \vee Q, P \vdash R, Q \vdash R}{R}$$

The meaning of the schema above is the following: we have $P \vee Q$, and we have *subproofs* that P yields R and Q yields R. Then we have R.

Negation Introduction

$$\frac{P \vdash \bot}{\neg P}$$

This rule can be considered a definition of negation.

$\bot$ Introduction

$$P, \neg P \vdash \bot$$

This property of negation consists of deducing "bottom" from both P and its negation.

$\bot$ Elimination

$$\bot \vdash P$$

This is the property of "bottom": anything follows from it. This property is suitable for proving the existence of supernatural creatures and entities, as well as their non-existence.

Proof by Contradiction This rule consists of the following: to prove $\neg S$, assume S, and deduce $\bot$ (that is, *false*).

Example 9 Let's prove that $\sqrt{2}$ is irrational, using the rule above, *Proof by Contradiction.*

First assume it is rational, $\sqrt{2} = p/q$, where p and q are natural numbers, and mutually prime (no common divisors). You may ask, "Why mutually prime?" Because common divisors don't count, $(p * x)/(q * x) = p/q$.

If we have $\sqrt{2} = p/q$, then $p^2 = 2 * q^2$.

Can p be odd? No! It is $2 * k$, for some k.

If p were odd, its square would be odd too, but we just found that the square is $2 * (something)$.

When p is even, $p^2 = p_1^2 * 4$, right? So $q^2 = 2 * p_1^2$. Now, it turns out that q is also even, so p and q do have a common divisor (=2)—oops! Contradiction! Our assumption was wrong.

Proof by Cases This is an extension of *Disjunction Elimination*, covering cases where the number of alternatives is not necessarily two.

To prove that $P \vee Q \vdash R$, it is enough to prove that $P \vdash R$ and $Q \vdash R$.

This rule can be extended to a list of options $P_1, P_2, ..., P_n$ and then to proving that $P_1 \vee P_2 \vee, ... \vee P_n \vdash R$

Example 10 Prove that a rational number a can be represented as b^c where both b and c are irrational numbers.

Proof We know that $\sqrt{2}$ is irrational. Now take $d = \sqrt{2}^{\sqrt{2}}$. If d is rational, we have found our example. For a case where d is not rational, we are free to take $d^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}*\sqrt{2}} = \sqrt{2}^2 = 2$. It is a rational number produced from two irrationals.

In this example, we had no information whether d is rational or not. Rational or not, we prove our statement.

Identity Elimination Rule Also known as *Indiscernibility of Identicals*, *Substitution Principle*, and *Identity Elimination*. The rule is:

$$P(a), a = b \vdash P(b)$$

For example,

$x^2 > x^2 - 1,\ x^2 - 1 = (x + 1) * (x - 1) \vdash x^2 > (x + 1) * (x - 1)$

Identity Introduction Rule Also known as *Reflexivity of Identity*. The rule is:

$$P \vdash x = x$$

Here x is a variable. The rule means anything can be substituted for x.

From these two rules, we can deduce that identity is symmetric and transitive:
$a = b,\ a = a \vdash b = a$
$a = b,\ b = c \vdash a = c$

Neat, right? The properties seem to come from nothing, but they arise through substitution, which turns out to be a rather powerful rule.

Expressions Simplification

The meaning of a long expression involving our three operations—$\wedge, \vee, and \neg$—may be hard to understand. For example, $(A \vee B) \wedge C \wedge (\neg(\neg B \wedge \neg A) \vee B)$ may confuse most people. Fortunately, the rules above help us to refactor such expressions into a standard form, which is usually simpler.

Negative Normal Form

The first such refactoring consists of pushing negation as deeply as possible, right in front of atomic formulas (that is, variable names). The result of such normalization is called *negative normal form*, also known as *NNF*.

When we encounter two negations in an expression, we eliminate these two negations, by the double negation rule. Every time a negation is encountered before a disjunction or a conjunction, we can apply a De Morgan Law. This way, for example, we can perform the following transformation:

$$\neg\neg\neg(\neg A \vee \neg(B \wedge C) \vee D) \equiv$$

$$\neg(\neg A \vee \neg(B \wedge C) \vee D) \equiv$$

$$A \wedge B \wedge C \wedge \neg D$$

Disjunctive Normal Form

After applying negative normalization, we are left with an expression consisting of a mix of conjunctions and disjunctions. Knowing distribution laws, we see that a conjunction of two disjunctions can be represented as a disjunction:

$$(A \vee B) \wedge (C \vee D) \equiv (A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (B \wedge D)$$

In this way, any complex expression can be refactored to a disjunction.

An expression is said to be in *disjunctive normal form* (*DNF*) if it is a disjunction of conjunctions of atomic formulas (or their negations).

Here's an example:

$$(A \to (B \wedge C)) \ \wedge \ (\neg B \to (A \wedge C)) \equiv$$

(replacing implication)

$$(\neg A \vee (B \wedge C)) \ \wedge \ (\neg\neg B \vee (A \wedge C)) \equiv$$

(eliminating double negation)

$$(\neg A \vee (B \wedge C)) \ \wedge \ (B \vee (A \wedge C)) \equiv$$

(distributing over disjunction)

$$(\neg A \wedge (B \vee (A \wedge C))) \ \vee \ (B \wedge C \wedge (B \vee (A \wedge C))) \equiv$$

(distributing over disjunction)

$$(\neg A \wedge B) \vee (\neg A \wedge (A \wedge C)) \vee (B \wedge C \wedge B) \vee (B \wedge C \wedge (A \wedge C)) \equiv$$

(conjunction with negation is false)

$$(\neg A \wedge B) \vee (False \wedge C) \vee (B \wedge C) \vee (A \wedge B \wedge C) \equiv$$

(simplifying)

$$(\neg A \wedge B) \vee False \vee (B \wedge C) \equiv$$

(false can be eliminated)

$$(\neg A \wedge B) \vee (B \wedge C)$$

Conclusion

This was the first of three chapters dedicated to logic. Next, we will learn how to handle logic that does not have Boolean properties. And then, quantifiers will be discussed.

Chapter 5. Non-Boolean Logic

The Meaning of Non-Booleanness

Traditionally, as soon as we switch from the complicated logic of the real world into formal discussions, we assume that just two outcomes are possible for any statement: either it is true or not true. This is not a feature of mathematical discourse but rather an interesting feature of the modern mentality. In real life, we admit that there are more possibilities—something we may not know today, we may learn tomorrow. We may assume something to be true with a certain probability. Similarly, in science we may not know the answer, and the answer may be something we don't expect.

For example, for decades, mathematicians were trying to determine whether there exists an intermediate set size between countable and continuum. It turned out that if we assume there is such a size, then it exists; if we assume there is none, then it does not exist. It is not a theorem, it is an axiom. An axiom is not a fact that is always true. An axiom is a constraint within a theory. If we could prove that an axiom is true, we would not have needed it.

In intuitionistic logic, the double negation law does not have to apply. If it does, the logic is Boolean. In this chapter, we will gradually find a way to remove the double negation law while keeping the remaining rules and axioms that don't depend on double negation. Of course, if the double negation law is included, we still have an intuitionistic logic that is also Boolean.

A logic may have more than two logical constants, but that does not make it non-Boolean. The double negation rule may still be applicable in such a logic.

Example 1. Boolean Logic, but not 2-Valued Consider bytes and their operations: bitwise conjunction, bitwise disjunction, and negation. There are 256 different values, but we know that double negation in this bitwise logic is an identity. So, this 256-valued logic is Boolean.

An example of a non-Boolean logic will be provided a little bit later.

Dropping Booleanness

Remove the rule stating that either P or $\neg P$ is true. Formally, $\vdash P \vee \neg P$. Once we remove this rule, we cannot apply it anymore in general settings, although it may still work for some values.

We immediately bump into a problem. We used this rule to define implication via negation: $P \rightarrow Q \equiv \neg P \vee Q$, and this will not work. We must define it differently. Here is a rather reasonable solution:

$$(P \land Q) \vdash R \equiv P \vdash (Q \to R)$$

Informally, saying that R can be deduced from a conjunction $P \land Q$ is the same as saying that we can deduce the implication $Q \to R$ from P. Imagine Q is $\top$. Then, from the formula above, it follows that $Q \to R$ is the same as R. On the other hand, if Q is closer to the bottom, the situation may change. For example, if Q is "below" R, the implication $Q \to R$ is true. We can interpret the implication defined this way as the level of dependency of R on Q.

Consequently, we are able to define implication via conjunction, and negation is not required.

Does this remind you of currying? Given a function $f(P, Q) : R$, we transform it to a function $f_c(P) : Q \to R$.

Now that implication is defined, we can define negation as $\neg P = P \to \bot$. Its properties make it similar to classical negation, but the double negation law is not generally applicable anymore. Assume we use the definition above. In the table below, we list some properties that are valid for this definition:

Statement	**Meaning**
$P \land \neg P \vdash \bot$	Negation of P is incompatible with P.
$\neg\neg\neg P \vdash \neg P$	Triple negation is the same as single negation.
$P \vdash \neg\neg P$	Intuitively, we know that double negation of P may be weaker than P, but if we have P, we have $\neg\neg P$.
$\neg P \lor \neg Q \vdash \neg(P \land Q)$	If not P or not Q, then we can't have both P and Q
$\neg(P \lor Q) \vdash \neg P \land \neg Q$	If disjunction of P and Q is not true, then neither P nor Q is true.
$\neg P \land \neg Q \vdash \neg(P \lor Q)$	If neither P nor Q is true, their disjunction is not true.

Some properties do not apply in this kind of logic:

Statement	**What's wrong with it?**
$\neg\neg P \vdash P$	Negation of negation of P is not strong enough to give us P.
$\neg(P \land Q) \vdash \neg P \lor \neg Q$	Even if we can't have both at the same time, this does not mean that one of them is always wrong.

Example 2. Three Logical Values ("Ternary Logic") Start with three logical values, $\top$, $\bot$, and ?. Define operations for them:

x	$\neg x$	$\neg\neg x$
$\top$	$\bot$	$\top$
?	$\bot$	$\top$
$\bot$	$\top$	$\bot$

x	y	$x \wedge y$	$x \vee y$
$\top$	$\top$	$\top$	$\top$
$\top$	?	?	$\top$
$\top$	$\bot$	$\bot$	$\top$
?	$\top$	?	$\top$
?	?	?	?
?	$\bot$	$\bot$	?
$\bot$	$\top$	$\bot$	$\top$
$\bot$	?	$\bot$	?
$\bot$	$\bot$	$\bot$	$\bot$

It may take some time to check whether disjunction and conjunction are associative, or that distribution laws are applicable.

This is the simplest intuitionistic logic, and it is a good tool for testing our statements.

The value "?," or "unknown," is somewhere between "truth" $\top$ and "false" $\bot$.

Later, we will take a closer look at the idea of "between"—now, let's consider another example.

Logic Produces a Partial Order

When we take all available logical values, we can introduce a partial order on them, by the following rule: $(p \leqslant q) \equiv ((p \wedge q) = p)$. In this case, conjunction plays the role of $glb(p, q)$. Is this a partial order? For a partial order, we need antisymmetry and transitivity.

Antisymmetry is easy: we need $p \leqslant q, q \leqslant p \vdash p = q$. Why? By definition of our order, we have $p \wedge q = p$ and $p \wedge q = q$, hence $p = q$.

Transitivity, $p \leqslant q,\ q \leqslant r \vdash p \leqslant r$, amounts to having $(p \wedge q) = p,\ (q \wedge r) = q \vdash (p \wedge r) = p$.

Suppose we have $(p \wedge q) = p$ and $(q \wedge r) = q$. Then, $p \wedge r = (p \wedge q) \wedge r = p \wedge (q \wedge r) = p \wedge q = p$, and we have transitivity.

In the picture below, one can see the partial order for the ternary logic discussed above:

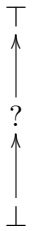


Figure 19: *Partial order of 3-valued logic.*

Example 3. Infinite Number of Logical Values Take $[0, 1]$, the set of all real numbers between 0 and 1, including 0 and 1. We can turn it into logic by defining:

- $\bot = 0$
- $\top = 1$
- $a \wedge b = glb(a, b)$
- $a \vee b = lub(a, b)$

In this example, implication is defined, as always, via the equivalence: $(x \wedge y) \leqslant z \equiv x \leqslant (y \to z)$.

We consider the following two cases separately:

First, when $y \leqslant z$, $(x \wedge y) \leqslant z$ is always true, for any x. Hence $y \to z$ cannot be smaller than any x, that is, it must be the top element, 1.

In the opposite case, when y is bigger than z (we are talking about numbers), $(x \wedge y \leqslant z) \equiv (x \leqslant z)$, so now we will have $x \leqslant z \equiv x \leqslant (y \to z)$, which means that $y \to z = z$.

Remember that we defined negation, $\neg x$, as $x \to 0$, and we have $\neg x \equiv if\ (x = 0)\ 1\ else\ 0$. You can also check whether double negation maps 0 to 0 and any non-zero value to 1.

In this example, we started with a set of values that was popular in "fuzzy logic," and we were able to build a pretty sound logic, except that it could not be made Boolean: double negation is not an identity.

Finding Good Partial Orders

We may try to build a logic from every partial order we find. Of course, not every partial order is good to build a logic from. In our examples, we had linear partial orders, where *glb* and *lub* were always defined, and were used for defining conjunction and disjunction. In general, though, *glb* and *lub* may not even exist, as shown in this picture:

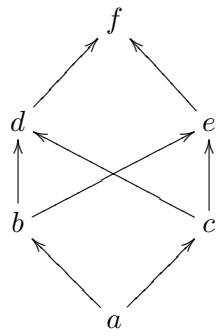


Figure 20: *Partial order with no glb/lub.*

There are no unique closest elements below both d and e: neither a nor b nor c satisfy the obvious requirements for *glb* to be unique and the closest possible. We need *glb* and *lub*, since they are our conjunction and disjunction operations.

Definition: Lattice

A partial order where *glb* and *lub* are defined for every two elements is called a *lattice*.

So, we need a lattice if we want conjunction and disjunction to be defined.

Yet, we also need $\top$ and $\bot$, neutral elements for conjunction and disjunction, that is, from a partial order point of view, *top* and *bottom* elements.

Definition: Bounded Lattice

A lattice having top and bottom elements is called a *bounded lattice*.

We are almost there in our path to defining an algebraic structure suitable for logic. What remains is the operation of implication, $a \to b$, that has to be defined for all a, b. Once this operation is defined, with the right properties, we have what is called a *Heyting algebra*.

Definition: Heyting Algebra

A bounded lattice that has an operation $a \to b$ with the following property:

$$a \leqslant (b \to c) \equiv (a \wedge b) \leqslant c$$

is called a *Heyting algebra*.

This is the last property required for defining a intuitionistic logic. Remember that a Boolean logic is a special case of intuitionistic logic.

Example 4. Define Implication for the Ternary Logic The ternary intuitionistic logic from Example 2 must have the operation of implication, satisfying the definition. That is, $a \leqslant (b \to c) \equiv (a \wedge b) \leqslant c$.

For the case when $b = \bot$, the definition is simple: only
$a \leqslant (\bot \to c) \equiv (a \wedge \bot) \leqslant c$ is required, but since $a \wedge \bot = \bot$, this is the same as having $\bot \leqslant c$, which is always true. So, $a \leqslant (\bot \to c)$ needs to be true for any possible a. The only candidate for this property is $\top$, meaning that $\bot \to c = \top$.

A similar argument shows that $\top \to c = c$. For $? \to c$, there are more complications. Let's skip the argument (perform it as an exercise) and add a column to the table of operations instead:

x	y	$x \wedge y$	$x \vee y$	$x \to y$
$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$?$	$?$	$\top$	$?$
$\top$	$\bot$	$\bot$	$\top$	$\bot$
$?$	$\top$	$?$	$\top$	$\top$
$?$	$?$	$?$	$?$	$\top$
$?$	$\bot$	$\bot$	$?$	$\bot$
$\bot$	$\top$	$\bot$	$\top$	$\top$
$\bot$	$?$	$\bot$	$?$	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$

Example 5. Add a Value to Ternary Logic What happens if we take the ternary logic from the example above and add one more intermediate value, as shown in this picture:

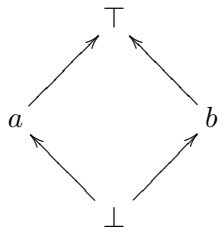


Figure 21: *Sample Heyting algebra that is actually Boolean.*

What will the logic look like now? Conjunction and disjunction are obvious, since they are defined by *lub* and *glb*, so that $a \wedge b$, for instance, is $\bot$, and $a \vee b$ is $\top$. Implication looks a bit more complicated.

For our structure to be a Heyting algebra, we have to use its definition of implication. By definition, for any x the statement $x \leqslant (a \to b)$ is equivalent to the statement $x \wedge a \leqslant b$. There are only two values with the property $y \leqslant b$: b and $\bot$. So, the statement that $x \leqslant (a \to b)$ is equivalent to having the conjunction $x \wedge a$ to be equal to either b or $\bot$.

There is no such x for which such a conjunction is equal to b. Due to the properties of conjunction, $x \wedge a = x \wedge a \wedge a$, and if $x \wedge a$ were equal to b, then $b \wedge a$ would be equal to b, but we know that $b \wedge a$ is $\bot$.

The only other choice is $\bot$. So $x \leqslant (a \to b)$ is equivalent to $x \wedge a = \bot$. Two such elements x satisfy $x \wedge a = \bot$: $\bot$ and b. The implication $(a \to b)$ cannot be equal to $\bot$, since in this case $x \leqslant (a \to b)$ will not be satisfied. The only remaining candidate for $a \to b$ is b.

And the solution is: $(a \to b) = b$.

Similarly, by definition, for any x, stating that $x \leqslant (a \to \bot)$ is equivalent to stating that $x \wedge a \leqslant \bot$. Just one value y has the property $y \leqslant \bot$: this value is $y = \bot$. So, $x \leqslant (a \to \bot)$ is equivalent to $x \wedge a = \bot$. Since $a \to \bot$ should dominate all such x, we can try to find the largest, the *lub* of these x, which is b. b satisfies the equation, $b \wedge a = \bot$, so the answer is: $a \to \bot = b$.

Have you noticed the following? $b = \neg a$; and similarly, $a = \neg b$. The logic turned out to be Boolean. Of course, its size is 4, which is a power of 2.

The latter fact, that the size of a finite Boolean algebra needs to be a power of 2, follows from such an algebra having to be atomic. However, this goes beyond the scope of this book.

Example 6 Remember, the rule from Boolean logic $\neg(P \wedge Q) \vdash \neg P \vee \neg Q$ does not generally work for an arbitrary intuitionistic logic. Can we demonstrate this fact for a partial order?

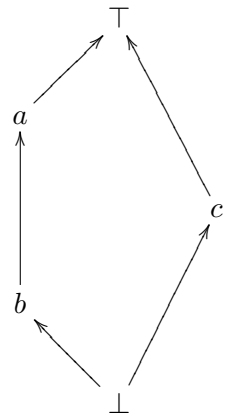


Figure 22: *Sample Heyting algebra that is actually Boolean.*

In this lattice, we see that $a \wedge c = b \wedge c = \bot$ and the negation of $\bot$ is $\top$. We have $\neg(a \wedge b) = \top$. Now $\neg a = c$, and $\neg b = c$, too. So $\neg a \vee \neg b = c$. Since $\top \vdash c$ is not true, we have our example or, rather, a counterexample.

Minimal Set of Operations? Unlike Boolean logic, which can be expressed through just a Peirce arrow or Sheffer stroke, there is no way to express all operations via just one in a generic, intuitionistic logic.

Proof in Intuitionistic Logic

Informally, a proof in intuitionistic logic consists of providing "evidence." Almost all the rules that were described in the previous part are applicable, with one exception: the *Negation Elimination* rule, $\neg\neg P \vee P$. By excluding this rule, we have a proof system that is good in intuitionism.

Chapter 6. Quantifiers

What Are They?

A predicate—something you are sure to be familiar with—is a judgment about an element of a collection. Its value may be true or false (or something else) depending on the element. Quantifiers are used to express judgments about the whole collection of elements.

Example 1 Consider the statement "all men are mortal." This statement says nothing about any particular man and we say nothing about any woman or any other possible kind of human. However, it applies to any man, due to the use of the word "all."

From this statement one could deduce that Justin Bieber, whoever this person may be, is mortal.

Example 2 "Every ten minutes a man is mugged in the streets of New York." This statement is, again, talking about men in general. We cannot say for sure from this statement whether any particular man is mugged in the streets of New York, but it definitely does not apply to those who are not in New York. What's more important is that while this statement may be generally correct, we may never manage to find such a man for an interview.

These two examples introduce the two quantifiers we are going to discuss next, existential and universal.

Universal Quantifier

The *universal quantifier*, denoted as $\forall$ (and called *for all*) is used for judgments that state something about all elements of a collection. An extreme case would be a statement about the whole world (e.g., "nobody's perfect" or "every statement is a lie").

We write this kind of statement as $\forall x\ P(x)$, and it is pronounced "for all x, P(x)."

When observing things from a computing point of view, we are either in a world with types or in a world without types. In a typeless world, the scope of a quantifier can only be delimited by specifying a collection. In a world with types, the scope can be delimited by either specifying a collection or by specifying a type for each variable used in the quantifier.

In Scala/JavaScript/Java

In Scala, given a `c:Collection[A]` (actually, given any `c:TraversableLike[A]`) of values of type `A`, and a predicate `p: A=>Boolean` (that is, a function from `A` to `Boolean`), we can

define `val itsPerfect = c forall p`, which is true if and only if `p(x)` is true for all `x` in `c`.

The `Array` object in JavaScript defines a function called `every`. For example, we can write `[1,2,3].every(n => n > 0)`, and the result will be Boolean `true`.

Java also has methods that implement quantifiers, for instance:

```
List<Integer> list = Arrays.asList(3, 7, 71, 89, 149);
boolean answer = list.stream().allMatch(n -> isPrime(n));
```

Existential Quantifier

Similar to the universal quantifier, the *existential quantifier*, denoted as $\exists$ (pronounced *there exists*) is used for judgments that state the existence of an element in a collection that satisfies a given predicate. Again, an extreme case would be when the range of the statement is unreasonably wide (e.g., "somewhere in our universe a perfect human exists" (our cat knows such a human: herself), or "a set exists that contains as elements all those sets that don't contain themselves as elements").

We write this as $\exists x\ P(x)$ and pronounce it "there exists an x, such that $P(x)$."

From a computing point of view, we may have the option to limit the range by either specifying a collection or a type.

In Scala/JavaScript/Java

In Scala, given a collection `c:Collection[A]` (a `c:TraversableLike[A}` would suffice) of values of type `A`, and a predicate `p: A=>Boolean` (that is, a function from `A` to `Boolean`), we can define

```
val haveSome = c exists p
```

which is true if and only if `p(x)` is true for some `x` in `c`.

The `Array` object in JavaScript defines a function called `some`. For example, we can write `[1,2,3].some(n => n > 0)`, and the result will be Boolean `true`.

Java also has methods that implement quantifiers, for instance:

```
List<Integer> list = Arrays.asList(3, 7, 71, 89, 149, 42);
boolean answer = list.stream().anyMatch(n -> isPrime(n));
```

Quantifiers and Logical Connectives

The predicate inside a quantifier may be a conjunction, a negation, a disjunction, or an implication. On the other hand, we can build a statement using such connectives, with quantifiers inside. So, are they related in any way? E.g., $\exists x\ (P(x) \wedge Q)$: how is it related to $(\exists x P(x)) \wedge Q$?

To figure out each combination's behavior, we can delimit the range of the quantifier. Instead of looking through a continuum of values and trying to figure out whether we can ever prove anything, we can use just two values in the range. If a statement is not true for a choice of two, it won't be true for a bigger collection, in general settings.

Also, to be on the safe side, check whether our statement is still true on an empty collection—further on, we will see why.

Below, we will investigate the behavior of logical connectives and quantifiers, and then look for differences in behavior for intuitionistic and Boolean logic.

Conjunction and Existential Quantifier

Can we check to see how $\exists x\ (P(x) \wedge Q(x))$ and $(\exists x\ P(x)) \wedge (\exists x\ Q(x))$ are related?

Suppose we only have x_0 and x_1.

The first statement says: $(P(x_0) \wedge Q(x_0)) \vee (P(x_1) \wedge Q(x_1))$

The second statement says: $(P(x_0) \vee P(x_1)) \wedge (Q(x_0) \vee Q(x_1))$

If we convert the second statement to disjunctive normal form (end of Chapter 4), we will now have:

$$(P(x_0) \wedge Q(x_0)) \vee (P(x_0) \wedge Q(x_1)) \vee (P(x_1) \wedge Q(x_0)) \vee (P(x_1) \wedge Q(x_1))$$

The second statement follows from the first, but the first does not follow from the second. So, now we have a good reason to suspect that:

$$\exists x\ (P(x) \wedge Q(x)) \vdash (\exists x\ P(x)) \wedge (\exists x\ Q(x))$$

But wait—what if our domain is empty? Then, obviously, both sides are false: there is no x to satisfy P or Q.

If the first one is true, we found an x such that both $P(x)$ and $Q(x)$ are true. Abandoning the information about Q, we produce $\exists x\ P(x)$. Similarly, we have $\exists x\ Q(x)$.

Conjunction and Universal Quantifier

The relation between the two will be an exercise for the reader to enjoy.

Disjunction and Universal Quantifier

Compare two statements, $\forall x\ (P(x) \vee Q(x))$ and $(\forall x\ P(x)) \vee (\forall x\ Q(x))$.

For a domain of just two elements, x_0 and x_1, we can rewrite the statements as $(P(x_0) \vee Q(x_0)) \ \wedge \ (P(x_1) \vee Q(x_1))$ and $(P(x_0) \wedge P(x_1)) \vee (Q(x_0) \wedge Q(x_1))$.

Converting the first one into disjunctive normal form, we get (reordered):

$$(P(x_0) \wedge P(x_1)) \vee (Q(x_0) \wedge Q(x_1)) \vee (P(x_0) \wedge Q(x_1)) \vee (P(x_1) \wedge Q(x_0))$$

As you can see, the second expression is a part of the first, so the first one follows from the second one.

$$(\forall x\ P(x)) \ \vee \ (\forall x Q(x)) \vdash \forall x\ (P(x) \vee Q(x))$$

We could argue differently, considering two cases, either $(\forall x\ P(x))$ or $(\forall x\ Q(x))$, and by deducing $\forall x\ (P(x) \vee Q(x))$ from each one, come to the same conclusion.

Disjunction and Existential Quantifier

If we look at two statements, $\exists x\ (P(x) \vee Q(x))$ and $(\exists x\ P(x)) \vee (\exists x\ Q(x))$, we can guess that they are the same.

Check it with a two-element domain, x_0 and x_1. Then you will have

$(P(x_0) \vee Q(x_0)) \ \vee \ (P(x_1) \vee Q(x_1))$ and $(P(x_0) \vee P(x_1)) \vee (Q(x_0) \vee Q(x_1))$

Thanks to the commutativity and associativity of disjunction, they are equal.

If we add more elements to the domain, nothing will change. It's still a disjunction of disjunctions, giving us the equivalence of the two:

$$\exists x\ (P(x) \vee Q(x)) \equiv (\exists x\ P(x)) \vee (\exists x\ Q(x))$$

Connectives and Quantifiers in Boolean Logic

All the discussion above applies to any intuitionistic logic, including Boolean. In this section, we will cover the aspects that are Boolean-specific.

Negation and Universal Quantifier

What is the meaning of $\forall x\ (\neg P(x))$? Is it related to $\neg\forall x\ (P(x))$ or to $\neg\exists x\ (P(x))$?

In the case of the domain $\{x_0, x_1\}$, the statement $\forall x\ (\neg P(x))$ is equivalent to $\neg\ P(x_0) \wedge \neg\ P(x_1)$, which is, in Boolean logic, the same as $\neg\ (P(x_0)\ \vee\ P(x_1))$.

We can generalize it to an arbitrary number of values, thus having $\forall x\ (\neg P(x))\ \equiv\ \neg\ (\exists\ x\ P(x))$. In plain English, "The statement 'for all x, $\neg P(x)$' is true if and only if there is no such x that $P(x)$."

Negation and Existential Quantifier

Now what about $\exists x(\neg P(x))$? Is it related to $\neg\forall x\ (P(x))$ or to $\neg\exists x\ (P(x))$?

Again, consider the case of a domain with just two elements, x_0 and x_1. The statement $\exists x(\neg P(x))$ is equivalent to $\neg P(x_0) \vee \neg P(x_1)$, which is, in our Boolean logic, the same as $\neg(P(x_0) \wedge P(x_1))$, that is, $\neg(\forall x P(x))$.

This is interesting. If we say that an x exists that does not satisfy P, then we know that not every x satisfies P. But what if we go in the opposite direction, and try to deduce the existence of such an x from the general fact that not every x satisfies P?

Here is an example. Suppose we decide to enumerate all possible sets of integers. Of course, some of them can be easily enumerated but definitely not all of them. So, a set of integers that cannot be enumerated must exist. Which set is it? (Suppose we found one: let's call it XXX. Then, we can introduce another enumeration, starting with this set XXX. Now we have it enumerated, thus contradicting the assumption that it cannot be enumerated.)

The problem is that Booleanness and the implicitly used axiom of choice promise us that any question has an answer, without saying whether this answer can be found.

A non-Boolean intuitionistic logic usually does not make any such assumptions.

Connectives and Quantifiers in Intuitionistic Logic

Remember, in intuitionistic logic negation is defined via implication, and implication is defined via conjunction and some equivalences. As a result, we cannot use in intuitionistic logic the

properties of quantifiers that we observed in Boolean logic.

When we switch from Boolean to a general case of intuitionistic logic, we won't rely on negation anymore. Implication is more important since $\neg P$ is defined as $P \to \bot$, while in Boolean logic we had $P \to Q$ defined as $\neg P \vee Q$.

In this section, we address the aspects that make intuitionistic logic different. We will have to look more closely into how quantifiers interact with implication and, cosequently, negation. Note that implication is a binary operation, $P \to Q$, where the two parameters, P and Q, behave differently.

Implication and Universal Quantifier

We can combine implication and the universal quantifier in the following three cases and then try to figure out if we can pull the implication outside:

1. $\forall x\ (P \to Q(x))$ vs $? \to ?$
2. $\forall x\ (P(x) \to Q)$ vs $? \to ?$
3. $\forall x\ (P(x) \to Q(x))$ vs $? \to ?$

Assume again that there are just two values in the domain, x_0 and x_1. Then our three cases turn into the following:

1. $(P \to Q(x_0)) \wedge (P \to Q(x_1))$
2. $(P(x_0) \to Q) \wedge (P(x_1) \to Q)$
3. $(P(x_0) \to Q(x_0)) \wedge (P(x_1) \to Q(x_1))$

We will cover them one by one.

Case 1 $\forall x\,(P \to Q(x))$**.** This case is rather simple. It is equivalent to $(P \to Q(x_0)) \wedge (P \to Q(x_1))$, which is the same as $P \to (Q(x_0) \wedge Q(x_1))$. Generalizing, $\forall x\ (P \to Q(x))$ is equivalent to $P \to (\forall x\ Q(x))$.

In plain words, if for each x the statement $Q(x)$ can be deduced from P, then we can deduce from P that $Q(x)$ is true for each x.

Case 2 $\forall x\,(P(x) \to Q)$**.** Let's try to see if $\forall x\ (P(x) \to Q)$ is equivalent to $(\exists x\ (P(x))) \to Q$.

This case is not as trivial as case 1. Assume again that we have just two values in the universe. Then the statement $\forall x\ (P(x) \to Q)$ can be rewritten as $(P(x_0) \to Q) \wedge (P(x_1) \to Q)$. We may try to check whether it is equivalent to $(P(x_0) \vee P(x_1)) \to Q$.

$(P(x_0) \vee P(x_1))$ is the l.u.b. of $P(x_0)$ and $P(x_1)$, so, if Q follows both from $P(x_0)$ and from $P(x_1)$, then it follows from their l.u.b., $P(x_0) \vee P(x_1)$.

Proving the equivalence in the opposite direction is trickier. Assume we have $(P(x_0) \vee P(x_1)) \rightarrow Q$, and assume $P(x_0)$ is true. Then $P(x_0) \vee P(x_1)$ is true, and so Q is true. We deduce Q from $P(x_1)$ the same way; so $(P(x_0) \rightarrow Q) \wedge (P(x_1) \rightarrow Q)$ is true.

While the equivalence "obviously" works in a populated universe, the situation is different when there are no values (our universe is empty). In an empty universe, there are no values x. The statement $\forall x\ (P(x) \rightarrow Q)$ in this case is true, whatever P and Q might be. What about $(\exists x\ P(x)) \rightarrow Q$? This is always true: since there is no x, $\exists x\ P(x)$ is false, and any Q follows from it. So, the equivalence works in an empty universe.

As a result, we have a good (but informal) hint that $\forall x\ (P(x) \rightarrow Q)$ is equivalent to $(\exists\ x\ (P(x)) \rightarrow Q$.

Case 3 $\forall x\ (P(x) \rightarrow Q(x))$**.** There seems to be no equivalent statement where implication is outside of quantifiers.

Combining with Universal Quantifier If we have $\forall x\ (P(x) \rightarrow Q(x))$, and $\forall x\ P(x)$ is true, then $\forall x\ Q(x)$ is also true. Explanation: $\forall$ commutes with $\wedge$, so $\forall x\ (P(x) \rightarrow Q(x)) \wedge \forall x\ P(x)$ is equal to $\forall x\ (P(x) \rightarrow Q(x)) \wedge P(x)$, which yields $\forall x\ Q(x)$. So, we see that $\forall x(P(x) \rightarrow Q(x)) \vdash (\forall x P(x)) \rightarrow (\forall x Q(x))$.

Combining with Existential Quantifier If we have $\forall x\ (P(x) \rightarrow Q(x))$, and $\exists x\ P(x)$ holds, then $\exists x Q(x)$ is also true. This is more or less obvious, and you can informally check it using the two-valued example. So, we see that $\forall x(P(x) \rightarrow Q(x)) \vdash (\exists x P(x)) \rightarrow (\exists x Q(x))$.

Implication and Existential Quantifier

Similar to the previous part, consider the following three cases:

1. $\exists x(P \rightarrow Q(x))$
2. $\exists x(P(x) \rightarrow Q)$
3. $\exists x(P(x) \rightarrow Q(x))$

In a domain with just two values, they will look like this:

1. $(P \rightarrow Q(x_0)) \vee (P \rightarrow Q(x_1))$
2. $(P(x_0) \rightarrow Q) \vee (P(x_1) \rightarrow Q)$

3. $(P(x_0) \to Q(x_0)) \vee (P(x_1) \to Q(x_1))$

For case 1, we can immediately demonstrate that:

$$\exists x\ (P \to Q(x)) \vdash P \to (\exists x\ Q(x))$$

Assume $\exists x\ (P \to Q(x))$ and that P is true. Then we have $P \wedge (\exists x\ (P \to Q(x)))$, which is the same as $\exists x\ (P \wedge (P \to Q(x)))$. Then we deduce that $\exists x\ Q(x)))$.

Can you demonstrate that the argument does not work in the opposite direction, so there is no equivalence? Try it as an exercise.

In case 2, assume we have $(P(x_0) \to Q) \vee (P(x_1) \to Q)$, and assume $P(x_0) \vee P(x_1)$. What does this give us? Nothing. If we have $P(x_0) \to Q$ and $P(x_1)$, can we deduce Q? No, we cannot. On the other hand, if we have $\exists x(P(x) \to Q$ and $\forall x P(x)$, then we can deduce Q.

Formally:

$$\exists x(P(x) \to Q) \vdash (\forall x P(x)) \to Q$$

For case 3, if we combine it with $\exists x P(x)$, it gives us nothing. Suppose that, for some x, we have $P(x) \to Q(x)$. What's next? Having one such x, without knowing whether $P(x)$ holds, gives us nothing.

Negation and Quantifiers in Intuitionistic Logic

In intuitionistic logic, $\neg P$ is the same as $P \to \bot$. Therefore, everything that we talked about in the previous two sections applies but no more than that. Namely:

- $\forall x \neg P(x) \equiv \neg(\exists x P(x))$
- $\exists x \neg P(x) \vdash \neg(\forall x P(x))$

As you can see, there are no rules that would magically state the existence of an entity without an explicit demonstration of how to reach that entity.

Summing Up

Below are some of the useful properties of quantifiers. These properties are valid for any intuitionistic logic, hence, they are valid for any Boolean logic:

1. $\forall x(P \to Q(x)) \equiv P \to \forall x Q(x)$
2. $\forall x(P(x) \to Q) \equiv (\exists x P(x)) \to Q$

3. a. $\forall x(P(x) \to Q(x)) \vdash (\forall x P(x)) \to (\forall x Q(x))$

   b. $\forall x(P(x) \to Q(x)) \vdash (\exists x P(x)) \to (\exists x Q(x))$

4. $\exists x(P \to Q(x)) \vdash P \to \exists x Q(x)$

5. a. $\exists x(P(x) \to Q) \vdash (\forall x P(x)) \to Q$

   b. $\exists x(P(x) \to Q) \vdash (\exists x P(x)) \to Q$

Combining Quantifiers

Quantifiers of the same kind commute: $\forall x\ \forall y\ P(x, y)$ is the same as $\forall y\ \forall x\ P(x, y)$, and $\exists x \exists y P(x, y)$ is the same as $\exists y \exists x P(x, y)$.

What about $\forall x \exists y\ P(x, y)$ vs. $\exists y \forall x\ P(x, y)$? They don't commute. Let's illustrate this with an informal example. Let $P(x, y)$ stand for "x is a Facebook friend of y." The first quantifier, $\forall x \exists y\ P(x, y)$, says that everybody has a friend on Facebook. The second one, $\exists y \forall x\ P(x, y)$, says that somebody is everybody's friend on Facebook. This may be an invisible friend or, rather, Big Brother (who may not even be unique).

We could go deeper into details by asking what the differences are between the two, but readers can surely investigate them on their own.

Chapter 7. Models and Theories

In this chapter, we discuss theories, models, and relations between them. This is a confusing and controversial area. What is a theory? What is a model? If we turn to physics, these words have a pretty strict meaning, but unfortunately, their meaning in physics is different from their meaning in mathematics.

In mathematics, a theory (informally speaking) is an abstract construct consisting of types, operations, and axioms. A model is some structure that implements a given theory.

There is an old tradition of defining theories in terms of models. Here are examples of that. When people talk about geometry, they may mention "a set of all points such that, etc." But is Euclidean geometry a part of set theory? Or is set theory a part of Euclidean geometry? Neither is a part of another. Set theory is many centuries younger than Euclidean geometry. They are actually unrelated.

We often encounter a definition of a monoid as a set of elements. But, can a monoid be defined without *a* set theory? Moreover, if everything is defined using sets, then how can a set theory be defined without basing it on a set theory? There is obviously something wrong with this approach. A monoid based on a set is actually a model of a monoid, and this model of a monoid is a model in that particular set theory (where the underlying set belongs).

We should definitely find a way to separate one from another, or "an implementation from a declaration," as people say in the programming world. Theories play the role of declarations, and models play the role of implementations.

Theories

Definition: Theory

A *theory* consists of *types*, *operations*, and *axioms*. There are two kinds of operations: *functions* and *relations*. A function has a name and a signature, which includes a list of "argument types" and a "result type." A relation has a name and a list of argument types. Axioms are first-order logic expressions combining functions, relation variables (of various types), and quantifiers. All of these components provide us with a "first order language" (see Chapter 4). In addition to a language, a theory may also have *axioms*, which are formulas in the given language.

When a theory has no axioms, it is called a *free theory*, and it coincides with the first order language based on the theory's functional and relational symbols.

Expressions used in a theory involve identifiers, and each identifier has a type. When an identifier x has type T, this is denoted as $x : T$.

Although you are likely already familiar with some examples of theories, it makes sense to rephrase them in reference to this new aspect.

Example 1. Monoid A monoid theory (T, Op, Z) has only one type, T, and two operations: one binary and one nullary. The nullary operation is the neutral element Z, and the binary operation is Op. Since there is just one type, the binary operation has the signature $(T, T) \to T$, and the nullary operation has the signature $() \to T$.

The axioms of this theory are the following:

- **associativity**
  $\forall a, b, c,\ Op(a, Op(b, c)) = Op(Op(a, b), c)$
- **neutral element**
  $\forall a,\ Op(a, Z) = a\ \wedge\ Op(Z, a) = a$

Example 2. Partial Order A partial order theory (T, lt) has just one type, T, and one relational symbol, lt. The axioms of this theory are the following:

- **antisymmetry**
  $\forall a, b,\ lt(a, b)\ \wedge\ lt(b, a) \to a = b$
- **transitivity**
  $\forall a, b, c,\ lt(a, b)\ \wedge\ lt(b, c) \to lt(a, c)$

Example 3. A Fragment of Euclidean Geometry Euclidean Geometry has a large variety of definitions. In this section, a definition of a small portion of it is introduced, without going too deep into tricky details.

Euclidean Geometry includes the following types:

- P: points
- L: lines
- C: circles

It also includes the following relational and functional symbols:

- $eq(a : P, b : P)$, also written as $a = b$: equality relation for points.
- $eq(a : L, b : L)$, also written as $a = b$: equality relation for lines.
- $eq(a : C, b : C)$, also written as $a = b$: equality relation for circles.
- $between(a : P, b : P, c : P)$: a, b, c are points, and b is between a and c.

- $isOn(a : P, b : L)$: a is a point and lies on line b.
- $isOn(a : P, c : C)$: a is a point and lies on circle c.
- $center(c : C) : P$: c is a circle, and the result is a point that is the circle's center.

Note that there is no function that, given two points, produces a line. That's because every two points don't necessarily produce a unique line. If two points are equal, they don't produce a unique line. We might ask, where do the lines come from, if there's no function to build a line? The answer is that lines and points are independent from one another, but they are connected via certain relations.

- **continuity**
  $\forall p : P, q : P,\ p \neq q \rightarrow\ \exists r : P\ between(p, r, q) \wedge r \neq p \wedge r \neq q$
- **line by two points**
  $\forall p : P, q : P,\ \ \exists a : L\ isOn(p, a) \wedge isOn(q, a)$
- **unique line by two points**
  $\forall p : P, q : P, a : L, b : L,\ p \neq q \wedge isOn(p, a) \wedge isOn(p, b) \wedge isOn(q, a) \wedge isOn(q, b) \rightarrow a = b$
- **parallel lines**
  $\forall p : P, a : L,\ \neg isOn(p, a) \rightarrow \exists b : L\ isOn(p, b) \wedge \forall q : P,\ isOn(q, b) \rightarrow \neg isOn(q, a)$
  Euclidean Geometry has many more axioms, however, building a full geometry theory is not the goal here.

In two of the examples above, we learned that equality, $=$, needs to be defined, and its properties need to be specified in axioms. We could, instead, refactor our examples a bit and extract the general idea of equality as used in theories.

Definition: Equational Theory

A theory is called *equational* if it has an *equality relation* for each type, and the relation is symmetric, reflexive, and associative. For instance, Euclidean Geometry is an equational theory.

Example 4. Peano Arithmetic *Peano Arithmetic* is an equational theory that has one type and two operations: one nullary, 0, and one unary, S ("next" or "succ"). The axioms of this theory are the following:

- $\forall a, b\ Sa = Sb \rightarrow a = b$: that is, S is an injection.
- $\forall a\ Sa \neq 0$: that is, 0 is next to none.

Example 5. Zermelo-Fraenkel Set Theory *Set theory* is an equational theory with one type and one more relational symbol (in addition to $=$), $\in$. This relation is defined for sets only. In Zermel-Fraenkel set theory, members of sets are also sets and nothing else. There are other set theories, in which entities other than sets can exist and be elements of sets. Such elements are called "urelements."

Zermelo-Fraenke set theory has over a dozen axioms:

- **set equality**
  $\forall a, b\ (a = b) \leftrightarrow (\forall c,\ c \in a \leftrightarrow c \in b)$
- **empty set**
  $\exists \emptyset\ \forall a\ \neg a \in \emptyset$
- **union of two sets**
  $\forall a, b\ \exists c\ \forall x\ x \in c\ \leftrightarrow\ (x \in a \vee x \in b)$
- ...

Example 6. Theory of Three Values or More This is a pretty simple equational theory with one type and one axiom:

- $\exists a, b, c\ a \neq b \wedge a \neq c \wedge b \neq c$: Three distinct values exist.

Example 7. Theory of Exactly Three Values This example is the same as above, but one more axiom is added to the one defined above. The theory says that a) there are three distinct values, and b) every value is one of (these) three distinct values.

- $\exists a, b, c\ a \neq b \wedge a \neq c \wedge b \neq c$: Three distinct values exist.
- $\forall a, b, c, d\ a \neq b \wedge a \neq c \wedge b \neq c \rightarrow d = a \vee d = b \vee d = c$: There is no fourth value.

Dealing with Theories

When defining a theory, only types, functions, and relations are defined. If we don't provide any axioms, our theory is a free theory. If we provide axioms, on the other hand, using first-order logic deduction rules, we can try to deduce more propositions out of the existing axioms. These new propositions are called *theorems*. A theorem is not something provided magically out of nowhere but is a consequence of axioms of a certain theory. A theorem belongs to that theory and makes no sense outside of the theory. If you are told that any certain theorem is always true, you should immediately become suspicious. If a theorem is a property of a certain theory, will it still be correct if we remove an axiom? If we revert the axiom (stating its negation), will the theorem hold? And what are the premises in the proof of the theorem? None?

Example 8. A Theorem In Monoid Theory, if, for a given x and for all y, $x\ Op\ y = y$, then $x = Z$ (where Z is the neutral element). We can prove it using the axioms defining a monoid and the properties of equality relation.

In some cases, we can add one more axiom to a given theory. In other cases, it seems impossible.

Definition: Complete Theory

A theory is *complete* if every statement expressed using this theory's terms and logical connectives can be proved to be either true or false.

One might expect that a good mathematical theory must be complete, and this was commonly believed until about 1930, when it was discovered that this is hardly ever the case.

From the view point of those who study number theory and geometry, we would expect that any statement in this theory could either be proved or disproved. However, Kurt Gödel has proven that as soon as a consistent theory involves natural numbers, it must be incomplete. As a result, mathematicians in the twentieth century learned to live with incomplete theories, and without any hope of solving all the problems that can be formulated in a theory.

But let us start with examples.

The theory in Example 7 from the previous section, "a theory of a three-element set" is complete. Mostly this is because there wasn't much to talk about: no functions and no relations except equality. On the other hand, the theory from Example 6 is, of course, not complete. It states that we have at least three values, but there may be four, or five, or just three. The statement that says there is an element number four cannot be proved or disproved.

Euclidean geometry, as formulated in the previous section, does look like a complete theory. We expect from this theory that any theorem of geometry can be proved or disproved. Unfortunately, this is not so. Our expectations that any certain theory is complete may lead to incorrect conclusions.

Example 9. Peano Arithmetic is Not Complete Having defined natural numbers via 0 and $next$, we can easily define addition and multiplication within this theory. With some effort, we can also introduce negative and rational numbers.

Is this arithmetic unique? Can any statement about numbers be proved? Gödel's first theorem states that there cannot be a consistent (no contradictions) and complete theory containing natural numbers. It may look like a daunting task to find an example demonstrating that there is a statement about numbers that we cannot check using just the rules of arithmetic. Fortunately, a relatively simple example exists: Goodstein's theorem.

Goodstein's Theorem Here we will build a sequence of natural numbers that "always ends" in 0, but the fact that it does is not provable in *Peano*.

Begin with a natural number n. Represent it as a sum of powers of 2: $2^{k_1} + 2^{k_2} + ...$ Each k_1 can be represented as a sum of powers of 2. By repeating this process, we eventually arrive at a representation of the number n as the sum of towers (powers of powers) of 2.

For example: $42 = 2^{2^{2^1+1}} + 2^{2^1+1} + 2^1$. Once we have such a tower, apply the following iteration step: replace 2 with 3 in this representation, subtract 1, and then represent the result again as the sum of towers, this time of 3. So, for $2^{2^{2^1+1}} + 2^{2^1+1} + 2^1$ we will produce $3^{3^{3^1+1}} + 3^{3^1+1} + 3 - 1 = 22876792455044$.

Now do the next step, and replace 3 with 4 in the formula, and subtract 1: $3^{3^{3^1+1}} + 3^{3^1+1} + 2 \rightarrow 4^{4^{4^1+1}} + 4^{4^1+1} + 1$ The next number will be
536312317197703883982960999928233845099174632823695735108942457748870561202941879072074971926676137107601274327459442034150155312477862797857345960243374 09

Repeating this operation (replacing 4 with 5, etc., while subtracting 1) will, as proved by the theorem, eventually produce 0 but the intermediate numbers are going to be huge.

This theorem, which states that the sequence eventually produces 0, can be proven but not in Peano arithmetic. So, the question is, if the theorem is formulated in Peano arithmetic, but it cannot be proven using its axioms, how do we know it is true? We don't. The theorem does not follow from Peano axioms.

The proof belongs to a so-called "second-order" arithmetic, which involves not only numbers but also predicates.

Alternatively, we can talk about numbers and "sets of numbers." Of course, no sets from a set theory are involved. We are talking about predicates defined on numbers. Second-order arithmetic is a powerful theory, but it is just a theory.

By tweaking it in the right places, we can arrive at a theory for which Goodstein's theorem is not true. This may remind you of Robert Sheckley's novel, "Mindswap," except in the book it is the reality that varies. In this discussion, it is theories, the products of our mind, that vary instead.

Example 10. Set Theory is Not Complete Set theory was developed, in its more modern and formal versions, as a foundation of all mathematics, except for set theory itself. But since it contains Peano Arithmetic, it is also incomplete. A popular example involves having (or not having) a set that has a size that is bigger than countable and smaller than continuum.

One would expect the answer to this question, whether such an intermediate size set exists, to be definite: either/or. A set either exists or it does not exist. That's what one would expect if we

were dealing with a "reality." Set theory is not a reality of any kind, however, it is just a theory. And it was discovered, in particular, that there is no answer to this question in Zermelo-Fraenkel set theory.

The existence of such a set is just another axiom. The negation of the existence of such a set is an axiom that is as good as the one that states its existence. The issue is known as the continuum hypothesis.

Interestingly, an ISO standard (ISO/IEC 13568:2002) specifies a version of set theory as *Z-notation*. Whether the intermediate set exists or not, according to ISO, it is impossible to determine. This axiom is not included in the definition of *Z-notation*. Since *Z-notation* is also used as a part of *TLA+*, a language for temporal logic, we can expect that *TLA+* is as incomplete as anything using a set theory (or Peano arithmetic).

Algebraic and Geometric Theories

Theories can be split into two distinct sorts: algebraic theories and geometric theories.

Definition: Algebraic Theory

A *(finitary) algebraic theory* is an equational theory that consists of *types*, *functions*, and *equations*. A function has a name and a signature. A function's signature is a list of the function's "argument types" and the "result type" of this function. Axioms of an algebraic theory have the form of equations: $f(a_1, a_2, ...a_n) = g(b_1, b_2, ..., b_m)$. Of course, the equality $=$ is a binary rilation, but that's the only relation, and it is only used in axioms that have a form of equations.

Example 11. Algebraic Theories. If we return to examples 1-7 above, we can easily classify their theories as algebraic and not algebraic. First, the theory of *monoids* is definitely an algebraic theory. It has a couple of functional symbols (of arity 0 and 2) and two axioms connecting them. A *partial order*, on the other hand, is definitely not an algebraic theory. There, we have a binary relation, not a function. We could throw in a *Bool* type for the results of comparison, but it would not eliminate the axioms of a *partial order*, which cannot be reduced to equations.

Neither Euclidean geometry, nor Peano arithmetic, nor set theory are algebraic. How about the two remaining examples, "three or more" and "exactly three"? We could try to turn the "three or more" theory into an algebraic form by introducing three nullary operations, which would give us a, b, c. This won't help though, since the axiom of these three elements being distinct goes way beyond the capabilities of algebraic theories.

Definition: Geometric Theory

A *geometric theory* is a synonym for a theory as defined in the beginning of this chapter. Reviewing this definition, we see that an algebraic theory is just a special case of a geometric theory. Some theories are not algebraic, and to disambiguate them, we mention that it is a geometric theory. For example, Euclidean geometry is geometric, set theory is geometric, and Peano arithmetic is geometric.

Look, for instance, at Peano arithmetic (Example 4). One of its axioms is an implication (s being an injection), and another is a negation. An implication can be expressed via negation, and a negation can be expressed via implication, but neither can be represented as an equation Therefore, an algebraic theory of Peano arithmetic is impossible.

Models

Models of theories are defined in this section. The definition is not universal, as we are limited to modeling only what we can model. Generally speaking, we would expect a *model* of a theory A in a theory B to be defined as a mapping of A to B that preserves all functional symbols, all relations, and all axioms. Unfortunately, this generalization is hard to define, so we start with a simple case.

Definition: Model of an Algebraic Theory

A *model* of an algebraic theory A in a set theory $\mathcal{S}et$ consists of mapping types to sets (that is, assigning each type in A some set $M[A]$) and mapping functional symbols of signature $(A_1, A_2, ...A_n) \to B$ to set functions $M[A_1] \times M[A_2] \times ... \times M[A_n] \to M[B]$. The mappings should satisfy the equations defined in the theory A.

For a given model, all the theorems of the underlying theory must be valid in this model.

The opposite statement is not necessarily true. Suppose we have a property in a theory, and this property holds (is provable) in all possible models of the theory in sets. Does it mean that the property is provable? No, it does not. The property may not be a theorem, and a model can be built where it does not hold. See, for example, the Goodstein theorem. One can find a discussion on Mathoverflow regarding how to build a model of Peano Arithmetic where Goodstein Theorem is false.

Example 12. Model of a Monoid For the monoid theory, a model would be any set X with an element $z \in X$ and an operation $f : \ X \times X \to X$, such that $\forall x \ f(z, x) = f(x, z) = x$ and

$f(a, f(b, c)) = f(f(a, b), c)$. This is a familiar monoid or, rather, a familiar model of a monoid in $\mathcal{S}et$.

Note, any such structure is a model of a monoid, e.g., $\mathbb{Z}_{10}$, integer numbers modulo 10, with addition as a binary operation and with 0 as a neutral element is a model. Given an alphabet set A, the set A^* of all strings is also a model of a monoid.

Also note that the theorem about the uniqueness of the neutral element holds in any model of a monoid.

Definition: Model of a Geometric Theory

A *model* of a (geometric) theory A in a set theory $\mathcal{S}et$ consists of mapping types to sets (that is, assigning each type in A some set $M[A]$), mapping functional symbols of signature $(A_1, A_2, ...A_n) \to B$ to set functions $M[A_1] \times M[A_2] \times ... \times M[A_n] \to M[B]$, and the relational symbols of signature $A_1, A_2, ...A_n$ to n-ary relations over $M[A_1] \times M[A_2] \times ... \times M[A_n]$. The mappings should satisfy the axioms defined in the theory A.

Example 13. Model of a Three-element Set To provide a model of this theory we will need any set consisting of exactly three elements.

Example 14. Model of a Partial Order For the partial order theory, a model would be any set X with binary relation $x < y$ on it. Satisfying the axioms would mean that the relation is antisymmetric and transitive.

Counterexample 15. Model of Set theory It is a known fact that a set theory, generally speaking, cannot be modeled in itself. To do this, we would need a set of all sets. This does not mean set theory cannot be modeled, it only means that Set Theory cannot be modeled in sets. We would need another theory. Such theories exist: the von Neumann model of set theory is a good example. Anything that can be modeled in a set theory can be modeled in that theory, as modeling is transitive.

Conclusion

This chapter only touched the surface of theories and models. There are more than two approaches to this discussion. The approach discussed so far is closer to Lawvere's categorical approach. We could not go deeper into Lawvere's model theory, because doing so would require a knowledge of toposes. Topos theory is based on category theory, so there are a couple of steps required before we can study model theory in depth.

Chapter 8. Category: Multi-Tiered Monoid

Monoid of Functions

Probably the most interesting of all monoids is the monoid of functions $X \to X$, for which the binary operation is defined as composition of functions, and the identity, id_X, takes the role of a neutral element. Note that we can either take all possible functions from X to X, thus ensuring that it is closed under composition, or we can use a certain kind of function. In the latter case, we will need to have id_X in this collection of functions, and if $f : X \to X$ and $g : X \to X$ are included, their composition must be included, as well.

In fact, any monoid can be represented as a monoid of functions. Take a monoid (T, Z, Op) and use T as the domain of functions. How do we represent elements of T as functions on T? Through multiplication! Every element t of T turns into a function, $f(t)$, that maps every other element, s, to $f(t)(s) = t\ Op\ s$.

Clearly, Z maps to an identity function, just by its property. Given x and y, and $f(x)$ and $f(y)$, what happens with the result of the operation, $x\ Op\ y$? It should map to $f(x) \circ f(y)$ and checking that it does is easy.

Of course, not every function defined on the collection of monoid values can be built from the elements of the monoid this way. Look at the following example:

Example 1 The set of natural numbers, with the addition operation and with 0 as a neutral element forms a monoid. Any natural number n produces a function $x \mapsto n + x$. Nevertheless, not every function defined on natural numbers is adding a constant to the argument.

More Than One Domain

What if, instead of having exactly one object and a monoid of functions, we add one more object, Y? Now we can have functions of four kinds:

$X \to X$, $X \to Y$, $Y \to X$ $Y \to Y$

We are not in a monoid anymore. Not every function between X and Y can be composed with every other function, so Op, that is, the composition operation, $\circ$, is now a partial operation. Meaning, it is not defined for all possible pairs of elements. Moreover, we now have two distinct neutral elements: id_X and id_Y.

Without introducing any more complexity, we can add more domains and functions between them—just keep in mind that composition is not defined for any pair, and that there must be as many identity functions as there are objects.

Example 2 Take all possible (finite-dimensional) vector spaces as objects and linear transformations between them. Instead of linear transformations, we can talk about matrices. Two matrices can be multiplied if their respective dimensions are the same, and for each n, there is a unit matrix $n \times n$, such that it is neutral for matrix multiplication. We do not have a monoid, but the structure is somewhat similar.

Example 3 Remember Partial Order? We can look at every relation $a \leqslant b$ as a function $a \to b$.

Composition is not a problem: if $a \leqslant b$ and $b \leqslant c$, then we have a "function" that serves as a composition $a \leqslant c$. Since there is, at most, one "function" from a to b, associativity is not a problem.

Now let's elaborate on this idea. We need functions and a collection of objects that serve as domains/codomains for functions. In the case of just one object, we have a monoid. If there is more than one object, the operation of composition has to be partial, and there is an identity for each object. We also need associativity.

In the example above, $a \leqslant b$ is not a function but just an edge connecting two nodes. That's why in category theory the notion of "function" is generalized and the term "arrow" is used. Another, synonymous, term for an arrow is "morphism."

Definition: Category

A *category* $\mathcal{C}$ consists of a collection of *objects*, $\mathcal{C}_0$, and a collection of *arrows*, $\mathcal{C}_1$, that satisfy the following constraints:

- Each arrow $f \in C_1$ has a domain $X = dom(f) \in C_0$ and a codomain $Y = cod(f) \in C_0$. This fact is also denoted as $f : X \to Y$.
- For every $f : X \to Y$ and $g : Y \to Z$, their composition is defined, $h = g \circ f : X \to Z$.
- Composition is associative: $h \circ (g \circ f) = (h \circ g) \circ f$.
- Identity arrows are defined for each object, so that $id_Y \circ f = f = f \circ id_X$.

Arrows are abstract notions and do not necessarily take "elements" of one object and produce elements of another. All we know about an arrow is its signature, $f : X \to Y$.

Examples

We start with tiny examples and then move on to bigger ones.

Examples of Finite Categories

Example 4. Empty category An empty category does not have objects or arrows. It is just empty.

Example 5. Category $\mathbb{1}$ It has one object and one arrow (identity).

Example 6. Discrete Category A category denoted as $\mathbb{1} + \mathbb{1} + ... + \mathbb{1}$ consists of n objects and n arrows (identities). Since none of these arrows have a common domain/codomain, there is nothing to compose except identities with themselves.

This kind of category can represent sets. A fragment of set theory can be viewed as a part of category theory. Of course, most of the axioms of set theory are missing, so there is not much we can do with such sets. For instance, a power set cannot be guaranteed to exist, or a subset may not exist for a predicate, since the comprehension axiom is not defined for categories.

Example 7. Category $\mathbb{2}$ It has two objects and three arrows and can be schematically drawn as $* \to *$.

Note that we do not care about the names of the two objects or about their nature. We can actually give them names, e.g., 0 and 1. Then the whole category $\mathbb{2}$ can be represented as $0 \to 1$. The "true nature" of the objects does not matter, similar to counting 1+2=3, whether we are counting apples or oranges.

Example 8. Category $\mathbb{3}$ It has three objects and six arrows:

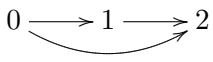


Figure 23: *Category $\mathbb{3}$.*

Identities are not shown, and arrows don't have names, but they are all unique, so we do not care (yet). And the names of objects—$0, 1, and 2$—are arbitrary.

Example 9. Category $\mathbb{4}$ It has four objects and 10 arrows. Look at the picture below.

This category has enough arrows to have non-trivial associativity involved. Since 03 is unique, it has no choice but to be equal to both $23 \circ 02$ and $13 \circ 01$.

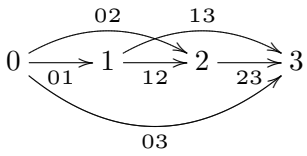


Figure 24: *Catetory* $\mathbb{4}$*. Identities not shown.*

Examples of Bigger Categories

Example 10. SQL Tables

```
create type kind as enum ('cat', 'dog', 'fly', 'hamster', 'e.coli');
create table Person (id bigint, name varchar(80), pet bigint,
  primary key (id),
  constraint pet_fk foreign key (pet references Animal(id)));

create table Animal (id bigint, kind kind, name varchar(80),
  owner bigint, primary key (id),
  constraint owner_pk foreign key owner references Person(id)));
```

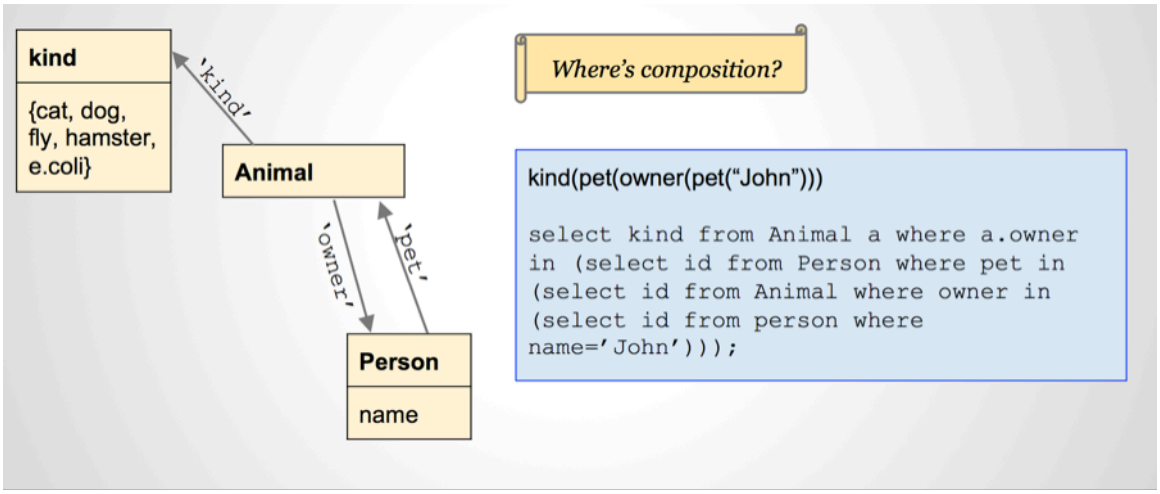


Figure 25: *Animal kingdom, limited.*

Example 11. Graph Given a (directed multi-) graph, $G = (N,\ E,\ from : E \to N,\ to : E \to N)$, we can build a category of paths on this graph. All the nodes serve as objects, and all possible paths serve as arrows. Composition is defined via concatenation, which is associative.

The Pet Database (Example 10) gives us a chance to look into a specific example of such a graph.

Example 12. Partial Order Any (non-strict) partial order can be represented as a category. A relation $a \leqslant b$ is an arrow from a to b. Associativity and identity are automatically provided.

Example 13. Category of Integer Numbers, Ordered This category is denoted as $\mathbb{Z}$. Its objects are integer numbers, and it is a partial order.

Example 14. Category of Real Numbers, Ordered This category is called $\mathbb{R}$. Its objects are real numbers, and it is a partial order.

Example 15. Monoids Any monoid is a category and has just one object x (of abstract nature). All elements of the monoid are interpreted as arrows, $x \to x$. The monoid's neutral element behaves as the identity arrow. The monoid's associativity ensures the associativity of the arrows under composition.

Example 16. Category of Sets and Functions If, given some set theory (there are plenty of them), we take sets as objects and their regular set arrows as arrows, we get a *category of sets*, $\mathcal{S}et$. This category has many interesting features, and may be used as a basis for building Boolean logic and as a foundation for a lot of mathematics.

Example 17. Category of Sets and Partial Functions Similar to the previous example, take sets as objects, but for arrows—instead of functions—take partial functions. A partial function from set X to set Y is a function that is defined on some subset $X_0 \subset X$ and takes values in Y. X_0 can even be empty, so that the partial function is not defined at all!

Z-notation uses the arrow symbol, $\nrightarrow$, to denote a partial function.

Every regular function is a partial function too, when $X_0 = X$.

Composition of partial functions is defined like this: for $f : X \nrightarrow Y$ and $g : Y \nrightarrow Z$, where f is defined on X_0 and g is defined on Y_0, we take $X_2 = \{x \mid x \in X_0 \wedge f(x) \in Y_0\}$ and map $x \in X_2$ to $g(f(x))$.

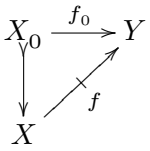


Figure 26: *Partial function.*

Associativity follows from this definition. If we use regular set identity functions as identity arrows in this category, it is clear that they will be neutral for composition. The following diagram shows composition of partial functions.

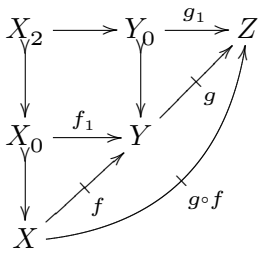


Figure 27: *Composition of partial functions.*

We now have a category denoted as $\mathcal{S}et_{Part}$.

In programming, many functions we are dealing with are actually partial functions. These functions may break or throw exceptions on values outside of their reasonable domains. Somehow, almost all programmers are used to this, but we don't need to be. Instead, we can treat them properly as arrows in a category of partial functions.

In Scala, partial functions are a part of the language. The type is `PartialFunction[X, Y]`, and such a function can be defined in Scala like this:

```
val inverse: PartialFunction[Double, Double] = {
  case d if math.abs(d) > eps => 1/d
}
```

How can we use this kind of function? One method would involve checking whether the function is defined for a value, e.g.:

```
if (inverse.isDefinedAt(42.0)) {
  println(s"inverse of 42.0 is ${inverse(42.0)}") }
else { println("Oops") }
```

Example 18. Category of Sets and Binary Relations This category is denoted as $\mathcal{R}el$. Its objects are plain sets, and its arrows are binary relations, also known as "many-to-many" relations. Composing binary relations is not difficult. Given two binary relations, $R \subset X \times Y$ and $S \subset Y \times Z$, their composition, $S \circ R \subset X \times Z$, is defined as $\{(x, z) \in X \times Z \mid \exists\, y \in Y : xRy \wedge ySz\}$.

Associativity follows from natural properties of quantifiers. The regular unit arrow, or rather its graph, also known as a diagonal, $\Delta \subset X = \{(x, x) \mid x \in X\}$, is obviously the identity.

We encounter binary relations in relational databases, and the definition of composition could well be rewritten as `select distinct R.x, S.z from R,S where R.y = S.y`.

Example 19. Vector Spaces and Linear Transformations We already discussed this category in Example 2, but now we will be more formal. It is denoted as $\mathcal{F}in\mathcal{V}ect$. Its objects are vector spaces of finite dimensions, and its arrows are linear transformations, which can loosely be associated with matrices (for finite-dimensional spaces). A composition of two linear transformations is a transformation represented by the product of two matrices. The unit matrix, $n \times n$, represents the identity arrow on an n-dimensional vector space.

Chapter 9. Working with Categories

Arrows in a Category

A category is defined as consisting of objects and arrows. An "arrow" is an abstraction of function. It does not have to be a function in any usual sense. An arrow does not have to take values and produce other values. All we know about an arrow is its domain, its codomain, and how it composes with other arrows. Can we talk about them based on this small amount of data? As it turns out, yes—the ability of arrows to compose provides for a rich realm of ideas.

Remember that objects are not necessarily sets, so they don't have to consist of elements, and an arrow does not have to be defined by its application to elements, even if objects do consist of elements.

Since we cannot always define arrows in a category via their actions on elements, we must come up with definitions that are based on compositions only.

Monomorphism

When talking about sets and functions, the notion of *monomorphism*, aka *injection*, was introduced.

For two sets, A and B, an arrow $f : A \to B$ is called an *injection* if from $f(a_1) = f(a_2)$ it follows that $a_1 = a_2$. Since "elements" are not always easily provided, like they were in the category of sets, we need an alternative definition.

Definition An arrow $f : A \to B$ is a *monomorphism*, if for any pair $g_1, g_2 : X \to A$, from $f \circ g_1 = f \circ g_2$, it follows that $g_1 = g_2$.

This does not appear to be similar to the definition of a monomorphism for sets, but the two are actually equivalent when we use them in the category of sets.

How can we demonstrate the equivalence of the two definitions, in the case of sets?

- Suppose we are dealing with sets, and an arrow f is a monomorphism. Take two elements, a_1 and a_2, and define two arrows, g_1 and g_2, from a singleton $\{.\}$ to A: $g_1(.) = a_1$, and $g_2(.) = a_2$. Then, compose these two arrows with f. We will have $(f \circ g_1)(.) = f(a_1)$ and $(f \circ g_2)(.) = f(a_2)$. Suppose $f(a_1) = f(a_2)$. It is the same as having $f \circ g_1 = f \circ g_2$. Since f is a monomorphism, $g_1 = g_2$, that is, $a_1 = a_2$—and now we have an injection.
- Now suppose we have an injection f. In this case, for every two arrows g_1, g_2 such that $f \circ g_1 = f \circ g_2$, we have $f(g_1(x)) = f(g_2(x))$, and so $g_1(x) = g_2(x)$ for every x.

If we look at our examples in the previous chapter, we can find many cases where every arrow is a monomorphism. For instance, in partial orders there is not more than one arrow from one object to another, so uniqueness is guaranteed.

As an exercise, can you prove that identities are monomorphisms?

Another exercise: can you prove that a composition of two monomorphisms is a monomorphism?

With sets, an inclusion of a subset into a set is a special kind of monomorphism. In general settings, there is no such thing as inclusion, and on most occasions, we can only say that an object A is a sub-object of an object B if there is a known monomorphism from A to B.

Epimorphism

The definition of *surjection* (*epimorphism*) in sets requires that we provide, for each $b : B$, an $a : A$ such that $f(a) = b$. However, if we don't have enough elements, we will need to try something different.

Definition An arrow $f : A \to B$ is called an *epimorphism* if, given any pair $g_1, g_2 : B \to C$ such that $g_1 \circ f = g_2 \circ f$, it follows that $g_1 = g_2$.

How is this definition related to the old set-theoretic definition? Look closer. If, in the case of sets, every $b \in B$ is equal to $f(a)$ for some $a \in A$, and the values of g_1 and g_2 on such b are the same, it means that the values of g_1 and g_2 are the same on every $b \in B$. So, according to the definition of functions in sets, they are equal.

On the other hand, in sets, if a b exists for which there is no a that is not equal to $f(a)$, we can introduce a function that is $false$ for this specific b and $true$ for every value $f(a)$. Such a function, composed with f, is equal to a constant $true$ composed with f. If these two functions, the composition and the constant $true$, are equal, no such b can exist.

The advantage of this definition is that we do not need to rely on the existence of anything; it is defined using a so-called universal property, similar to the one used in the definition of monomorphism.

Again, in partial orders, every arrow is an epimorphism, due to the uniqueness of arrows between objects.

Can you prove that identities are epimorphisms?

Can you prove that a composition of two epimorphisms is an epimorphism?

Isomorphism

Definition: Isomorphism An arrow $f : A \to B$ is an *isomorphism* if it has an inverse, $g : B \to A$, such that $g \circ f = id_A$, and $f \circ g = id_B$.

Every isomorphism is both an epi and a mono, but if an arrow is both an epi and a mono, this does not mean it is an isomorphism. Remember partial order? Every arrow in a partial order as a category is an epi and a mono, but only identities are isomorphisms.

It is easy to see that an inverse of an isomorphism is also an isomorphism, and a composition of two isomorphisms is an isomorphism.

Definition: Isomorphic Objects Two objects, X and Y, in a category $\mathcal{C}$, are *isomorphic* if there is an isomorphism between them. This relation is denoted as $X \cong Y$. Note that this is an equivalence relation: X is isomorphic to itself. If $X \cong Y$, then $Y \cong X$; if $X \cong Y$ and $Y \cong Z$, then $X \cong Z$.

Many notions of category theory are defined up to an isomorphism. We will see examples in the next section. The term "definition up to an isomorphism" means the following: given a definition, it does not specify an object uniquely. But any object satisfying the definition will be isomorphic to any other such object. Besides, if two objects are isomorphic, and one satisfies the definition, another one must also satisfy the definition. The reason behind this is that category theory is not equational. We don't have equality for objects.

Example 1. Three Isomorphic Objects In the category pictured in the following figure, objects a, b, and c are all isomorphic, but they are definitely not equal to one another.

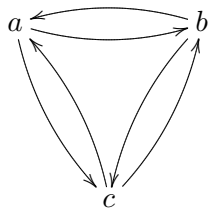


Figure 28: *Category with three isomorphic objects.*

Example 2. Isomorphisms in the Category of Sets In this category, every single-element set (singleton) is isomorphic (but not equal) to every other single-element set.

Initial and Terminal Objects

If you look at category $\mathbb{3}$:

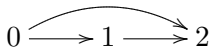


Figure 29: *Category* $\mathbb{3}$.

you will notice that the object 0 has a special feature—there is exactly one arrow from it to each object (including itself). In addition, the object 2 has the opposite property: exactly one arrow coming into it from each object.

Definition: Terminal Object An object T of category $\mathcal{C}$ is a *terminal object* if, for any object X, there is a unique arrow $f : X \to T$.

Note that because every arrow to T is unique, there is just one arrow $T \to T$, and this arrow must be an identity $id : T \to T$.

We can easily see that any two terminal objects are isomorphic: if we have T_1 and T_2, there is a unique arrow from T_1 to T_2 and vice versa. Their compositions must be identities (on T_1 and T_2 respectively). So, these arrows are isomorphisms, and the objects T_1 and T_2 are isomorphic.

Definition: Initial Object An object I of category $\mathcal{C}$ is an *initial object* if, for any object X, there is a unique arrow $f : I \to X$.

Similar to terminal objects, an initial object has only one arrow to itself, an identity, and if we find another initial object, these two will be isomorphic. Initial objects are defined up to an isomorphism.

Example 3. Initial and Terminal Object in Sets In a category $\mathcal{S}et$, the empty set plays the role of an initial object, and any singleton plays the role of a terminal object. The empty set is unique and is a subset of any set. This standard universal inclusion makes it initial. Singletons are terminal objects in sets. Given a singleton, we can define an arrow from any set to this singleton, and there is exactly one way to do so.

Example 4. Initial and Terminal Object in Monoids A unit monoid, 1, the one with just one element, is a good candidate for a terminal object. One can easily see that for each monoid M, there is exactly one arrow from M to 1.

Now, how about an initial monoid? Any monoid should have a neutral element, meaning that there is no such thing as an empty monoid. Moreover, there is always a unique function from 1 to M, for every monoid M: this function maps the neutral element to the neutral element. So, it turns out, 1 is both an initial and a terminal object in the category of monoids!

Example 5. Initial and Terminal Object in Category $\mathbb{1} + \mathbb{1}$ This discrete category has two objects and no arrows beyond the identities of these objects.

In this category, no object can be initial or terminal, because there are not enough arrows.

Example 6. Initial and Terminal Object in Partial Order A partial order may have a terminal object and an initial object. They are usually called *top* and *bottom* and are denoted as $\top$ and $\bot$. See Chapter 3.

Here are a couple of partial orders, one with a terminal object and another with an initial object:

Figure 30: *Two partial orders.*

Example 7. Initial and Terminal Object in the Scala Programming Language The Scala programming language can be viewed as a category, in which types play the role of objects and single-argument functions play the role of arrows.

Scala has a pretty rich type system. In particular, it has `Unit` and `Nothing` types that play the roles of terminal and initial objects, respectively. `Nothing` can be cast to any type, and that is all that can be done with `Nothing`, since it has no values. Therefore, `Nothing` fits the definition of an initial object.

Another special type, `Unit`, plays the role of default return type for a function, an analog of `void` in Java. Every type can be cast to `Unit`, and (if we ignore side effects) there is just one value of type `Unit`. This means that any two "pure" functions `f1: T => Unit` and `f2: T => Unit` are, from the point of view of functional programming, the same. Of course, their side effects—if there are any—may still be different.

Chapter 10. Manipulating Objects in a Category

Product of Two Objects

The category of sets is always a good prototype for generalization. If we have two sets, X and Y, we can build their Cartesian product $X \times Y$. The product is a set of pairs, $\{(x, y) \mid x \in X, y \in Y\}$.

In spite of the "multiplication" notation that we use, the operation of forming a Cartesian product of two sets is not associative: $\{((x, y), z)\}$ is not the same as $\{(x, (y, z))\}$, which means that the elements of $(X \times Y) \times Z$ are different from the elements of $X \times (Y \times Z)$. Instead of equality, we can provide a canonical one-to-one correspondence, that is, an isomorphism, between the two:

$$X \times (Y \times Z) \cong (X \times Y) \times Z$$

So, there is an associativity, defined *up-to-an-isomorphism*.

Similarly, if we build a product of a set X with a singleton $\{\cdot\}$, the result is *isomorphic* to the original set: $\{\cdot\} \times X \cong X \times \{\cdot\} \cong X$ for any set X.

This definition of Cartesian product cannot be generalized literally for an arbitrary category where objects do not necessarily consist of elements. Lacking elements in this sense, a good approach would be to generalize the *idea* of elements of an object X to arbitrary *arrows* pointing to X. These arrows can be viewed as *generalized elements*.

Can we perform this replacement for sets, as well?

For sets, we can say that a product of two sets—X and Y—is also a set, such that any function $f : Z \to X \times Y$ is in a one-to-one correspondence with pairs of functions $(f_X : Z \to X, f_Y : Z \to Y)$, so that $f(z) = (f_X(z), f_Y(z))$. See, no elements involved!

Why is this an equivalent definition? First, if we take $Z = \{\cdot\}$, we see that for this specific kind of Z, the definitions are equivalent. What about an arbitrary set Z? Remember, any such Z is a set, and consists of points, representable as $\{\cdot\} \to Z$. If our definition of the product works for points, it will work for any element of Z, that is, it will work for the whole Z.

Product in a Category

Now, we can try to define a product in general settings by means of arrows and universal properties.

Definition: Product

Given a category $\mathcal{C}$ and two objects of $\mathcal{C}$, X and Y, their *product* is defined as an object $X \times Y$, together with two arrows, $p_X : X \times Y \to X$ and $p_Y : X \times Y \to Y$, with the following *universal property*: for any object Z and two arrows $f : Z \to X$ and $g : Z \to Y$, there is a unique arrow $h : Z \to X \times Y$, such that $f = p_X \circ h$ and $g = p_Y \circ h$.

Graphically, it looks like this:

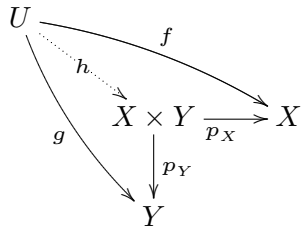


Figure 31: *Cartesian product.*

We will use the term "product" instead of "Cartesian product." These two terms are equivalent.

In a category of sets, this is exactly the definition given at the beginning of this section. However, we are not restricted to objects being sets. We don't have to assume the existence of products. In some cases, products may not exist, and in other cases, we just don't know whether they exist.

Note that if two objects satisfy the definition of a product of X and Y, they are isomorphic, due to the uniqueness of the arrow h defined by (f, g). This means that a product of two objects may not be unique, but all such products are isomorphic.

For example, if we take $X \times Y$ and two arrows, (p_X, p_Y), to X and Y respectively, and take $Y \times X$ with two arrows (q_X, q_Y) to X and Y, we will get, according to the definition above, a unique arrow $swap_{X,Y}$ from $Y \times X$ to $X \times Y$, such that the $p_X \circ swap_{X,Y} = q_Y$ and $p_Y \circ swap = q_X$. Obviously, this $swap_{X,Y}$ has an inverse, $swap_{Y,X}$, so the two products, $X \times Y$ and $Y \times X$, are isomorphic.

Similarly, we can demonstrate that the product is, up to a canonical isomorphism, associative: $(X \times Y) \times Z \cong X \times (Y \times Z)$.

We call p_X and p_Y *projections*, and this may give the impression that they are epimorphisms. This is not always so: if you build a product of an empty set with any set X, the second projection, p_X, is an epimorphism only in one case: when X is empty, too.

Since, for any given objects X and Y, their product, even if it exists, is not necessarily uniquely

defined, it would have been more correct to call a product not an operation but a relation: "Z satisfies the conditions of being a product of X and Y." Currently, though, this approach is not very widespread, and further on we'll be looking at a product of two objects as if it were produced by applying an operation called "Cartesian product." Such an operation can be defined in $\mathcal{S}et$ and in some other categories.

Let's walk through some examples of products.

Example 1. Tuples in Programming Languages Given two types, `T` and `U`, their product type is the type of *tuple*, `(T,U)`. In Scala you can alternatively denote it as `Tuple2[T,U]`. For example,

```
scala> val x: Tuple2[String, Int] = (”a”, 4)
x: (String, Int) = (a,4)
```

Projections to the component types for this product are defined in Scala as `_1` and `_2`: `x._1==“a”` and `x._2==4`.

Example 2. Product via Scala `For` Comprehension Suppose that, instead of involving Scala's entire realm of objects as a category, we are limited by a typeclass `Set[T]`, that is, a subcategory of Scala sets for all possible types `T` and Scala functions between these sets. Since, in Scala, one can produce sophisticated data collections by just using loops (with Scala's "`For` comprehension"), we can express our idea using such a loop. In the following example, given two sets, a set that is actually a product of the two is built:

```
def product[X, Y](xs: Set[X], ys: Set[Y]): Set[(X, Y)] =
  for {
    x <- xs
    y <- ys
  } yield (x, y)
```

Example 3. Data Structures Unnamed types are not the only solution to building product types. Any data structure, e.g., in `C`,

```
struct S {
  int id;
  String name;
  Date date_of_birth;
}
```

can be thought of as a product type. The base types of the product components are `Int`, `String`, and `Date`, and we have projections from `S` to these types. The projections are named `id`, `name`, and `date_of_birth`.

Why is this structure a product? Given a type `T` and three functions, `i: T => Int`, `n: T => String`, and `d: T => Date` (using Scala notation), we can produce a unique function (constructor) `s: T => S`, by defining `s(t) = new S(i(t), n(t), d(t))`, so that composing with projections results in the original functions. This makes the structure universal, as required by the definition.

Example 4. Cross Join in SQL A relational database is a category, as we saw earlier, if we consider tables, views, and statements as objects.

A *cross join* is a table produced by the following statement:

```
select * from table1, table2;
```

Since no conditions are being added, all possible pairs of table rows are included. Categorically, every pair of references to rows in `table1` and `table2` can provide a reference to a row in this product.

Example 5 Categories $\mathbb{1} + \mathbb{1} + ... + \mathbb{1}$. The only products that exist in this category are a product of object with themselves, and any such product is equal to the original object.

Example 6 Category 2. For this category, we have two objects, 0 and 1, and we need to figure out the following products: 0×0, 0×1, and 1×1. The first one, 0×0, amounts to 0, as it has two "projections" to itself, and since there is no other arrow into 0, there are no other products in this category. A bit more elaboration leads us to the conclusion that 0×1 is 0 and 1×1 is 1.

Example 7 Consider any nonempty partial order P as a category. Choose any two objects of this category, X and Y. Suppose a product, $X \times Y$, exists. The existence of projections from $X \times Y$ to X and to Y means that $X \times Y \leqslant X$ and $X \times Y \leqslant Y$. So far, so good. Now if we have an object Z with arrows to X and to Y, it means that, in our partial order P, $Z \leqslant X$ and $Z \leqslant Y$. By the definition of product, every such Z should satisfy $Z \leqslant X \times Y$. This is exactly the definition of the greatest lower bound of X and Y.

Example 8 Given any set A, take its lattice of subsets, $P(A)$. View it as a partial order, hence, a category. We already know from the previous example that in such a category $g.l.b.(X, Y)$ plays the role of $X \times Y$.

What is this product, that is, the g.l.b., for two subsets? It is their intersection, $X \cap Y$. Although we are dealing with sets, the category is different in this example—it is $P(A)$, not $\mathcal{S}et$. So the product in this category is not the classical product of two sets but, rather, the greatest lower bound of the two.

Example 9 Consider a monoid M as a category with one object. The only candidate for this object's product with itself must be the same object. But is it a product? To have $X \times X = X$, we need projection arrows, which are obviously identities (neutral elements of the monoid). For any two f and g, there must be an h such that $f = h$ and $g = h$. Tough luck. This is only possible if we have a trivial monoid (that is, having only one element). No other monoid, viewed as a category, can have a product.

Example 10 A Heyting algebra, H, as a partial order, is a category. In such a category, $a \wedge b$ plays the role of a product. Why is it a product? First, we have $a \wedge b \leqslant a$ and $a \wedge b \leqslant b$—so we have two "projection" arrows. Second, if $c \leqslant a$ and $c \leqslant b$, then $c \leqslant (a \wedge b)$. This is the universal property of a product.

Neutral Element for Cartesian Product Operation

We already observed that a product is, up to an isomorphism, associative. To make it look more like a monoid, we need a neutral element. If the category has a terminal object, this terminal object plays the role of a neutral element for Cartesian categories, and we can have one: it is a terminal object (if it exists). From now on, a terminal object will be denoted as 1. Remember that a terminal object may not be unique, but all such instances of terminal objects are isomorphic to one another.

Now, how is it that $1 \times X \cong X$? Take any Z and two arrows, $f : Z \to 1$ and $g : Z \to X$. The first one is unique. So, having $g : Z \to X$ is enough to define the pair, (f, g). We see that g followed by id_X is g, and f followed by the projection $X \to 1$ is the same $f : Z \to 1$. Therefore, X is a valid instance of $1 \times X$ (remember, a product is defined up to an isomorphism).

As we now have enough examples and illustrations of Cartesian products, in a variety of categories, we can try to figure out whether (and how) set union can be generalized to arbitrary objects of an arbitrary category.

Sum of Two Objects

Some programming languages have union types, in order to reflect the idea that values may belong to either one type or another. E.g., in Haskell this idea is implemented like this:

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

This expression, for those who are not familiar with Haskell, says that, given two types, `a` and `b`, a *data type* `Either a b` is defined, and instances of this type are represented either with the constructor `Left a` or with the constructor `Right b`. `Left a` references a value of type `a`, and has the marker `Left`. Similarly, `Right b` references a value of `b` and has the marker `Right`. The same idea is expressed more verbosely in Scala as:

```
trait Either[A, B]
case class Left[A, B](a: A) extends Either[A, B]
case class Right[A, B](b: B) extends Either[A, B]
```

If we translate this into sets, we get a disjoint union of two sets. For instance, a disjoint union of a set A with itself, $A + A$, consists of elements of A—two copies of each—with a label indicating whether we are dealing with a "left copy" or a "right copy." Actually, we could express this idea using a product of A with $\{0, 1\}$. We would then obtain the following set: $\{(a, i) \mid a \in A, i \in \{0, 1\}\}$, which is equivalent to having a union of two labeled copies of A.

The question now is how to generalize it to any category. Given a category $\mathcal{C}$, and two objects in it, X and Y, what would be their sum, $X+Y$? We will need two inclusions, $i_X : X \to X+Y$ and $i_Y : Y \to X+Y$. But what is the universal property making $X+Y$ unique? This object, together with the pair (i_X, i_Y), must be the closest to X and Y among all possible triples $(Z, X \to Z, X \to Z)$. This is similar to the the product of X and Y, which is the closest to X and Y among all possible triples $(A, A \to X, A \to Y)$.

Definition: Disjoint Sum

Given a category $\mathcal{C}$ and two objects in it, X and Y, their *disjoint sum* (or just *sum*) is an object denoted as $X + Y$, together with two arrows, $i_X : X \to X + Y$ and $i_Y : Y \to X + Y$, with the following *universal property*: for any object Z and two arrows $f : X \to Z$ and $g : Y \to Z$, there is a unique $h : X + Y \to Z$ such that $f = h \circ i_X$ and $g = h \circ i_Y$. Graphically, it looks like this:

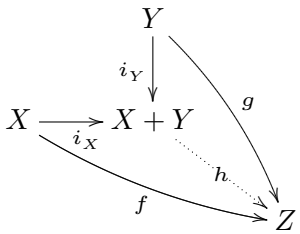


Figure 32: *Sum.*

Disjoint sum is also known as *disjoint union* or just *union*. Note that, depending on the category, the union does not *actually* need to be disjoint (see example 11).

Similarly to the situation with product, sum is not always an operation. It is a relation defined by the predicate, "an object Z satisfies the definition of sum for X and Y." In some categories, this amounts to an operation, e.g., in $\mathcal{S}et$, where the existence of a union is guaranteed by the union axiom. In others, it may not exist, or it may not be defined uniquely. Any other object satisfying the definition is also a union.

Example 11 Take two sets, X and Y, with no common elements. Their union, as sets, is the union in the category of sets as described in the definition above. Why is it a union in the categorical sense? If we have a set Z and two functions, $f : X \to Z$ and $g : Y \to Z$, then we can extend these two functions to $h : X + Y \to Z$ so that, restricted to X, it would give f, and restricted to Y, it would give g. Of course, a sum is defined by its universal property, so any set with the same number of elements as $X + Y$ can serve as the sum of X and Y. We would just need to provide inclusions i_X and i_Y and make sure that even if X and Y may have the same elements, the union is disjoint, and the elements of X are mapped via i_X to something different from the elements given by i_Y.

Example 12 Take a set A and its lattice of subsets, $P(A)$. The elements of this lattice are subsets of A, and the order is given by the "subset" relation, $X \subseteq Z$.

For two subsets of A, X and Y, their union, $X \cup Y = \{a \mid a \in X \vee a \in Y\}$, satisfies the universal property: if $X \subseteq Z$ and $Y \subseteq Z$, then we have an inclusion $(X \cup Y) \subseteq Z$.

Example 13 In any partial order P, the union of two elements, X and Y, is the same as the least upper bound of X and Y. The term "disjoint" is meaningless in this specific case.

Monoidal Properties of Sum

Due to its universality, similar to that of product, we can demonstrate that the sum of two objects does not depend on their order and that sum is associative. Both properties are valid up to an isomorphism: $X + Y \cong Y + X$ and $(X + Y) + Z \cong X + (Y + Z)$.

A disjoint sum can be viewed as a monoidal operation on the objects of our category. Of course, we do not actually have a monoid, because Sum is defined up to an isomorphism. Still, to have something that reminds us of a monoid, we need a neutral element. An initial object is such an element. This argument is similar (dual, actually) to finding a neutral element for a Cartesian product.

Before coming up with a construction that generalizes notions like terminal objects and products, we will have to look at more specific notions and examples. We will start with equalizers.

Equalizer

Before discussing equalizers and coequalizers, we need to be clear regarding what we are equalizing (or coequalizing).

Definition: Parallel Arrows

Two arrows, f and g, in a category are parallel if they have the same domain and the same codomain: $dom(f) = dom(g)$ and $cod(f) = cod(g)$. Note also that any two endomorphisms of the same object are parallel.

Definition: Equalizing Two Arrows

Given a category $\mathcal{C}$ and two parallel arrows, $f : X \to Y$ and $g : X \to Y$, an arrow $h : Z \to X$ is said to *equalize* f and g if $f \circ h = g \circ h$.

Out of all such equalizing arrows for f and g, any such arrow that has a universal property (see the definition below) is called an equalizer of f and g.

Definition: Equalizer

Given a category $\mathcal{C}$ and two parallel arrows, $f : X \to Y$ and $g : X \to Y$, the equalizer of f, g is an object $Eq(f, g)$ with an arrow $q : Eq(f, g) \to X$ equalizing (f, g) such that any other arrow $h : Z \to X$ equalizing (f, g) can be represented as $q \circ z$ for some unique $z : Z \to Eq(f, g)$. The following picture illustrates this:

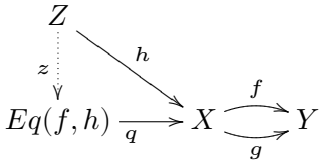


Figure 33: *Equalizer.*

Example 14. Equalizer in the Category of Sets This is the simplest case—given two functions, f and g, their equalizer is:

$$\{x \mid f(x) = g(x)\}$$

We can easily check whether or not this subset of X satisfies the definition.

Example 15. Fixpoint In the category of sets, when g is an identity, we will have $Eq(f, id_X) = \{x \mid f(x) = x\}$, and it is clear that it satisfies the definition of the set of fixpoints for the function f.

We have a choice of either viewing $Eq(f, id_X)$ as a set of all possible fixpoints of f or defining it as a "generalized fixpoint" (so that every other fixpoint, as a subobject of X, is included in it). Note that the existence of a set of fixpoints does not mean it is not empty.

Not every endomorphism has a fixpoint. If an endomorphism does not have one, the "generalized fixpoint" is just an empty set.

Example 16. Equalizer in a Partial Order In a partial order, every two parallel arrows $X \to Y$ are equal, so equalizing them is easy: the equalizer is always the domain X with its identity arrow.

Example 17. Equalizer in a Database Suppose we have a database with a table named `user` that has a foreign key named `boss`, pointing to the same table `user`, assuming that every user has a boss. It is probably a Soviet Russian database. Then consider following SQL query:

```
select * from users where id = boss;
```

It amounts to finding all fixpoints, that is, all the people who are their own bosses. The result is an equalizer, as mentioned above.

Note that an equalizer is also a monomorphism. Go ahead and prove it yourself; it is not difficult.

Example 18. Equalizer in Scala We will use the category of Scala sets, as discussed in Example 2. This implementation of equalizers is pretty much the same as shown in Example 14 but expressed in Scala:

```
def equalizer[X, Y](xs: Set[X], f: X => Y, g: X => Y): Set[X] =
  for {
    x <- xs if f(x) == g(x)
  } yield x
```

Coequalizer

Dual to the notion of equalizing and equalizer, we can introduce *coequalizing* and *coequalizer*. To do so, we can just revert the arrows (see below). Unfortunately, this duality does not extend to set-theoretical definitions. In $\mathcal{S}et$, a coequalizer is more complicated.

Definition: Coequalizer

Given a category $\mathcal{C}$ and two parallel arrows, $f : X \to Y$ and $g : X \to Y$, we say that arrow $h : Y \to Z$ *coequalizes* f and g if

$$h \circ f = h \circ g$$

A *coequalizer* is an arrow that is universal (rather, *couniversal*) among all arrows coequalizing f and g.

Generally speaking, building a coequalizer may take more computational efforts. E.g., if we're talking about sets, a coequalizer can be built as a *factor set* of Y, defined by an equivalence relation $f(x) \cong g(x)$. To check if $y_a \cong y_1$, one must find a chain $y_1 = f(x_1) \cong g(x_1) = f(x_2) \cong ... \cong g(x_n) = y_2$. This may take forever, so operations of this kind—together with coequalizers—are unpopular, and we won't focus on them either.

Pullback

Now we have two essential structures, product and equalizer, that can help build other universal structures. The first such new structure is called a *pullback*, which encapsulates common features of a product and an equalizer.

Below is a pullback diagram for (f, g).

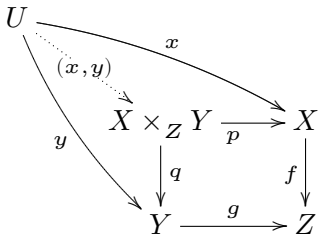


Figure 34: *Pullback.*

Definition: Pullback

Given a category $\mathcal{C}$, three objects, X, Y, and Z, and two arrows, $f : X \to Z$ and $g : Y \to Z$, a *pullback* of the pair (f, g) is an object denoted as $X \times_Z Y$ (read as "X cross Y over Z") together with two *projections* (p, q) and having the following two properties:

- The pair (x, y) equalizes (f, g), in the sense that the two arrows, q and p, satisfy $g \circ q = f \circ p$ (this is different from the definition of equalizing that we saw at the beginning of this chapter).
- Any other (U, x, y), such that $g \circ y = f \circ x$, can be split through $X \times_Z Y$ (see the diagram above).

The definition may look overcomplicated, but the following database example should help you readily understand it.

Example 19. Pullback in the Category of Sets Having two arrows, $f : X \to Z$ and $g : Y \to Z$, we can express their pullback as

$$\{(x, y) \mid x \in X \land y \in Y \land f(x) = g(y)\}$$

Example 20. Pullback in Scala Using the category of Scala sets from Examples 2 and 18, we can easily define a pullback. It looks similar to a combination of a product and an equalizer:

```
def pullback[X, Y, Z](
  xs: Set[X], ys: Set[Y], f: X -> Z, g: Y -> Z): Set[(X, Y)] =
  for {
    x <- xs
    y <- ys if f(x) == g(y)
  } yield (x, y)
```

Example 21. Pullback in Databases

```
select * from tableX, tableY where tableX.f = tableY.g;
```

This SQL join query conveys the essence of a pullback: take all the data from two tables such that certain values match.

```
select * from user, company where user.address = company.address;
```

This query essentially represents a pullback. It lists all people that have companies in their garages, together with the company information.

Example 22. Partial Order In a partial order, there is at most one arrow between two objects, so a pullback is the same as a product, that is, the greatest lower bound, if it exists, is also a pullback.

Properties of a Pullback

Given two arrows, $f : X \to Z$ and $g : Y \to Z$, one can express their pullback as an equalizer of two arrows from $X \times Y \to Z$. See the diagram:

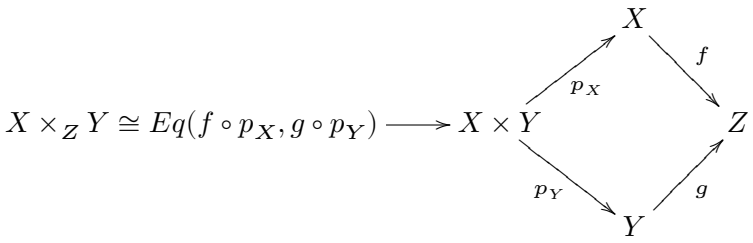


Figure 35: *Pullback is an equalizer.*

We can also express a product as a pullback. Just take $Z = 1$, then arrows f and g become irrelevant when forming the pullback:

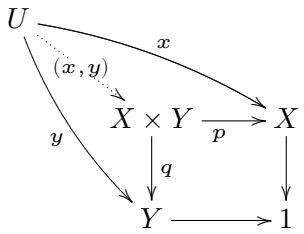


Figure 36: *Product is a pullback.*

Expressing an equalizer via a pullback is less obvious but still doable. Look at the following diagram:

What do we see here? Two arrows, $f : X \to Y$ and $g : X \to Y$, give us an arrow $(f, g) : X \to Y \times Y$. Another arrow, known as a *diagonal*, is from Y to $Y \times Y$, namely: $(id_Y, id_Y) : Y \to Y \times Y$. The pullback (eq_X, eq_Y) of these two arrows equalizes them: $(f, g) \circ (eq_X, eq_X) = (id_Y, id_Y) \circ eq_Y$, and hence $f \circ eq_X = id_Y \circ eq_Y = g \circ eq_X$.

It remains to be proven that the solution is universal, that is, closest to (X, Y). The reader may attempt it as an exercise.

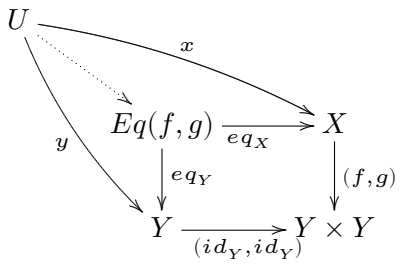


Figure 37: *Equalizer is a pullback.*

Pushout

Similar to pullbacks, we can define pushouts. To do this, we reverse all the arrows in the definition.

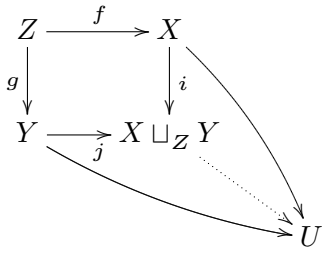


Figure 38: *Pushout.*

This construction can be expressed via sums and coequalizers. Additionally, since coequalizers are hard to calculate, pushouts are also hard to calculate, with the exception of partial orders, where a pushout is the same as a union.

Chapter 11. Relations Between Categories

So far, we have looked at individual categories as if they were isolated galaxies, interesting structures inside but no connections between them. In this chapter, we will investigate how categories relate to one another so that we can discover a "category of all categories," something that is impossible for sets, for example.

To build a category of categories, we need to introduce arrows between categories, their composition, and identity arrows for categories.

We start with two precursors of categories: monoids and graphs.

In $\mathcal{G}rph$, the category of all graphs, an arrow from one graph to another consists of mapping nodes of the first graph to the nodes of the second graph and properly mapping edges: if e is an edge from s to t in the first graph, $f(e)$ should be an edge from $f(s)$ to $f(t)$ in the second graph.

In $\mathcal{M}on$, the category of all monoids, arrows are monoid functions, that is, set functions that preserve the neutral element and the binary operation.

Since a category is both a graph and a generalization of a monoid, it would be natural to define an arrow from category $\mathcal{A}$ to category $\mathcal{B}$ as a couple of mappings, objects to objects and arrows to arrows, so that identities and compositions are preserved.

Functors

Definition: Functor

Given two categories, $\mathcal{A}$ and $\mathcal{B}$, a *functor* $F : \mathcal{A} \to \mathcal{B}$ consists of the following components:

- F_0, which maps objects of $\mathcal{A}$ to objects of $\mathcal{B}$.
- F_1, which maps arrows of $\mathcal{A}$ to arrows of $\mathcal{B}$.

Since these two components, F_0 and F_1, act on collections of data of two different types, for simplicity, we can safely omit the subscripts and further denote both with just one letter F. Also, the application of a functor, according to a tradition followed by certain programming languages, will be denoted not as $F(X)$ but as $F[X]$. You will see that, although unusual, this difference makes expressions more readable.

These two mappings should have the following two properties:

- $F[id_X] = id_{F[X]}$
- $F[f \circ g] = F[f] \circ F[g]$

Traditionally, the definition of a functor also requires that for an arrow $f : X \to Y$, we should have $F[f] : F[X] \to F[Y]$. This follows from our definition above: $F[f] = F[f \circ id_X] = F[f] \circ id_{F[X]}$, which means that $id_{F[X]}$ can be followed by $F[f]$, which is possible only if the domain of $F[f]$ is $F[X]$.

You may have heard the term "functor" in a variety of confusing contexts. Some programming books use the term to denote any arrow that acts on other arrows. This is incorrect. A mapping from one arrow to another is called an *operator* in mathematics (e.g., a derivative, $f \mapsto f'$). A mapping from an arrow to a scalar value is called a *functional* (e.g., an integral or map/reduce).

The simplest example of a functor is an identity functor. Map everything to itself, and you have a functor, specifically, an *endofunctor*, since its domain and codomain are the same category.

Before providing examples, we will demonstrate that we have a category, in fact a "category of all categories," called $\mathcal{C}at$. In $\mathcal{C}at$, categories play the role of objects in $\mathcal{C}at$, and functors play the role of arrows. The composition of two functors is easy to define: apply one, then apply another. Composition is associative simply because for any X, we have $(F \circ G \circ H)[X] = F[G[H[X]]]$.

Note that $\mathcal{C}at$ is so huge that it contains itself as an object. It may sound unusual to a mathematician, but we are not dealing with sets and do not have set-theoretic axioms, specifically, the comprehension axiom.

The comprehension axiom would allow us to filter objects to build another collection, and this could lead to a contradiction if we also applied the filter to the resulting collection. This operation is not generally required in a category, one of the reasons being that category theory is not equational.

Examples of Functors

Example 1. `List[+T]` This is the most popular functor in programming. We will not discuss its features but examine it from a categorical point of view. Consider a category for an unspecified programming language (as long as it is Scala): its types are objects of the category, and single-argument functions are the arrows. Given a type `T`, we can produce a type `List[T]`. So, we have a `List` defined for objects. What about arrows? Given $f : T \to U$, what would serve as $List[f] : List[T] \to List[U]$? We don't have much of a choice; it is provided by `List.map`. Namely, `List[T].map(f)` is the function from `List[T]` to `List[U]` that we were looking for.

To make sure that it is actually a functor, it remains to check that `List[T].map(identity[T])` is the same as `identity[List[T]]` and that `List[T].map(f andThen g)` is equal to `List[T].map(f) andThen List[U].map(g)`, where `f: T => U` and `g: U => V`.

Instead of a regular category $\mathcal{S}cala$, we can use another category (a partial order) in which only subtypings are allowed as arrows. In this category, an arrow $isSubtype : T \to U$ is written as `T <: U`. For such a category, the map function for functor `List` does not need to be introduced—it can be provided by the language. What we need is from `T <: U` it would follow that `List[T] <: List[U]`. This is provided by defining the signature of `List` like this: `List[+T]`. More on this later.

Example 2. Set[T] In Scala and Java, and likely in some other languages, `Set[T]` is a parameterized type representing "a set of values of type T." Of course, just mapping types is not enough. Mapping for functions also needs to be defined, and this presents a problem.

The obvious candidate is the traditional `Set[T].map(f: T=>U)`, which creates a new set out of the values of `f` for elements of the original set, by building an image of function `f`.

While this solution is theoretically correct, in practice, it may not be a good idea to populate a new set for each transformation. On the other hand, if the new set is virtual (lazy), for every call of the `set.contains(x)` method, we would need to scan through the entire original set and compare the result of the function application with the value provided.

Example 3. A Functor from Category $\mathbb{1}$ $\mathbb{1}$ is a category consisting of one object and its identity arrow. What does a functor from $\mathbb{1}$ to a category $\mathcal{C}$ look like? To specify object mapping, we only need to select an object in $\mathcal{C}$. For any object x there is exactly one functor $x : \mathbb{1} \to \mathcal{C}$.

Example 4. A Constant Functor Given two categories, $\mathcal{A}$ and $\mathcal{B}$, any object x in $\mathcal{B}$ can be thought of as a functor from $\mathcal{A}$. What does this functor look like? We map every object of $\mathcal{A}$ to x and every arrow to id_x. All the properties of a functor are satisfied. Check it for yourself.

Example 5. A Functor from $\mathbb{2}$ to the Category of Sets You may remember the category $\mathbb{2}$ consisting of two objects, 0 and 1, an arrow, let's call it 01, from 0 to 1, and a couple of identity arrows. $\mathcal{S}et$ is the category of sets. To define a functor F from $\mathbb{2}$ to $\mathcal{S}et$, we will need three items: a set $F[0]$, a set $F[1]$, and a function $F[01]$ from $F[0]$ to $F[1]$.

$$F[0] \xrightarrow{F[01]} F[1]$$

Figure 39: *Functor from $\mathbb{2}$ to $\mathcal{S}et$.*

To make our notation les awkward, we rename $F[0]$ to F_0, $F[1]$ to F_1, and $F[01]$ to F_{01}. Now we can see that every functor from $\mathbb{2}$ to $\mathcal{S}et$ is just an arrow $F_{01} : F_0 \to F_1$ in sets, and every

arrow $f : A \to B$ can be thought of as a functor from 2 to $\mathcal{S}et$ where $F_0 = A$, $F_1 = B$, and $F_{01} = f$.

Indeed, the fact that we are dealing with the category $\mathcal{S}et$ is irrelevant. The same argument would work for any category $\mathcal{C}$: functors $2 \to \mathcal{C}$ are arrows in $\mathcal{C}$.

Example 6. A Functor from the Category of Sets to 2 What functors from $\mathcal{S}et$ to 2 can we find? Two obvious functors are constants. Map all objects of $\mathcal{S}et$ to object 0 of 2 and all arrows to id_0. Similarly, map all objects of $\mathcal{S}et$ to object 1 of 2 and all arrows to id_1.

In addition to these two obvious functors, there may be others that cover both 0 and 1. Note that the empty set $\emptyset$ is a subset of every set. That is, there is an arrow from $\emptyset$ to every set X. In the category 2, on the other hand, there is no arrow from 1 to 0. Hence, if a functor F maps $\emptyset$ to 1, every other set should also map to 1. Now we have the constant functor we discussed above.

Similarly to singleton sets, which are terminal objects in $\mathcal{S}et$ (with an arrow from every set X to every singleton), if a singleton maps to 0, every set maps to 0.

A non-constant functor would map some sets to 0 and some to 1. As follows from the observations above, $\emptyset$ should be mapped to 0, otherwise the functor is constant 1. A singleton should be mapped to 1, otherwise the functor is constant 0. What are our options for mapping other sets? Every nonempty set X has an element and there is an arrow from a singleton to X, so F should map every nonempty X to 1. This determines our functor uniquely.

Exactly three functors exist from $\mathcal{S}et$ to 2:

- constant 0
- constant 1
- $\emptyset$ maps to 0, all others map to 1

Example 7. Product with an Object Given a category $\mathcal{C}$ that has products and an object A in $\mathcal{C}$, we can produce a functor that consists of multiplying by an object A, that is, $A \times \_ : \mathcal{C} \to \mathcal{C}$. The functor maps each object X to $A \times X$, and for an arrow $f : X \to Y$, it provides $A \times f : A \times X \to A \times Y$. You have probably already figured out how it works.

Example 8. Set Exponentiation In the category of sets, given a set A, we can always build, for any set X, a set X^A consisting of functions from A to X. This is a functor, denoted as $\_^A : \mathcal{S}et \to \mathcal{S}et$. What makes it a functor? We need to define, for any $f : X \to Y$, a function $f^A : X^A \to Y^A$, which can be defined element-wise. Given $x_a \in X^A$, that is $x_a : A \to X$, we produce $y_a : A \to Y$ by defining it as $y_a = f \circ x_a$.

Example 9. Monoids A monoid can be represented as a category with one object. So, if we have two monoids, a monoidal function from one to another is the same as a functor: it preserves multiplication (acting as composition) and the neutral element (acting as an identity arrow).

Example 10. Functors between Partial Orders A partial order is also a category and a functor between two such categories is a partial order function that preserves order (see Chapter 3).

Example 11. From Integers to Reals The regular inclusion of $\mathbb{Z}$ (the partial order of integer numbers) into $\mathbb{R}$ (the partial order of real numbers) preserves the order, so this inclusion is a functor, from a categorical point of view.

Example 12. From Reals to Integers If we map real numbers to integers, by taking $X \mapsto [x]$, we also have a monotone function, which is a functor $\mathbb{R} \to \mathbb{Z}$.

Example 13. Inclusion of Sets into Partial Functions and Binary Relations If we take two categories $\mathcal{S}et$ and $\mathcal{S}et_{Part}$, and map each set to itself and each function to itself, viewed as a partial function, we have an inclusion. Since it preserves composition and identities, we also have a functor.

Similarly, we can include $\mathcal{S}et$ to $\mathcal{R}el$, by mapping each set to itself and each function to its graph (which is a binary relation): $f \mapsto \{x, f(x)) \mid x \in X\}$.

Building New Categories

Now that we know that categories form a category, we can try to figure out how to build unions, products, pullbacks, and equalizers in $\mathcal{C}at$ and whether the category has initial and terminal objects. Let's walk through these objects.

Initial Category

This is a category that has a unique functor to any (other) category. Such a category cannot have objects. If it did, we could apply a constant functor to it, for each object in any target category, and have more than one such functor, generally speaking. Thus, the only choice is the empty category, $\mathbb{0}$. Feel free to define a functor from $\mathbb{0}$ to any category $\mathcal{C}$.

Terminal Category

For a terminal category, each category $\mathcal{C}$ has a unique functor ending in it. If we take category $\mathbb{1}$, for each category $\mathcal{C}$, there can be exactly one functor $\mathcal{C} \to \mathbb{1}$. Now we see that there is a terminal object in $\mathcal{C}at$.

Product of Two Categories

This structure can be built similar to $\mathcal{S}et$. Given two categories, $\mathcal{C}$ and $\mathcal{D}$, take as objects of $\mathcal{C} \times \mathcal{D}$ all pairs of objects (x, y) where x is an object of $\mathcal{C}$ and y is an object of $\mathcal{D}$. The fact that math allows us to form such pairs is beyond the scope of this text, and this can be done internally if we are within a certain domain. For arrows in this new category, take all pairs of arrows (f, g), where f is an arrow in $\mathcal{C}$ and g is an arrow in $\mathcal{D}$. Composition is defined component-wise, so that $(f_1, g_1) \circ (f_2, g_2) = (f_1 \circ f_2, g_1 \circ g_2)$.

To establish that we have a category, provide identities $id_{(X,Y)} = (id_X, id_Y)$ and verify that they are neutral regarding composition and that the composition is associative.

Does this category have the universal property in $\mathcal{C}at$? It can be proven component-wise that it does.

Sum of Two Categories

Given two categories, $\mathcal{C}$ and $\mathcal{D}$, and assuming that we can build a category consisting of objects of $\mathcal{C}$ and objects of $\mathcal{D}$ and arrows from these two categories, we have a new category, $\mathcal{C} + \mathcal{D}$. Examples of this have already been provided: the sum of n instances of $\mathbb{1}$, that is, $\mathbb{1} + \mathbb{1} + ... \mathbb{1}$, is a discrete category consisting of n objects and only identity arrows.

Equalizer? Pullback? Pushout?

Generally speaking, these constructions are not available in $\mathcal{C}at$, for many reasons. One reason is that equality for objects is not defined in categories, only isomorphisms, so we would have to define everything up to an isomorphism, which would require higher-order categories. Higher-order categories are not covered this book.

Reversing the Arrows

Remember, arrows in a category are not necessarily functions that take an argument and return a value; they are just formal abstractions. So, given a category $\mathcal{C}$, nothing can stop us from producing another category from it, by reversing the direction of all the arrows.

Definition: Opposite Category

Given a category $\mathcal{C}$, its *opposite*, or *dual* $\mathcal{C}^{op}$, is a category with the same objects and the same arrows but with the direction of the arrows reversed. If in $\mathcal{C}$, $f : X \to Y$, then in $\mathcal{C}^{op}$ we have $f^{op} : Y \to X$.

Note that any knowledge about arrows has no value here—these arrows are just symbols. Composition in $\mathcal{C}^{op}$ is defined in the opposite direction as well, so

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

The fact that it is a category can easily be proven as an exercise.

For some categories, the opposite is isomorphic to the original category (e.g., $\mathbb{1}$, $\mathbb{2}$, $\mathbb{3}$…). There is a bigger category with this property, $\mathcal{R}el$. It is a category of sets and their binary relations, and it is symmetrical relative to the operation of taking *op*: $\mathcal{R}el^{op}$ is isomorphic to $\mathcal{R}el$. For other categories, finding the opposite may not be trivial at all. For instance, $\mathcal{S}et^{op}$ is the category of *complete atomic Boolean algebras.*

Omitting exact definitions, we can intuitively describe it like this: given a set, we have its characteristic function, a predicate that is true only for members of that set. Now, given an arrow $f : X \to Y$ in the category of sets, and a predicate p_Y defined in the set Y, we can produce a predicate p_X on X by defining $p_X(x) = p_Y(f(x))$. We have just mapped a predicate over Y to a predicate over X. This way, for each map f between sets, we have a mapping between predicates: $Pred[f] : Pred[Y] \to Pred[X]$. We can view collections of such predicates, for each given set, as objects in a category. Given certain assumptions, we may also think of such predicates as being the same as the underlying sets.

In programming languages, this operation is equivalent to defining a set via its "contains" predicate. Of course, this is not enough. We also need to make sure that every such predicate can be represented as a disjunction of atomic predicates, with one "characteristic" predicate for each element of the original set.

Contravariant Functor

The functors we've been discussing so far are frequently called *covariant* functors, due to their actions on arrows that map domain to domain and codomain to codomain. Another kind of functor, one that maps the domain of an arrow to a codomain, and its codomain to domain, is called *contravariant.*

Strictly speaking, we do not need a special term because a contravariant functor can be always thought of as a (covariant) functor $\mathcal{C}^{op} \to \mathcal{D}$ (or $\mathcal{C} \to \mathcal{D}^{op}$).

Variance plays an important role in computer science, and we must spend some time discussing it.

Example 1. `Map[_,T]` If, in Scala, we fix the second argument of the parameterized type `Map`, for an arrow `f: X => Y`, we will be able to produce an arrow `Map[Y,T] => Map[X,T]`. This mapping preserves identities and composition, therefore we have a contravariant functor. In Scala, contravariance is denoted using a minus symbol: `Map[-X,+Y]` is the signature of this type.

Variance in Programming Languages

In languages allowing subtyping (e.g., Scala), a functor—represented as a parameterized class—may have a variance marker. This marker does not mean that we are dealing with a covariant or contravariant functor. This variance marker is about inclusions ("subtyping") of types into other types.

E.g., if `A <: B` and `X <: Y` (a notation for the compiler's ability to subtype one type into another), we have `Map[B,X] <: Map[A,X]` and `Map[A,X] <: Map[A,Y]`. This is a category in which types are objects, and the relations of subtyping are arrows. This category is a partial order, so deciding whether we need covariance or contravariance is easier than when we deal with generic arrows.

Chapter 12. Relations Between Functors

Natural Transformations

We gradually climbed up the hierarchy of relations. Objects and arrows constitute a category. Then, we studied relations between categories, which are represented by functors in the simplest case. Now, it is time to study relations between functors. This investigation will also provide a fresher look at constructs within categories.

For the sake of simplicity, we will start with $\mathcal{A} = \mathbb{1}$. Functors from $\mathbb{1}$—"points"—are just objects in $\mathcal{B}$. So, given two such functors, x and y, we also have arrows from x to y. We are practically dealing with the same category $\mathcal{B}$, but we treat its objects as functors and arrows between objects as arrows from one functor to another.

Now, let's expand on this idea a little bit and take $\mathcal{A} = \mathbb{2}$. In this case, each functor x from $\mathbb{2}$ to $\mathcal{B}$ consists of two objects, x_0 and x_1, and an arrow $x_{01} : x_0 \to x_1$. If we have two such functors, x and y, we have $x_{01} : x_0 \to x_1$ and $y_{01} : y_0 \to y_1$.

Similar to the case of $\mathcal{A} = \mathbb{1}$, we want to define arrows from x to y. Such an arrow will consist of two components, $f_0 : x_0 \to y_0$ and $f_1 : x_1 \to y_1$. The following diagram needs to be a commutative square (that is, $y_{01} \circ f_0 = f_1 \circ x_{01}$):

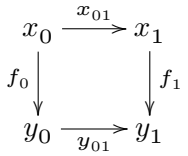


Figure 40: *Arrow between two arrows.*

There are functors and arrows between them, and we may now expect them to form a catogory: the collection of functors from $\mathbb{2}$ to $\mathcal{B}$ and pairs of arrows, (f_0, f_1). But is it a category?

To have a category, identity arrows need to be defined, and for each appropriate pair of arrows, we need their composition to be defined, as well, and the composition has to be associative.

An identity for x consists of (id_{x_0}, id_{x_1}), and we define composition per component: $(f_0,\ f_1) \circ (g_0,\ g_1) \equiv (f_0 \circ g_0,\ f_1 \circ g_1)$.

The two examples above will help us define arrows between functors $\mathcal{A} \to \mathcal{B}$ in a general case.

Definition: Natural Transformation

Given two categories, $\mathcal{A}$ and $\mathcal{B}$, and two functors, $F, G : \mathcal{A} \to \mathcal{B}$, a natural transformation $t : F \to G$ consists of arrows $t_a : F[a] \to G[a]$ such that for any $h : x \to y$ in $\mathcal{A}$, we have $t_y \circ F[h] = G[h] \circ t_x$.

$$\begin{array}{ccc} F[x] & \xrightarrow{F[h]} & F[y] \\ {\scriptstyle t_x}\downarrow & & \downarrow{\scriptstyle t_y} \\ G[x] & \xrightarrow[G[h]]{} & G[y] \end{array}$$

Figure 41: *Natural transformation.*

A nice way to draw a natural transformation is a diagram. Given categories $\mathcal{A}$ and $\mathcal{B}$ and functors, $F, G : \mathcal{A} \to \mathcal{B}$, a natural transformation $t : F \to G$ is drawn like this:

$$A \overset{F}{\underset{G}{\Downarrow t}} B$$

Figure 42: *Natural transformation notation.*

We will need more examples to illustrate what a natural transformation may look like in different circumstances.

Example 1. Flatten a `List`

```
flatten: List[List[T]] => List[T]
```

This action is pretty simple and familiar: take a list of lists and flatten it, concatenating its elements to produce one single list of values of type `T`. We need to check that this action is a natural transformation.

First, two functors are involved in a natural transformation. One functor we know from Chapter 11: it is `List[_]`. But is `List[List[_]]` a functor? It must be, since it is produced by composing `List[_]` with itself. Being a composition of functors, it is also a functor.

So, `flatten` is defined on each type of form `List[List[T]]`. Is this a natural transformation? We need to check whether, given an arrow `f: T => U`, it makes the square commute:

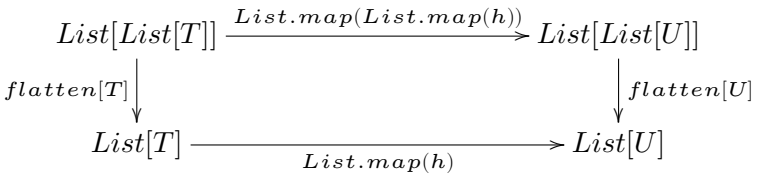


Figure 43: *'List.flatten' is a natural transformation.*

Why is the square commutative? One path in the square consists of flattening, then mapping; the other one consists of mapping, then flattening. You can check this out using the definitions of `flatten` and `map`. As a result, we see that `flatten` is a natural transformation.

Example 2. Singleton `List`

```
_::Nil: T => List[T]
```

This operation consists of making a singleton list out of an instance of `T`. The functor on the left is the identity functor and the functor on the right is `List[_]`. Is it a natural transformation?

For this to be a natural transformation, given a function `f: T => U`, we need to have

```
f(t)::Nil = (t::Nil).map(f)
```

which should be obvious.

Example 3. Arrows are Natural Transformations Recapping the beginning of this chapter, we know that a functor from $\mathbb{1}$ to category $\mathcal{C}$ is just an object of $\mathcal{C}$—any object. So, given two such functors, x and y, any arrow $f : x \to y$ is a natural transformation from x to y .

Example 4. Cone In the previous chapter, we saw a constant functor. Now, take a constant functor $x : \mathcal{A} \to \mathcal{B}$ (mapping every object of $\mathcal{A}$ to a given object $x \in \mathcal{B}$ and all arrows of $\mathcal{A}$ to id_x).

What would a transformation from such a constant functor to an arbitrary functor $F : \mathcal{A} \to \mathcal{B}$ look like?

For each object a in $\mathcal{A}$, we will need an arrow $f_a : x \to F[a]$ that is compatible with all $F[a] \to F[b]$, as shown in the following diagram:

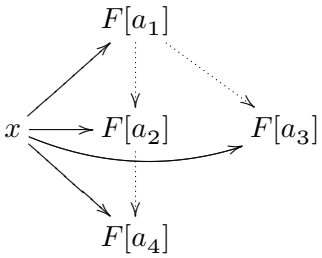


Figure 44: *Cone.*

This diagram is called a *cone*. The object x in such a diagram is called the cone's *vertex*. A similar diagram, with a natural transformation from a functor F to a constant functor, is called a *cocone*.

Knowing that all functors from $\mathcal{B}$ to $\mathcal{A}$ form a category, we can introduce a notation for this category: $\mathcal{A}^{\mathcal{B}}$. This notation makes a lot of sense. If we have $\mathcal{A}^{\mathcal{B}}$ and $\mathcal{A}^{\mathcal{C}}$, their product, $\mathcal{A}^{\mathcal{B}} \times \mathcal{A}^{\mathcal{C}}$, is equivalent to $\mathcal{A}^{\mathcal{B}+\mathcal{C}}$. For example, $\mathcal{A}^{\mathbb{1}+\mathbb{1}}$ is equivalent to $\mathcal{A} \times \mathcal{A}$. This notation is similar to the case with numbers, for which $a^{b+c} = a^b \times a^c$ and the reasons are more or less the same.

Adjoint Functors

We have covered relations between functors with the same domain and codomain, that is, pointing in the same direction.

In contrast, consider a pair of parallel functors pointing in opposite directions. A spooky entanglement had been observed between such functors. This section is dedicated to that case.

Example 5. Galois Connection We start with a simple case: two partial orders, for example, $\mathbb{Z}$ and $\mathbb{R}$. We have an inclusion functor, $Include : \mathbb{Z} \to \mathbb{R}$, and another functor, "integral part," $Int : \mathbb{R} \to \mathbb{Z}$. Can you see that these two are related? For each $x \in \mathbb{R}$ and $n \in \mathbb{Z}$, $x < n$ is

equivalent to $[x] < n$. Therefore, there is a one-to-one correspondence between arrows $x \to n$ in $\mathbb{R}$ and $[x] \to n$ in $\mathbb{Z}$. Such a correspondence between two monotone arrows, and between two partial orders, is called a Galois connection.

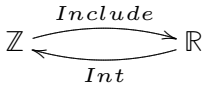


Figure 45: *Galois connection between $\mathbb{Z}$ and $\mathbb{R}$.*

Example 6. Partial Functions Now, let's look at a richer example. If we take two familiar categories, regular sets with functions, $\mathcal{S}et$, and sets with partial functions, $\mathcal{S}et_{Part}$, we can try to build a similar related pair of functors.

The category $\mathcal{S}et_{Part}$ consists of sets as objects and partial functions as arrows:

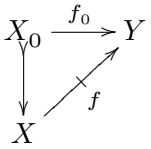


Figure 46: *Partial function.*

There is an obvious inclusion functor from $\mathcal{S}et$ to $\mathcal{S}et_{Part}$, where each function is perceived as a partial function (with $X_0 = X$).

For the purpose of this example, we want to build a functor in the opposite direction, $\mathcal{S}et_{Part} \to \mathcal{S}et$. How can this be done?

Here is one solution. Fix a singleton in $\mathcal{S}et$ and call it $1 = \{0\}$. For every set X in $\mathcal{S}et_{Part}$, choose $X + 1$ in $\mathcal{S}et$, that is, the same set with one appended element. This is a disjoint union, so whether X contains 1 as a subset is irrelevant—we are adding one more point.

This defines the object mapping for the functor we are building, $Option[X] = X + 1$.

For an arrow f in $\mathcal{S}et_{Part}$, that is, for a partial function $X \nrightarrow Y$, represented by $X_0 \to Y$ where $X_0 \subset X$, define a function $Option[f]$ in $\mathcal{S}et$ like this:

$Option[f](0) = 0$
$Option[f](x) = if(x \in X_0)\ then\ f(x)\ else\ 0$

Be sure to check that we have a functor, that is, that identity and composition are preserved.

With these two functors, $Include$ and $Option$, the picture is similar to the one in Example 5: partial functions $f : Include[X] \nrightarrow Y$ are in a one-to-one correspondence with functions $g : X \to Option[Y]$.

How does it work?

An arrow $f : Include[X] \nrightarrow Y$ is uniquely defined by a function $f_0 : X_0 \to Y$ for some $X_0 \subset X$, and this function is in a one-to-one correspondence with a function $X \to Y + 1$.

Similarly, a function $g : X \to Y + 1$ is uniquely defined by $g_0 : X_0 \to Y$ for some $X_0 \subset X$.

We have a Galois connection again:

$$Set \underset{Option}{\overset{Include}{\rightleftarrows}} Set_{Part}$$

Figure 47: *Galois connection between $\mathcal{S}et_{Part}$ and $\mathcal{S}et$.*

We can actually model this in Scala, since Scala has partial functions. What happens to them in our construction? For a partial function `f: PartialFunction[X, Y]`, `f.lift` is a regular function `X => Option[Y]`.

Given a function `g: X => Option[Y]`, we can produce a partial function:

```
val pf: PartialFunction[X, Y] = Function.unlift(f)
```

This Scala implementation shows that the base category does not have too be $\mathcal{S}et$. How this observation can be generalized is hard to determine at our current level of expertise.

Example 7. Binary Relations In chapter 7, we discussed the category $\mathcal{R}el$, and in chapter 11, we saw a functor that embeds $\mathcal{S}et$ into $\mathcal{R}el$. Here, we will build a functor in the opposite direction. First, we need to introduce one more Z-notation: $\leftrightarrow$ for binary relation, e.g. $R : X \leftrightarrow Y$. Another notation: $Pow[X]$ is the set of all subsets of a set X.

Our functor will be a powerset functor: it maps each object X in $\mathcal{R}el$, that is, each set X to $Pow[X]$ as an object in $\mathcal{S}et$. As to the mapping of functions, given a relation $R : X \leftrightarrow Y$, we can build a function $Pow[f] : Pow[X] \to Pow[Y]$ by defining, for each $X_1 \subset X$, a subset of Y, $Y_1 = Pow[f](X_1) = \{y \mid \exists x \in X_1 : xRy\}$.

You can check that this mapping takes an identity relation on X (the graph of id_X) to $id_{Pow[X]}$ and that it preserves composition. So this is a functor. How is it related to the embedding of $\mathcal{S}et$ into $\mathcal{R}el$?

Given two sets, X and Y, and a relation $R : X \leftrightarrow Y$, we can build a function $p_R : X \to P[Y]$ by defining $p_R(x) = \{y \mid xRy\}$. This is a well-defined function. Having such a function, we can restore the relation: $R = \{(x, y) \mid y \in p_R(x)\}$. Thus, the same kind of connection exists between the two functors.

By the way, the latter construction, the lifting of a relation to a function, can be illustrated in SQL:

```
select y from MyTable where x=?;
```

In this example, given a binary relation `MyTable`, we can "curry" it by returning a set of values for each given `x`.

Now, we are ready for a definition.

Definition: Adjoint Functors

Given two categories, $\mathcal{A}$ and $\mathcal{B}$, and two functors, $F : \mathcal{A} \to \mathcal{B}$ and $G : \mathcal{B} \to \mathcal{A}$, these two functors are called *adjoint* (F is a *left adjoint* and G is a *right adjoint*) if there is a one-to-one correspondence between arrows $F[X] \to Y$ and $X \to G[Y]$. This relation is denoted as $F \dashv G$.

You should be able to see that the three examples above are examples of adjoint functors. Moreover, every Galois connection is actually just an adjunction.

Returning to the examples above, we can observe now that in Example 5, with Z and R, we have $Include \dashv Int$. In Example 6, with $\mathcal{S}et$ and $\mathcal{S}et_{Part}$, we have $Include \dashv Option$, and in Example 7, Set and Rel, we have $Include \dashv P$.

More examples, using the functors that you are already familiar with, will be helpful.

Example 8. Cartesian Product Take a category $\mathcal{A}$ and a *diagonal functor*, $\Delta : \mathcal{A} \to \mathcal{A} \times \mathcal{A}$, which takes an X to a pair (X, X). We will show that this functor is part of an adjoint pair: a left adjoint, specifically. What would this mean in practice?

Take two objects, Z in $\mathcal{A}$ and (X, Y) in $\mathcal{A} \times \mathcal{A}$.

An arrow from (Z, Z) to (X, Y) in $\mathcal{A} \times \mathcal{A}$ consists of two arrows in $\mathcal{A}$: $f : Z \to X$ and $g : Z \to Y$.

To have a right adjoint $\Pi : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$, we will need to define $\Pi(X, Y)$ and have a one-to-one correspondence between pairs $(f : Z \to X, g : Z \to Y)$ and arrows $Z \to \Pi(X, Y)$. The solution is the familiar Cartesian product functor, $\Pi : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$: having an arrow $Z \to X \times Y$ is the same as having a pair of arrows, $(Z \to X,\ Z \to Y)$.

As a result, we have $\Delta \dashv \Pi$. There's no guarantee, though, that such a functor—Π, that is, a product—exists for a category $\mathcal{A}$.

Alternative Definition of Adjoint Functors

Another way to declare that two functors, $F: \mathcal{A} \to \mathcal{B}$ and $G: \mathcal{B} \to \mathcal{A}$, are adjoint, is to use a couple of natural transformations, unit $\eta : Id_{\mathcal{A}} \to GF$ and counit $\epsilon : FG \to Id_{\mathcal{B}}$. These two transformations must have the following properties (see Figure 9):

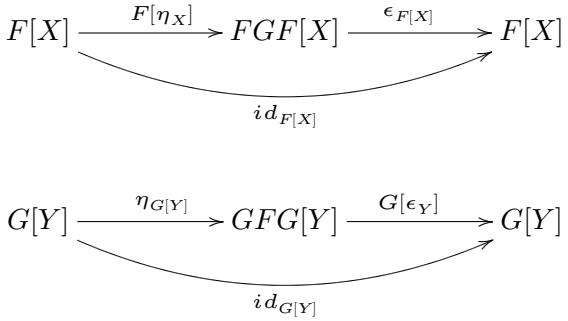


Figure 48: *Properties of Unit and Counit.*

Why are the two definitions equivalent? Starting with an adjoint pair $F \dashv G$, we take $id_{F[X]} : F[X] \to F[X]$ and reflect it, using our adjunction, to $\eta_X : X \to GF[X]$. Similarly, for $id_{G[Y]} : G[Y] \to G[Y]$, we use the adjunction and reflect it to $\epsilon_Y : FG[Y] \to Y$. The properties of adjunction follow.

On the other hand, suppose we have two transformations, a unit η and a counit ϵ. In this case, an arrow $F[X] \to Y$ gives us $GF[X] \to G[Y]$. When we compose it with the unit, we have $X \to GF[X] \to G[Y]$, which is the arrow from X to $G[Y]$, and which is also the other part of the adjunction we are looking for. Similarly, we can go in the opposite direction and demonstrate that these two give us an adjunction.

Can a functor have both left and right adjoints? It happens, but this situation is beyond the scope of this book. Can a functor have more than one left or right adjoint? Yes, but these adjoints are all isomorphic. If F has a right adjoint G_1 and another right adjoint G_2, there is an isomorphism between the two. So, adjunctions are defined up to an isomorphism, like many things in category theory.

Now, we have an alternative definition of adjunction, with unit and counit. What does unit and counit look like for the adjoint pairs that we've already seen?

Example 9. Unit and Counit for the Adjunction Ccaused by Partial Functions As discussed above, to build a unit $\eta : X \to X + 1$ in $\mathcal{S}et$ we need to take id_X as an arrow in $\mathcal{S}et_{Part}$ and reflect it

back to $\mathcal{S}et$. This gives an inclusion of X into $X+1$, and this is our unit η. Regarding counit $\epsilon : (X+1) \nrightarrow X$ in $\mathcal{S}et_{Part}$, it is a partial function on $X+1$ that consists of an identity on X.

Example 10. Unit and Counit for $\mathcal{S}et$ and $\mathcal{R}el$ Adjunction Again, given a set X, start with a binary relation $x == x$ (identity function in $\mathcal{R}el$), and reflect it back to $\mathcal{S}et$. We will have a function that maps $x \mapsto \{x\}$, that is, the singleton function $X \to Pow[X]$. This is our unit η. Again, counit ϵ is just an identity.

Example 11. Unit and Counit for Cartesian Product Adjunction In Example 8, we discussed the adjunction for a Cartesian product. Now, if we have a category $\mathcal{A}$, and an object X in $\mathcal{A}$, this object maps to (X, X) in $\mathcal{A} \times \mathcal{A}$. By reflecting the identity of (X, X) back to $\mathcal{A}$, we get $X \to X \times X$, and this function is obviously the diagonal, $(id_X, id_X) : X \to X \times X$. Now build the counit for this adjunction. Given a pair (X, Y) in $\mathcal{A} \times \mathcal{A}$, the functor Π maps it to the product $X \times Y$ in $\mathcal{A}$. So, when we map $X \times Y$, via the Δ functor, back to $\mathcal{A} \times \mathcal{A}$, we have $\Delta(X \times Y) = (X \times Y, X \times Y)$. The counit function $(X \times Y, X \times Y) \to (X, Y)$ consists of a pair of projections, (p_X, p_Y), where $p_X : X \times Y \to X$ and $p_Y : X \times Y \to Y$.

Limits

In Example 4, we discussed a natural transformation from constant functors (objects of a category $\mathcal{B}$) to arbitrary functors $F : \mathcal{A} \to \mathcal{B}$. We can look at this situation as follows.

The functor $Const : \mathcal{B} \to \mathcal{B}^{\mathcal{A}}$ may have an adjoint functor $Lim : \mathcal{B}^{\mathcal{A}} \to \mathcal{B}$, such that $Const \dashv Lim$. This functor, Lim, takes an $\mathcal{A}$-diagram in $\mathcal{B}$ and produces a limit for this diagram in $\mathcal{B}$.

Of course, officially, the definition of a limit involves not only an object but also a cone from that object to each component of the diagram $\mathcal{A} \to \mathcal{B}$. For simplicity, when talking about a limit, we will mean the limit object, that is, the *vertex* of the cone.

Let's get into more details. First, objects of the category $\mathcal{B}^{\mathcal{A}}$ are functors from $\mathcal{A}$ to $\mathcal{B}$. For a given object x in category $\mathcal{B}$, $Const(x)$ is an object of $\mathcal{B}^{\mathcal{A}}$, that is, a functor mapping every a_i in $\mathcal{A}$ to x, and every arrow to id_x.

A natural transformation from $Const(x)$ to an object (functor) F in $\mathcal{B}^{\mathcal{A}}$ consists of arrows $x \to F[a_i]$, commuting with arrows $F[a_i] \to F[a_j]$. Since $Const(x)[a_i]$ is always x, we actually have a cone:

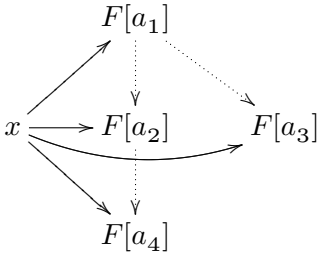


Figure 49: *Cone.*

If we want a right adjoint to $Const$ to exist, we need a one-to-one correspondence between cones $x \to F$ and regular arrows in category $\mathcal{B}$, $x \to Lim[F]$.

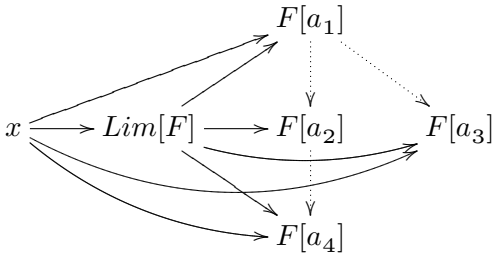


Figure 50: *Limit.*

In this figure, the arrow from x to $Lim[F]$ is the unit of the adjunction. As to the counit, it must be an arrow in $\mathcal{B}^{\mathcal{A}}$, that is, an arrow in $\mathcal{B}$ for each object a_i of $\mathcal{A}$ from x to $F[a_i]$. These two, $x \to Lim[F]$ and the collection of arrows, are in one-to-one correspondence due to the definition of a limit.

If the limit exists for each functor from $\mathcal{B}$ to $\mathcal{A}$, it is a right adjoint to the $Const$ functor.

The next three examples will show that many, now familiar categorical constructs are limits.

Example 12. Product as a Limit You might have noticed that if category $\mathcal{A}$ is $\mathbb{1}+\mathbb{1}$, the diagram is exactly the same as the diagram for a product. That's because $\mathcal{B}^{\mathbb{1}+\mathbb{1}}$ is $\mathcal{B} \times \mathcal{B}$.

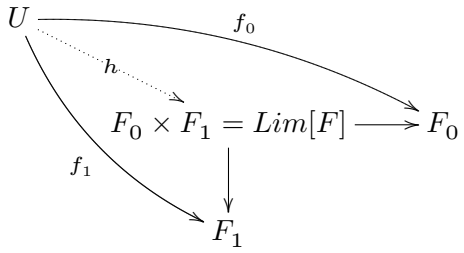


Figure 51: *Product as a limit.*

Example 13. Equalizer as a Limit To represent an equalizer of two arrows in category $\mathcal{C}$ as a limit, we need to find an appropriate domain category. The category of "parallel pairs" is a good choice.

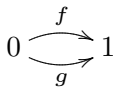


Figure 52: *Category of parallel pairs.*

A functor F from the category of parallel pairs to a category $\mathcal{C}$ consists of two objects, F_0 and F_1, and two parallel arrows, $F_f, F_g : F_0 \to F_1$. A limit of such a functor is an equalizer of the parallel pair (F_0, F_1):

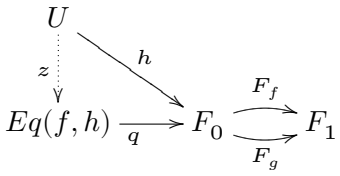


Figure 53: *Equalizer as a limit.*

Example 14. Terminal Object as a Limit A terminal object can be represented as a limit of an empty diagram, that is, a functor $\mathbb{0} \to \mathcal{C}$. What does this diagram consist of? It consists of nothing, so its limit must be an object to which there is a unique arrow from every other object.

$$U \dashrightarrow 1 = Lim[T]$$

Figure 54: *Terminal a limit.*

Chapter 13. Cartesian Closed Categories

Basic Ideas

In the category of sets, $\mathcal{S}et$, the functor of multiplying a set by a given set A has a right adjoint. By multiplying, I mean the following object mapping: $X \mapsto X \times A$. The adjunction is provided by the one-to-one correspondence between arrows $X \times A \to Y$ and arrows $X \to Y^A$. The operation is essentially *currying-uncurrying*, as described in Chapter 1, and it is popular in practical computing. Scala even has names for these operations: `curried` and `uncurried`.

```
val f: (Int, Int) => Int = _/_
val g = f.curried
val h = Function.uncurried(g)
g(10)(5) == 2
h(10, 2) == f(10, 2)
```

While Scala and $\mathcal{S}et$ have this kind of adjunction, it is not obvious how the idea can be generalized. In this chapter, we will investigate a special kind of category, in which the curry/uncurry adjunction is feasible.

Definition: Cartesian Closed Category

A category $\mathcal{C}$ is called a *Cartesian Closed Category* (CCC) if it has:

- Finite products, that is, for any finite (including empty) collection of objects, their Cartesian product exists.
- For each functor $\_ \times A : \mathcal{C} \to \mathcal{C}$, a right adjoint, *exponential* $\_^A$: $\_ \times A \dashv \_^A$.

We could equivalently replace the requirement of finite products with the requirement of a terminal object and products for any two objects in the category. The reason is that a finite product of arity $n > 2$ can be represented as the composition of a binary product and a product of arity $n - 1$. Therefore, everything is expressible via a terminal and a binary.

Examples

The object A^B can be interpreted as an object of arrows from B to A. We must keep in mind, though, that it is not necessarily a set of arrows. It is an object, and its internal structure and relations with other objects depend on the category $\mathcal{C}$.

Example 1 In the category $\mathcal{S}et$, then A^B is the set of functions from B to A.

Example 2 What does it look like in other categories? If we take $\mathcal{F}in\mathcal{V}ect$, the category of finite-dimensional vector spaces and linear transformations (think of them as matrices), then A^B is the vector space of transformations (matrices) $B \to A$, and its dimension is $dim(A)^{dim(B)}$.

Example 3 Here is an example of a Cartesian closed category that has nothing to do with $\mathcal{S}et$: a Heyting algebra H (see Chapter 5). We know that Heyting algebras have conjunction and implication operations, and by Modus Ponens (or, rather, by the definition of implication):

$$(b \wedge a) \leqslant c \equiv b \leqslant (a \to c)$$

The operation of conjunction, $\_ \wedge a$, is a functor $H \to H$, since $x \leqslant y \to (a \wedge x) \leqslant (a \wedge y)$. Similarly, implication, $a \to \_$, is an endofunctor in the category H (note that the character "$\to$" used here has two meanings: one for implication inside H and another for a functor defined on H, that is, an arrow in $\mathcal{C}at$).

These two functors are adjoint:

$$(\_ \wedge a) \dashv (a \to \_)$$

We also need a terminal object—$\top$ is one.

As you can see, any Heyting algebra H is a Cartesian closed category.

Features of a CCC

Properties of Products

- 1 is the neutral element for product operations: $A \times 1 \cong A$ (the relation $\cong$ means "isomorphic"). Why? Because a product is defined up to an isomorphism, and A is as good a candidate for $A \times 1$ as any other.
- $A \times B \cong B \times A$. Why? Because *swap*, which maps $A \times B$ to $B \times A$, is an isomorphism: $B \times A$ is as good a product as $A \times B$, and all products are isomorphic.
- Products are associative, up to an isomorphism: $A \times (B \times C)$ is isomorphic to $(A \times B) \times C$. For the same reason, each one can replace any other.

Properties of Exponential

The following properties do not require much proof. Each of them follows from the fact that an adjoint is unique, up to an isomorphism. So, these properties are just the analogs of the properties of product:

- 1^A is isomorphic to 1.
- A^1 is isomorphic to A.
- $A^C \times B^C$ is isomorphic to $(A \times B)^C$.
- $A^{B \times C}$ is isomorphic to $(A^B)^C$.

Given two exponentials, A^B and B^C, we can define a map $compose : B^A \times C^B \to C^A$ by taking its left adjoint $eval \circ eval : A \times B^A \times C^B \to B \times C^B \to C$.

Remember, an arrow $1 \to A$ is called a point of A. From the definition of exponential as right adjoint to product, it follows that any point $1 \to B^A$ is in a one-to-one correspondence with an arrow $A \to B$.

Definition: Bicartesian Closed Category

A CCC that has finite coproducts (disjoint sums and an initial object), and in which product is distributive over finite coproducts, is called a *Bicartesian Closed Category* ($BCCC$).

Distributivity here means that $A \times 0$ is isomorphic to 0, and $A \times (B + C)$ is isomorphic to $A \times B + A \times C$.

Coproducts have properties dual to the properties of products:

- 0 is neutral for sums: $A + 0 \cong A$.
- $A + B \cong B + A$.
- Sum is associative, up to an isomorphism: $A + (B + C)$ is isomorphic to $(A + B) + C$. For the same reason, each one can replace any other.

Due to distributivity, the following properties hold in a Bicartesian Closed Category:

- A^0 is isomorphic to 1.
- A^{B+C} is isomorphic to $A^B \times A^C$.

These properties follow from the adjunction. As an example, I will demonstrate why the second one holds. We need to prove that A^{B+C} is a product of A^B and A^C. A product is defined by its universal property: any pair $(f : U \to A^B, g : U \to A^C)$ is uniquely defined by $(f, g) : U \to (A^B \times A^C)$.

Now, such a pair corresponds to a pair $(f' : U \times B \to A, g' : U \times C \to A)$, which, by the definition of a coproduct, is defined by the following pair:

$$\begin{pmatrix} f' \\ g' \end{pmatrix} : U \times B + U \times C \to A$$

that is, by an arrow $U \times (B + C) \to A$, which corresponds, via the familiar adjunction, to the arrow $U \to A^{B+C}$.

Conclusion

The special value of Cartesian-Closed Categories is that, due to the Curry-Howard-Lambek correspondence, they are equivalent to the typed lambda calculus. Roughly speaking, in a typed lambda calculus, types are defined as a set of predefined types and types of functions from one type to another. Every expression also has a type. We can build expressions using variables, applications, and abstractions (in lambda calculus, an application is like a function call, and an abstraction is like a function definition).

Such a calculus defines a CCC (in which objects are types of the calculus), and every CCC defines a typed lambda calculus, in which objects of the CCC play the role of types. The theorem consists of stating that this correspondence is an isomorphism.

Further discussion of lambda calculus, as well as the Curry-Howard-Lambek correspondence, is beyond the scope of this book.

Chapter 14. Monads

Main Ideas

We will start with a vague explanation and examples from Scala.

Example 1. A Monad in Scala In a programming language, a functor is a parameterized type `F[T]` (e.g., `Set[T]`) which defines an operation `map(f):F[T]=>F[U]` for each function `f:T=>U`. Since the domain and codomain category of such a functor is the same, it is an *endofunctor*.

In Scala, given two composable functions, `f: A=>B` and `g: B=>C`, their composition is denoted as `(g compose f): A=>C`.

In a programming language, a monad is a functor `F` with two additional functions:

- *unit* `u[T]: T => F[T]`
- *multiplication* `m[T]: F[F[T]] => F[T]`

`List[T]` is a standard example of a monad. First, it is a functor, mapping a type `T` to a list of instances of type `T`. Second, the operation for forming a singleton list, `u(t) = List(t)`, is the unit transformation. And third, the function `flatten:List[List[T]] => List[T]` serves as "multiplication."

A monad must have the following additional properties:

- For an `f:T=>U`, the function `(t:T) => u[U](f(t))` is equal to the function `(t:T) => F[T].map(f)(u[T](t))`. In plain words, we can either first apply `f`, then build a singleton (if dealing with `List[T]`) or first build a singleton and then map it via `f`. This means that the unit `u` is a natural transformation (see Chapter 12).

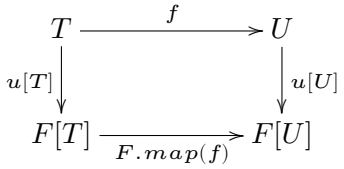


Figure 55: *A monadic unit is a natural transformation.*

- For an `f:T=>U`, we can provide two paths from `F[F[T]]` to `F[U]`. The first path consists of applying the twice-lifted `f`, that is, `F[F[T]] => F[F[U]]`, and then

flattening it with `m[U]: F[F[U]] => F[U]`. Their composition is `m[U] compose F[F[T]].map(F[T].map(f))`. Another path consists of first flattening `m[T]: F[F[T]] => F[T]` and then applying `F[T].map(f)`—that is, using `F[T].map(f) compose m[T]`. These two paths should be equal, which means that `m` is a natural transformation.

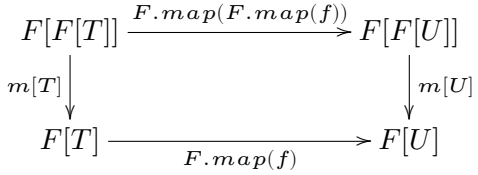


Figure 56: *Multiplication is a natural transformation.*

- `u` is neutral with regard to multiplication:
  `m[T] compose map(u[T]) == m[T] compose u[F[T]] == id[F[T]]`.

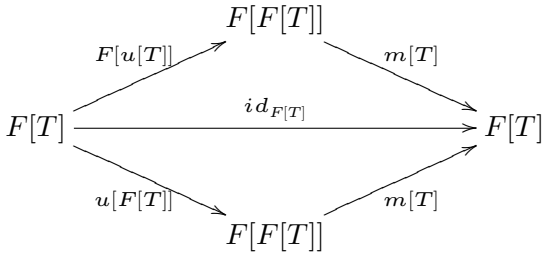


Figure 57: *'u' is a unit for multiplication 'm'.*

- Multiplication is associative:

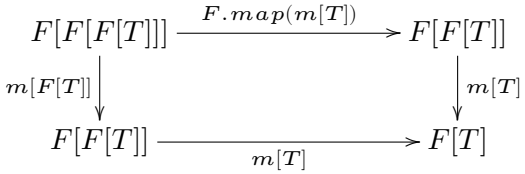


Figure 58: *Associativity of multiplication.*

These properties may be hard to assert for every specific monad, but if we look at the example of lists, the properties are obvious. For instance, consider associativity: for a `list` of `list`s of `list`s, the property says that when we flatten it to a `list`, we can either first flatten externally,

then flatten the result, or do flatten internally, on each element, and then flatten the result. Look at the following example of a `list` of `list`s of `list`s:

```
List(List(List("a11", "a12"), List("a21")),
List(List("b11"), List("b21", "b22")))
```

These can be first transformed to a list of four lists of strings:

```
List(List("a11", "a12"), List("a21"), List("b11"), List("b21", "b22"))
```

Then, we can flattern it further to a list of eight strings:

```
List("a11", "a12", "a21", "b11", "b21", "b22")
```

Or, the flattening can be done inside first:

```
List(List("a11", "a12", "a21"), List("b11",, "b21", "b22"))
```

The axiom of associativity says that it does not matter.

Definition: Monad

Given a category $\mathcal{C}$, an endofunctor $F : \mathcal{C} \to \mathcal{C}$ together with two natural transformations, $u : Id \to F$ and $m : F^2 \to F$, is called a *monad* if m is associative and u is a unit for the multiplication m. The meaning of these two properties is explained in the diagrams below:

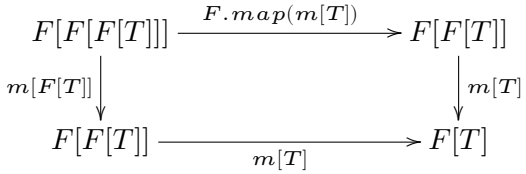


Figure 59: *Associativity of multiplication.*

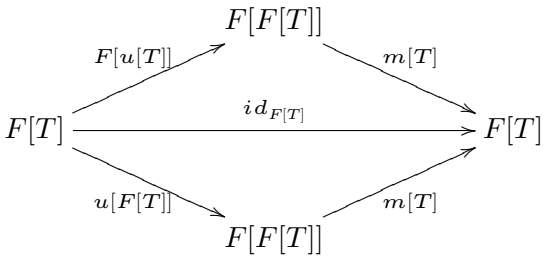


Figure 60: *Neutrality of unit.*

Examples of Monads

Example 2. Identity Functor. The simplest imaginable monad is an identity functor. Both unit and multiplication are identity natural transformations, $id_{Id_C} : Id_C \to Id_C$.

Example 3. Functor $X + E$. In a category that has an initial object and unions, fix an object E, and define a functor $\_ + E$ mapping an object X to $X + E$, and an arrow $f : X \to Y$ to $f + E : X + E \to Y + E$. This arrow is the same as f on X and is an identity on E. Can it be a monad?

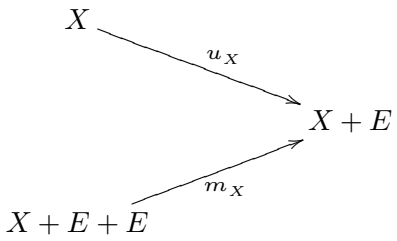


Figure 61: *Monad "$\_ + E$".*

Start with multiplication. We will need $X + A + A \to X + A$ for all X. This is the same as having an arrow $A + A \to A$, which is defined on each component A as its identity. So, now we have multiplication, but what about the unit? We will need $X \to X + A$ for each X, and this is induced by the inclusion of the initial object (e.g., the empty set) into $A : 0 \to A$. You can check that multiplication is associative and that the unit is neutral for multiplication.

Please note that the associativity of monad multiplication defined above has nothing to do with the associativity of sums (whereby $A + (A + A)$ is isomorphic to $(A + A) + A$). These are just two completely different kinds of associativities.

The monad defined above is known as *exception monad*. The object E is "an object of exceptions."

We can view this monad as a special case of a side effect, where a function either evaluates or throws an exception (that is, produces a value of type E instead of a value of type A).

As a special case of such a monad, take $A = 1$ and add a terminal object, also known as a "point." What we get, a functor $Z \mapsto Z + 1$, is known as a *Maybe* (or *Option*) monad.

Example 4. Functor $X \times A$. In a category that has products, a functor $X \mapsto X \times A$ may have a monoidal structure. To ensure that this functor is a monad, we would need a unit $X \to X \times A$ for every X and a multiplication $X \times A \times A \to X \times A$.

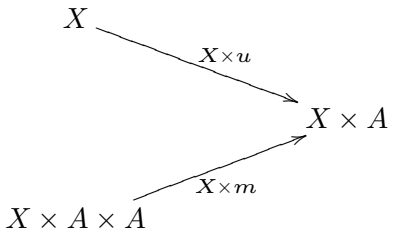


Figure 62: *Monad "$\_ \times A$".*

When the category has a terminal object, it would imply that we have the following:

- $u : 1 \to A$
- $m : A \times A \to A$

The requirements of monadic associativity and unit mean that (A, u, m) is a monoid in our category.

Example 5. Functor `List` in Scala. We can view it either programmatically or categorically. In Scala, `List[T]` is a functor and has a constructor `T=>List[T]`, which builds a singleton list. This constructor is a unit for the monad we are building. `List.flatten` serves as monad multiplication. You can check that `List[List[T]].flatten: List[List[T]]=>List[T]` is associative. Namely, `List[List[List[T]]].map(flatten) andThen flatten` is equal to `List[List[List[T]]].flatten andThen flatten`. This is associativity.

The neutrality of the singleton constructor means the following. If we map a list `xs: List[T]` to a list of singletons, `xss1: List[List[T]]` and then flatten the result, we obtain the same list `xs`. Or we can make a singleton list `xss2: List[List[T]] = List(xs)` and then flatten the result—we again obtain the same list `xs`.

Example 6. Functor `List` in a Category A generic *List*, aka *Kleene Star*, was already discussed in Chapter 2 as a free monoid. Here, we will look at it from a different perspective, discussing a functor that, given a set (an object) X, produces a Kleene Star.

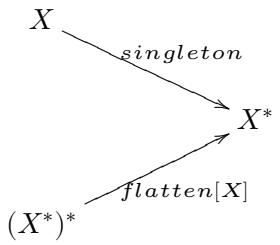


Figure 63: *Monad* $List$ *("*$X \mapsto X^*$*").*

Categorically, a list functor (if it exists), is defined like this: for a given object X, build $X \times X$, $X \times X \times X$, etc. and then sum them all up: $List[X] = 1 + X + X \times X + X \times X \times X + ...$. Here, 1 is a terminal object, and + denotes a disjoint union. So far, infinite sums have not been mentioned. Assume they exist (they don't have to).

Informally, such a sum can be represented as $List[X] = 1/(1 - X)$. Since division is not known yet as an operation on objects, we transform this equation to $List[X] \times (1 - X) = 1$. But wait: subtraction is also not defined on objects! So, we will transform this formula again, getting $List[X] = 1 + X \times List[X]$.

Not only is it legal, but it also looks like a pretty elegant way to define a list. A list is either an empty list, or an element of X followed by some list. Alternatively, as an exercise, we can look at $List$ as a fixpoint $F[X]$ of a functor $1 + X \times \_$ and then try to describe multiplication and unit for this to be a monad. Again, its existence is not guaranteed.

Example 7. Functor Pow In the $\mathcal{S}et$ category, define the power set endofunctor $Pow : X \mapsto Pow[X]$. While it is clear how objects in this category are mapped, defining the mapping of arrows is not as obvious. Given a function $f : X \to Y$, we need to properly define a function $Pow[f] : Pow[X] \to Pow[Y]$, so that an identity maps to an identity, and a composition maps to a composition.

As described in Example 7 of Chapter 12, a good solution would be to map $U \in Pow[X]$ to its image under f, that is, $\{y \in Y \mid \exists x \in U, y = f(x)\}$.

This functor turns into a monad if we observe that $singleton : x \mapsto \{x\}$ is the required unit, and $\bigcup : Pow[Pow[X]] \to Pow[X]$ serves as multiplication for this monad.

Every Adjunction Gives a Monad

In Chapter 12, adjunctions were introduced. Given two categories, $\mathcal{C}$ and $\mathcal{D}$, and two functors, $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$, they are called adjoint if there is a one-to-one correspondence between $F[X] \to Y$ and $X \to G[Y]$. Alternatively, an adjunction can be defined via two natural transformations, unit $\eta : id_{\mathcal{D}} \to GF$ and counit $\epsilon : FG \to id_{\mathcal{C}}$. These two transformations allow us to define a monad structure for the functor $M = GF$.

How does it happen? The unit u is the same as η and the multiplication operation $m[X] : GFGF[X] \to GF[X]$ is defined as $G[\epsilon[F[X]]]$.

We can now get back to examples of adjunctions in Chapter 12 and see what kinds of monads they produce.

Example 8. A Monad for Partial Functions For two categories, $\mathcal{S}et$ and $\mathcal{S}et_{Part}$, we already defined the inclusion $\mathcal{S}et \hookrightarrow \mathcal{S}et_{Part}$, and its right adjoint is the functor $Option : \mathcal{S}et_{Part} \to \mathcal{S}et$. The composition of the inclusion with this $Option$ functor gives us a familiar functor with the same name, $Option : \mathcal{S}et \to \mathcal{S}et$. This functor is known to be a monad, and its monadic structure is defined by the η and ϵ operations of the adjunction above.

Example 9. A Monad for Binary Relations In Example 7 from Chapter 12, we saw an adjunction between an inclusion of $\mathcal{S}et$ into $\mathcal{R}el$ and the Pow functor, also mentioned in Example 5 in this chapter. That adjunction produces the same monad as described in Example 5.

Example 10. Square Functor Looking at Example 9 from Chapter 12, we have another adjunction, $\Delta \dashv \Pi$, where $\Delta[X] = X \times X$. As in the previous examples, the composition $\Pi\Delta$, which maps $X \mapsto X \times X$, becomes a monad if we involve the unit and counit of this adjunction.

We will then have a monadic unit $u : X \to X \times X$, which is just a diagonal inclusion, and a monadic multiplication $m : (X \times X) \times (X \times X) \to X \times X$, which is, (if you look closely) a pair of two projections, $(p_1, p_2) : (X \times X) \times (X \times X) \to X \times X$, where $p_1 : X \times X \to X$ is the first component projection and $p_2 : X \times X \to X$ is the second.

In summary, the multiplication in this case takes two external components, $X \times X \times X \times X$, and ignores the two internal ones.

Example 11. Power of X If we are in a bicartesian closed category, the previous example would look like this, instead: the functor is X^2 (where 2 means 1+1), the monad unit is the exponential of $2 \to 1$, and the multiplication is the exponential of the diagonal $\Delta : 2 \to 2 \times 2$.

These two arrows, $2 \to 1$ and $2 \to 4$, can be lifted—for any X—to $X = X^1 \to X^2$ and $X^4 \to X^2$.

We don't need to be limited to 1 and 2. Any object Z has an arrow $Z \to 1$ and an arrow $\Delta_Z : Z \to Z \times Z$. Due to the associativity of products, and the fact that 1 is a unit for that operation, the functor $(\_)^Z : X \mapsto X^Z$ extends to a monad. In this monad, multiplication $(X^Z)^Z \cong X^{Z \times Z} \to X^Z$ is just X^{Δ_Z}, and $X \to X^Z$ is just an adjoint to the projection $X \times Z \to Z$.

Example 12. The Visitor Pattern is a Monad This example is courtesy of Kris Nuttycombe:

```
trait A {
  def accept[T](v: V[T]): T
}

class B(val s: String) extends A {
  override def accept[T](v: V[T]): T = v.visit(this)
}

class C(val s: String) extends A {
  override def accept[T](v: V[T]): T = v.visit(this)
}

trait V[T] { outer =>
  def visit(b: B): T
  def visit(c: C): T

  def map[U](f: T => U): V[U] = new V[U] {
    def visit(b: B): U = f(outer.visit(b))
    def visit(c: C): U = f(outer.visit(c))
  }

  def flatMap[U](f: T => V[U]): V[U]= new V[U] {
    def visit(b: B): U = f(outer.visit(b)).visit(b)
    def visit(c: C): U = f(outer.visit(c)).visit(c)
  }
}

object V {
  def pure[T](t: T): V[T] = new V[T] {
    def visit(b: B) = t
    def visit(c: C) = t
  }
}
```

In this example, `V[X]` is a functor that has a property of a monad: `map` provides the functoriality of this parametric type.

The `pure` function plays the role of the monadic unit, since it maps `T` to `V[T]`, and specifying

`flatMap` gives us an alternative to specifying monadic multiplication, which can be defined as:

```
flatten[T] = flatMap[V[T]](identity[V[T]])
```

Conclusion

Experiment with a variety of monads you know from programming, and try to find out what kind of adjunction each one is coming from or which adjunctions can come from each monad. Note that there's not just one adjunction available—there's a whole category of adjoint pairs of functors for each monad. Two of these canonical pairs of functors will be studied in the next chapter.

Chapter 15. Monads: Algebras and Kleisli

While monads are already an interesting and useful concept, they also give rise to other interesting concepts, such as *algebras* and *Kleisli categories*.

Monad Algebras

Definition: Algebra

Given a monad M in a category $\mathcal{C}$, an algebra over M is an object A of $\mathcal{C}$ with an arrow (called *action*) $\alpha : M[A] \to A$ such that they are compatible with M's unit and multiplication, namely:

- The composition of unit u and action α is an identity: $A \to M[A] \to A$.
- $M[M[A]] \to M[A] \to A$, which can be built by either applying multiplication then action or by mapping action and then action again. The result will be the same.

$$\begin{array}{ccc} M[M[A]] & \xrightarrow{M[\alpha]} & M[A] \\ {\scriptstyle m_A}\downarrow & & \downarrow{\scriptstyle \alpha} \\ M[A] & \xrightarrow[\alpha]{} & A \end{array}$$

Figure 64: *Algebra action and monad multiplication.*

This definition may look somewhat complicated, but it is actually less complicated than it appears. Following is the first example.

Example 1. Free Algebra $M[X]$ For a monad M and any object X, the arrow $m[X] : M[M[X]] \to M[X]$ provides an algebra over M.

$$\begin{array}{ccc} M[M[M[A]]] & \xrightarrow{M[m_A]} & M[M[A]] \\ {\scriptstyle m_{M[A]}}\downarrow & & \downarrow{\scriptstyle m_A} \\ M[M[A]] & \xrightarrow[m_A]{} & M[A] \end{array}$$

Figure 65: *Free algebra* $M[X]$.

The fact that u is neutral for m and that m is associative gives us the required algebra properties.

Example 2. Algebra over $Option$ Monad As shown before, the *Option* monad consists of adding a terminal object, $X \mapsto X + 1$.

To have an algebra, we need to properly define $X+1 \to X$. Since, by the requirement to preserve the unit, this arrow is already defined on the X part, we only need $1 \to X$, that is, a point in X, or, in programming language parlance, an instance of type `X`.

The choice is arbitrary, and, in the world of programming, opinions vary. On the one hand, in languages that have `null`, we can use `null` as a special instance of type `X`. On the other hand, Java philosophy suggests using the "null object pattern," whereby a special instance is created to denote "nullish" values. Again, the choice is arbitrary, but that's all we need for an algebra over an *Option* monad to be properly defined.

The Java tradition (or pattern) of having a special "null value" for each used type makes the whole programming style "algebraic"—it provides an algebra for each type. This helps Java programmers deal with the fact that most functions are not total. To make a function total, they lift it, using a default value when there is no value.

If you see a program that, when it cannot not detect a user name, assumes it is an empty string, or another program, when being unable to access a bank website, decides that your bank balance is `$0.00`, these are typical abuses of algebras over *Option*.

Example 3. Algebra over $List$ monad To have such an algebra with an underlying object X, we introduce an arrow $fold : List[X] \to X$, compatible with both singletons and list flattening.

Compatibility with singletons means that $fold(\{x\}) = x$: a singleton folds into its contents. Compatibility with list flattening means that we can either first flatten then fold or fold all the individual components and then flatten.

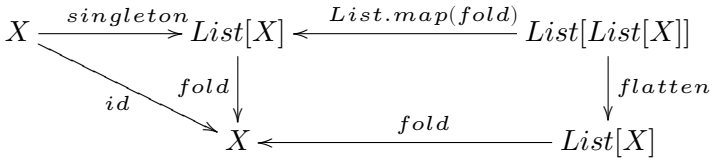


Figure 66: *Algebra over $List$ monad.*

We can prove that this feature is provided if we have an associative binary operation $op : X \times X \to X$ along with an arrow from the empty list, $u : 1 \to X$. Taken together, this gives us a monoid. They are the same: having a monoid or having an algebra over the *List* functor. Note that lists are also monoids, so, they are also algebras over the *List* monad. Lists are actually free monoids, as described in Example 1.

Example 4. Action of a Monoid Take an arbitrary monoid A and a functor $X \mapsto X \times A$. It is known to be a monad (see Chapter 14).

What would an algebra over this monad be? We would need an action $X \times A \to X$ that has the appropriate compatibility with monoidal operations. This is called an action of monoids. If we were in $\mathcal{S}et$, we could talk about sets with the action of A over them, which amounts to monoid arrows $A \to X^X$. We already know that the set of functions $X \to X$ is a monoid.

Category of Algebras

Yes, given a category $\mathcal{C}$ and a monad M, all algebras over M form a category. To demonstrate this, we need to define arrows between algebras. Such an arrow $f : X \to Y$ should preserve the action, $\alpha_Y \circ M[f] = f \circ \alpha_X$—see the diagram:

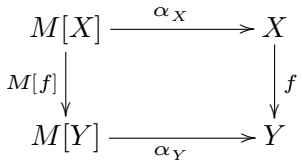


Figure 67: *Arrows between algebras.*

Since the arrows are defined on the underlying objects, we can see that a) identity arrows remain algebra arrows, and b) a composition of two algebra arrows is an algebra arrow. Thus, we have a category, and this category is named $\mathcal{C}^M$, in honor of its underlying category $\mathcal{C}$ and monad M.

Of course, given an algebra, or two algebras and an arrow between them, we can always forget the "algebra" part and just think of such an arrow as a regular arrow in $\mathcal{C}$. There is a natural mapping of algebras to their underlying objects as objects in $\mathcal{C}$ and of algebra arrows to themselves as just arrows in $\mathcal{C}$. This functor is called *forgetful*: $Forget : \mathcal{C}^M \to \mathcal{C}$.

Free Algebras as Functors

Out of all the algebras that we can find for a given monad M, one kind is especially easy to find (see Example 1): given an object X, take $M[X]$; it is already an algebra, with monad M multiplication as the action arrow, $M[M[X]] \to M[X]$.

Note that building a free algebra is natural, that is, if we have $f : X \to Y$, we can produce an algebra arrow $M[f] : M[X] \to M[Y]$, which is just a lifting of f into the world of algebras. As a result, we have a functor, $Free : \mathcal{C} \to \mathcal{C}^M$.

Forgetting and Freedom

There is a duality between taking an algebra and forgetting that it was an algebra and taking an object and building an algebra on it right away. Actually, the two functors are adjoint: $Free \dashv Forget$.

Given a category $\mathcal{C}$ and a monad M over it, we'll denote algebras over M, that is, objects of $\mathcal{C}^M$ as $M[Y] \to Y$.

For an object X of $\mathcal{C}$, a free algebra $Free[X]$ over it is then $M[M[X]] \to M[X]$.

An algebra arrow $f : Free[X] \to (M[Y] \to Y)$ gives us an arrow $M[X] \to Y$ in $\mathcal{C}$, which, composed with the unit $u_X : X \to M[X]$, gives us an arrow $f' : X \to Y$.

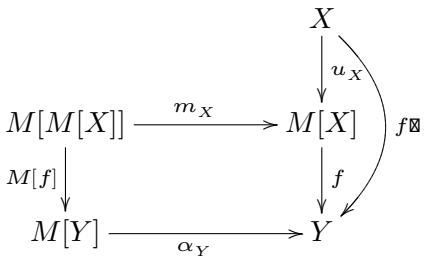


Figure 68: *Arrow from free algebra.*

On the other hand, starting with an arrow $g : X \to Y$ in $\mathcal{C}$, we can lift it to $M[g] : M[X] \to M[Y]$. Since $M[Y] \to Y$ is an algebra, we can compose $M[g]$ with the algebra action and have $g' : M[X] \to Y$. It remains to prove that this gives us an algebra arrow, which is an easy exercise.

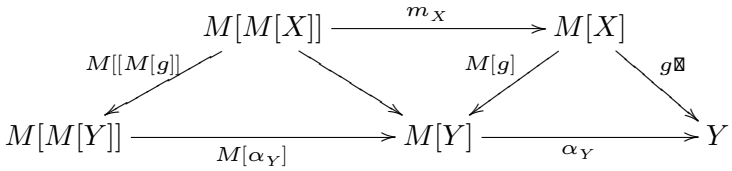


Figure 69: *Lifting an arrow to an algebra arrow.*

Although we already have some monads, it may make sense to demonstrate this adjunction with a couple of examples.

Example 5. *Option* **Monad and its Algebra Adjunction** As was shown before, algebras over the *Option* monad in a category $\mathcal{C}$ are objects X with a point $1 \to X$, and arrows between such algebras should preserve these points.

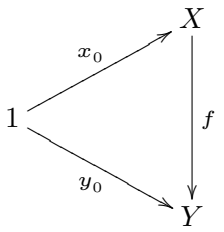


Figure 70: *An arrow in the $\mathcal{C}^{Option}$ category.*

In this specific case, the forgetful functor will take an object with a point and forget that there was a point. The whole adjunction looks like this:

$$\frac{X + 1 \to Y \quad in\ \mathcal{C}^{Option}}{X \to Y \qquad\qquad in\ \mathcal{C}}$$

Kleisli Category

In Chapter 14, we saw how a monad is produced by an adjoint pair. One of the candidates for such a pair is $Free \dashv Forget$, but there are others with the same effect. Following is another such pair.

Definition: Kleisli Category

Given a monad M in a category $\mathcal{C}$, we can define it via the *Kleisli Category* for that monad, denoted as $\mathcal{C}_M$. This category is built from $\mathcal{C}$ and M in the following way:

- Objects of $\mathcal{C}_M$ are objects of $\mathcal{C}$.
- Arrows of $\mathcal{C}_M$ are arrows of $\mathcal{C}$ that have the following form: $f : X \to M[Y]$. While the codomain is "actually" $M[Y]$, this arrow is considered to be an arrow $f : X \to Y$ in $\mathcal{C}_M$.

But is this a category? How is identity defined? How is composition of arrows defined?

The composition of $f : X \to Y$ and $g : Y \to Z$ in $\mathcal{C}_M$ is defined in the following way: we have $f : X \to M[Y]$ and $g : Y \to M[Z]$ in $\mathcal{C}$. These two give us $h = (m_Y \circ M[g] \circ f) : X \to M[Z]$,

that is, an arrow $h : X \to Z$ in $\mathcal{C}_M$. Its associativity follows from the associativity of monad multiplication.

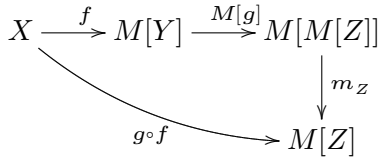


Figure 71: *Composition in a Kleisli category.*

Now, the monadic unit $u : X \to M[X]$ (in $\mathcal{C}$) gives us $u : X \to X$ (in $\mathcal{C}_M$), and when you compose it with another arrow, you see that the other arrow remains intact, so we have an identity.

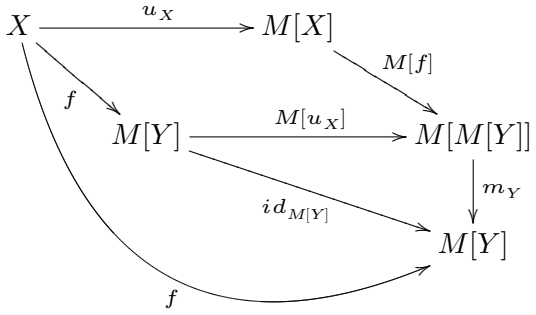


Figure 72: *Kleisli identity is neutral for composition.*

In this picture, all squares and triangles are commutative, so you can see that composing, in Kleisli, u_X with an $f : X \to M[Y]$ gives us the same f.

As you can see, we have a category $\mathcal{C}_M$, but how is it related to the original category $\mathcal{C}$? How is M defined by this relation? We need to build two functors, $\mathcal{C} \to \mathcal{C}_M$ and $\mathcal{C}_M \to \mathcal{C}$, then check that they are adjoint and that their composition is M.

The first functor, $\mathcal{C} \to \mathcal{C}_M$, is almost an inclusion. It maps objects to themselves and an $f : X \to Y$ to $u_Y \circ f$. We must check the functoriality, of course, but this is trivial.

The second functor, $\mathcal{C}_M \to \mathcal{C}$, maps an object X to $M[X]$ and an $f : X \to Y$ in $\mathcal{C}_M$, that is, an $f : X \to M[Y]$ in $\mathcal{C}$ to $m_Y \circ M[f] : M[X] \to M[M[Y]] \to M[Y]$.

The composition of these two functors takes an X to $M[X]$ and an $f : X \to Y$ to $M[f] : M[X] \to M[Y]$.

Examples of Kleisli Categories

Example 6. Kleisli Category for $Option$ Monad in $\mathcal{S}et$ This category has the same objects, and for arrows, it has functions of the form $X \to Y + 1$, which is, as you saw in Example 6 of Chapter 12, equivalent to partial functions, $X \nrightarrow Y$. So, in this case, the category $\mathcal{S}et_{Option}$ is "the same as" (that is, isomorphic to) the category of sets and partial functions, $\mathcal{S}et_{Part}$.

Since we have the same monad, we can consider using the Kleisli category instead of the category of algebras for software implementation. This means that we won't need any null objects, but instead, we will deal with partial functions, which probably better reflects the facts of reality: that not every function is total. In Scala, the code that works with the Kleisli category for the `Option` monad typically looks like this:

```
for {
  user <- findUser(userId)
  page <- user.loadProfilePage
  password <- findPassword(page)
} {
  doSomething(user, password)
}
```

Here `findUser` produces an `Option[User]`, `loadProfilePage` produces an `Option[Html]`, and `findPassword` produces an `Option[String]`. These three functions don't necessarily produce a good result, but they are combined to produce a reasonable answer in the case of success and no answer in the case of any failure. That's how partial functions are expected to compose.

Example 7. $\mathcal{R}el$ is a Kleisli Category for Pow Monad in $\mathcal{S}et$ For this monad, an arrow in its Kleisli, $\mathcal{S}et_{Pow}$, is an arrow $X \to Pow[Y]$, that is, a binary relation on $X \times Y$. Of course, we must at least prove that the composition defined in this Kleisli category is the same as the composition of binary relations, but it follows from definitions, so you can just do this as an exercise.

Example 8. Kleisli Category for the "Action of a Monoid" Monad This monad is from Example 4, where we saw an algebra over a monoid action. Since the monad consists of multiplying X by a specified monoid A, a Kleisli should consist of the same objects (e.g., from $\mathcal{S}et$) and arrows of the kind $X \to Y \times A$. A composition $X \to Y \times A \to Z \times A \times A$ should multiply two elements of A. This is a typical case of functions with side effects, where the side effect consists of values of the monoid, and two side effects combine by applying the monoid operation.

For a specific case where the monoid is a monoid of a `String` type, the side effect can be thought of as logging.

Plurality of Adjunctions

For a given monad M over a category $\mathcal{C}$, there are at least two distinct adjoint pairs producing this monad: algebras, $\mathcal{C}^M$, and Kleisli, $\mathcal{C}_M$. Actually, except for the trivial cases, there's a whole (and large) category of adjoint pairs producing the same monad—algebras and Kleisli are extreme cases. In this category of adjunctions, the Kleisli category is an initial object, and the category of algebras is a terminal object.

Conclusion

Thank you for reading my book. You should now have a broader picture of the areas of mathematics that are currenlty used in modern programming or are becoming increasingly popular within the programming community. I hope this picture will help you properly deal with the fast-changing world of programming paradigms. If you are interested in delving deeper into other topics of mathematics, read books that include proofs and excercises to gain practical skills.

Bibliography

Faure, R., Kaufmann, A., Denis-Papin, M. *Mathémathiques Nouvelles*, Dunod, 1964.

Hein, J. L. *Discrete Structures, Logic, and Computability*, Jones & Bartlett Learning, Fourth edition, 2015.

Mac Lane, S. *Categories for the Working Mathematician*, Springer, Second edition, 1998.

Milewski, B. *Category Theory for Programmers*, [https://github.com/hmemcpy/milewski-ctfp-pdf/releases/download/v0.6.0/category-theory-for-programmers.pdf], 2018.

Pierce, B.C. *Basic Category Theory for Computer Scientists (Foundations of Computing)* The MIT Press; First edition, 1991

FC

Contents