Introduction

This book is written with the specific purpose of helping practicing programmers that have gaps in their education to fill these gaps without having to go to study all over again. The world is changing; mathematics itself is undergoing a great revolution; foundations of the so-called Computer Science are being replaced; the notions that were officially designated as abstract nonsense even by the algebraists are becoming a daily reality in our coding practice.

Let me name a few. Monoids: you need a monoid to do map/reduce, to implement or use a very efficient storage, finger tree, to discuss traversals of applicative functors (this one is a generalization of map/reduce); you need monads and Kleisli category to explain and formalize side effects in your big application; you need irreducible polynomials to deal with CDMA or with CRC. Not much of this has been studied in a regular math course for engineers; the course of mathematics, except for the version studied by post-grads in the best universities, mostly consists of century-old ideas and notions. These ideas may be good, but the world of ideas in computing is already way ahead; and we have to either catch up or perish, mentally and professionally.

This book is not for mathematicians. They have to find better sources of information or inspiration, those that are mathematically strict and contain enough theorems and exercises, and are, probably, targeting a more distant future. E.g. there is no topology here, no homotopy, no functional analysis, no equations. No linear algebra either. Linear algebra is these days very 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, and which have been totally alien to about 90% of practicing programmers.

The reader is expected to be a practicing programmer (but not necessarily); so, instead of a set theory, on most occasions types are being used. These days programmers, for a good reason, know more about types than about sets. Set theory was used as a foundation for mathematics, with axioms making it possible to prove theorems about functions and real numbers that have long been accepted. In programming practice these theorems are often irrelevant or just inapplicable, stating 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 needed in practice. We will cover set theory and its axioms in details later on; just keep in mind that not everything in math is based on set theory. If you are looking for an example, take set theory. It's not based on set theory.

Code samples are mostly in Scala; some are in JavaScript. If you are a Java or C#/++ pro-

grammer, you will understand Scala easily; if you are a Haskell programmer, please be patient, you still may find something new in this book; if you use dynamic types or no types, this may be a good opportunity to get some *practical* ideas with no type theory involved.

Over 50 years ago an amazing book was published, Faure, Kaufman, Denis-Papin, "A New Mathematics"; this book was a rather easy reading, and it was an adventure in math, a version of "Harry Potter". That's the ideal.

Credits:

I am very thankful to my friends who had reviewed chapters of the book while I was writing them; their remarks 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, Pavel Lyutko.

The 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 and surjective and bijective; currying, etc. All these notions are probably taught in high school or in the university; but actually, nobody remembers. Set theory is not actually involved in discussing these notions; we rely, informally, on types.

Chapter 2. Abstractions of Algebra

Monoid: this is the basis of many programming constructs.

Semigroup: these are "monoids without neutral elements". If we fold, instead of reducing, we need just a semigroup.

Magma: it is a "semigroup without associativity" – like tuples and binary trees; they are everywhere in the code.

Laws of associativity, commutativity, idempotence,

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

When we have algebraic structures, we have to define functions that preserve these structures. So such functions are discussed too.

Chapter 3. Partial orders, Graphs, DAGs

A *poset* is a partially-ordered set; it is a very simple, unassuming structure. A special kind of poset is the one where each pair of elements has a minimum and a maximum; add top and bottom elements, and we have two monoids; this structure is called lattice. *Graph* can be defined in many ways; in this book by graph I mean a directed multigraph. If the graph is acyclic (what is known as *DAG*, Directed Acyclic Graph), it can be also looked at as a poset. A *tree* is a special kind of DAG.

Chapter 4. Boolean Logic

Discussing propositional logic, truth tables, conjunction, disjunction, negation, implication, Sheffer stroke, Peirce arrow. Sound and valid arguments, De Morgan laws, double negation. And we have *two monoids* - one with conjunction, another with disjunction.

Chapter 5. Logic Does Not Have To Be Boolean

It is very easy to introduce a logic with more than two logical values; such a logic does not have to support the law of double negation (but it can); a special case is intuitionistic logic. We will see truth tables for intuitionistic logic, and examples; we will see which De Morgan laws work in this case. And we still have two monoids.

Chapter 6. Quantifiers

Universal ∀ and existential ∃, `collection.foreach`, `collection.exists`, Google Guava library, informal checking of sentences with quantifiers; differences between ∃∀ and ∀∃, with code samples.

Chapter 7. Models and Theories

A gentle introduction to Model Theory.

Chapter 8. Category is a Multi-tiered Monoid

Introducing the notion. Comparing categories with monoids and graphs. Many examples of categories, including types in programming languages and database structures.

Chapter 9. Working With A Category

Kinds of arrows; how they are defined in categories. Initial and Terminal Objects.

Chapter 10. Manipulating Objects in a Category

Sums and products of objects, their properties (we have monoids). Examples of products and sums in categories of a different nature; code examples.

Chapter 11. Equalizers and Pullbacks, and their Duals

The notion of *pullback* must be familiar to anybody familiar with the notion of `join` in SQL. An *equalizer* is just a generalization of the idea of fixpoint. All these notions are introduced in a "pointless" style; and the dual notions, *coequalizer* and *pushout*, are also discussed, with examples.

Chapter 12. Relationships between Categories

Categories form a category, where categories themselves are objects, and what are arrows there? They are called *functors*. Sums and products of categories are just special examples of sums and products; these are discussed too.

Chapter 13. Relationships 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. Introducing *adjoint functors*.

Chapter 14. Monads

Looking at an adjoint pair of functors, we discover monoid-like properties and build a functor with such properties. It is called a *monad*. We look at several examples of monad; then study the notion of algebra and free algebra, 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 Kleisli category are defined, and examples demonstrate these categories for certain popular monads.

Chapter 1. Functions

General Ideas

You either know, or you think you know, you are supposed to know what a function is. Definitions and opinions vary; one of them is that a function is 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 to practice; so we will just ignore it and will be much more vague, and in this vagueness 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.

There are many ways to specify a function; one of them is the set-theoretical way, as mentioned above, that is, is pretty much like we define a Map literal, by just listing key/value pairs. The only thing, though, is that, to do this in the actual code, we need a type A that is finite (and rather small), to be able to practically list all such pairs, like in the Scala code below:

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

(the code says that 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 type `A` can have, this is a legitimate way of defining a function. If not, we actually have a partial function, since for some `a` the function is not defined.

Probably a more general way would be to define

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

A specific example may look like this:

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

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

Among the many ways to define a function, there are ways for which science does not know the result, and, strictly speaking, it is not clear whether we can add "yet" to this sentence or not. Here is an example of such a function (in JavaScript):

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

A pretty simple function, but we don't know if it produces a result for every possible integer number `n`. Of course, the function doesn't have to provide its result immediately. But in this specific case we don't know if we can *ever* provide it.

Nevertheless, while the feasibility of providing a result for every argument may be under question, we still classify it as a function.

So far we did not put any constraints on functions; but we could; this way we narrow the realm, as long as some laws are satisfied (we definitely want a *composition* of two functions to be a function.

In all these cases we say that function $f : A \to B$ is defined on type A, and takes values in type B.

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

Example 1 `select firstname from users where id=?` – this SQL statement can be considered as specifying a function, with `id` as a parameter. The problem here is that not every id gives us a result; but we can delimit ourselves to a collection of known `id`s, and expect that in normal circumstances we will always have a name as a result.

Counterexample 1 `select from users where firstname=?` is not a function returning a `User` record: for a single first name we can get more or less than one result. We can turn it into a function if we define its codomain as a set of collections of users. Because, see, we can always have a collection, even if the collection is empty.

Counterexample 2 $x^2 + y^2$ 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 of y, call it y_0, and then $x^2 + y_0^2$ becomes a function of just one parameter, x.

Counterexample 3 Java static method `System.currentTimeMillis()` is not a function, because its value changes independently of any parameter we pass (we pass no parameters). For some obscure reason, it is customary in programming practice to treat these things as functions. Actually this is just a name of a channel from which input is coming. It is not a functin, 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 is it. We just get its value as an input. *We are a function.*

Now let us introduce the proper terminology.

Main Definitions

Definition Given a function $f : A \to B$, we call A the *domain* of f, and B the *codomain*, also known as *range*, of f.

Note here 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. For instance, we can have 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 at the time when the function is defined, does not make the singleton $\{b_0\}$ its codomain.

Example 1 First, let me remind you the traditional notation. $\mathbb{R}$ is a type of real numbers; they can be represented in various ways in your code; how exactly, is not relevant now.

$sin : \mathbb{R} \to \mathbb{R}$ – this is a regular sine function; of course, with parameter being a real number, it only takes values in range $[-1.0, 1.0]$; but we here define its codomain as $\mathbb{R}$. We have the right to do this (why?).

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

Here it is defined in Scala (with type parameter `A`, because all identities are different):

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

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

But we have to define composition first…and we also did not talk about what does it mean that two functions are the same.

Definition 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 $a : A$.

We just have, given two functions, 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 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 maybe one of them takes hours to calculate and another is just a constant – like is (supposed to be) the case with the function `s2c2` above.

This also means that two clearly different expressions can define the same function; and we do not care about 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, by just comparing expressions, see that $(h \circ g) \circ f = h \circ (g \circ f)$, where h is some function from C to D. This property of composition is called *associativity*; we'll get back to all this in the next chapter.

Now a couple of examples, in the rare case the reader is still confused.

Example 2 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 take the salary of an employee and calculate its sine. That would be a composition of the two functions.

Example 3 In JavaScript we can easily implement composition of two functions, like this:

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

If you are not familiar with JavaScript, the line above means the following: we define a constant value called compose; this value is a function; this function takes two arguments (`f` and `g`) and returns a value `(x) => g(f(x))`, which is also a function that takes an `x` and returns `g(f(x))`.

Special Classes of Functions

We already saw constant function and identity function. Here are more definitions.

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 this: $f : A \rightarrowtail B$.

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 this latter definition's expressionism may sound more powerful, we are on the territory of negative sentences, and it is known to be less powerful…although in many occasions it is the same thing.

Informally, a mono does not merge values. So a constant function will not be a mono, unless the domain of it consists of not more than value. 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. If A is a subtype of B, in the sense that every instance $a : A$ is also an instance of B, that is, $a : B$, then we have an *inclusion* of A into B. Since inclusion does not merge different objects, we have a monomorphism.

Example 4 Even if the function is not injective, we can restrict (see later) 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 5 `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 repeating names all the time.

Counterexample 6 $sin : \mathbb{R} \to \mathbb{R}$ 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. How come? 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, we have $a_1 = a_2$. QED.

QED stands for "Quod Erat Demonstrandum" – "what had to be proved"

Even the opposite is 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 another exercise to bring an example.

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

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 used as synonyms. They are not exactly synonyms; and this distinction must be covered later on, but for now let's assume they are synonyms.

Informally, an epi covers the whole range, and not necessarily once. Like, say, the function $sin(x)$, defined on the whole class of real numbers, and having 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 epi 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 such an $a : A$ that $f(a) = b$. Taking these two 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 7 Notation: $\mathbb{N}$ is the type of Natural Numbers.

$f : \mathbb{N} \to \mathbb{N}$ defined like this: $f(x) = x/100$. It is integer division. Any number natural number m in the codomain $\mathbb{N}$ is $f(100 * m)$, so there, we have a surjection.

Counterexample 8 $f : \mathbb{N} \to \mathbb{N}$ defined like this: $f(x) = x + 100$. You see that 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 *bijection*, or *one-to-one*, if it is both an injection and a surjection (aka mono and epi). That is, every element in B is the result of applying f to some unique element in A.

You can see that identity function is always a bijection, and that a composition of two bijections is a bijection. One may ask, how about an 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, not all of them work in programming practice.

Example 9 $f : \mathbb{N}_{26} \to \mathbb{N}_{26}$ `f(c) = (a.toInt+(c.toInt-a.toInt + 5) % 26).toChar`

This function maps `from fairest creatures we desire increase` to `kwtr kfnwjxy hwjfyzwjx bj ijxnwj nshwjfxj`. Can you provide the inverse?

Example 10 $f : \mathbb{N} \to \mathbb{Z}$ defined like this: $f(x) = ((i + 1)/2) * (i\%2 * 2 - 1)$. If you don't see what kind of function is it, 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 check that this is a bijection.

Isomorphism

Before defining an invertible function, we should define what an inverse function is.

Definition 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 $g = g \circ f \circ h = h$. QED. Also note, f is the inverse of its own inverse.

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

An isomorphism must be a bijection, that is, 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 the identity is a mono. We know that if a composition of two functions is a mono, then the first one is a mono too; so there.

Similarly, a composition of f^{-1} followed by f is an identity, so it is an epi; hence the second one, f, must 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 a definition of a new function that would serve as 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 it would be if we cannot get its value for a given argument? it is like in real life – the suspect may be hiding too well from the police, even if there is no doubt the suspect exists and is unique. Of course we could apply the Axiom of Choice from a Set Theory… could we, really?

If you write a program, you have some libraries, and suppose such a library has *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 is this: while formally providing an inverse for any bijection may work for sets, but for richer structures, like ordered sets, monoids, trees, functions are defined as preserving that additional structure. And if a function has one-to-one mapping on elements does not guarantee that the structure is preserved: meaning, we may not have an inverse. I'm not giving an example here; we will see later that this is the case.

Definition A *fixpoint* (also known as)for a function f is a values x such that $f(x) = x$.

Example 11 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 12 Again, for a 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, like in the following

Counterexample 13 Take a function $f(x) = x + 1$; this function has no fixpoints. Meaning, 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 14 In one of the previous examples we restricted $sin : \mathbb{R} \to \mathbb{R}$ to the segment $[0.1, 0.2]$ – this is exactly what it is about.

Example 15 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.

Restriction may look like a non-trivial operation; but actually it is just a composition. For a subtype $A_1 \subset A$ we have 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 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 (can be also viewed as $\{f(a) \mid a : A\}$) is called an *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 trying all $a : A$ and checking if we hit it.

Note that there is no dual notion that applies to A; every element $a : A$ maps to something in B.

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

Example 16 Take 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 just a circle of radius 1.

Decomposing a Function

Suppose we have function $f : A \to B$; and have $Im(f)$ 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 on their codomain; there's a popular misconception that these two are the same functions. 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 take another

function, `salary` (assuming that employees somehow get double salaries). `findEmployee` has the signature `Name -> Employee`, and `findEmployee` has the signature `Employee -> Double`.

Now suppose `Employee` is a subtype of `Person`. We can consider 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 salary function. But oops, salary is only defined on `Employee`; and, unless we are in Sweden or North Korea, some people in our country may be not getting any salary at all. We can compose `findEmployee` with `findEmployee`, but we cannot compose `findPerson` with salary.

Now that we have covered it, we can go back to our 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 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 very generic structure; all we need for it 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*. The reader is left to figure out the meaning of it, and to prove that this is the fact.

Example 17 Let's return to the example from the previous paragraph, 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; it us traditionally denoted as s^1. When we decompose it into an epi followed by 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: $\mathbb{R} \overset{e}{\rightarrowtail} s^1 \overset{i}{\twoheadrightarrow} \mathbb{R} \times \mathbb{R}$

Predicate

There is a special kind of function worth noting: the one which has logical values as codomain. It is customary to call logical values "*Boolean*"; we will do it occasionally, but please remember that Boolean logic is not the only one known to humankind, and we have to be prepared to be more generic.

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

Example 18

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

Logical values have some operations on them. 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 mathematics, in Set theory, they are used to create new sets via *set comprehension*: $\{x \in S \mid p(x)\}$. We have a set and a predicate, and build another set out of them.

We can act 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) = S contains x
```

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

More than One Parameter

So far we saw functions that have just one parameter; in real life we may have more. For functions of several parameters we can define domain too, but to do that, we need the notion of Cartesian product, and we will meet it later on. So let's just deal with such functions without specifying their domains 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 gives us an idea that its domain is a product of two types, $A \times B$. We will discuss products much later.

Without defining a domain for such a function, we cannot talk about such a function being an epi or a mono. But what we can do is take 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 $a : A$. So we, essentially, 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 is coming from Curry Haskell, although the whole idea comes from the works of Moses Schönfinkel; but he original idea was (first?) published by Frege in the end of XIX century.

Here is how currying looks in JavaScript.

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

In Haskell they don't even have functions of multiple parameters; all functions by default are in curried form. In Scala, on the other hand, you can explicitly curry or uncurry your function if you feel like it.

Two special kinds of functions of two parameters 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 *binary relation*. A binary relation is informally percieved as a relation between values of two types, A and B; the value of r is true exactly on those pairs (a, b) that are in the relation.

Example 19 Take a database where `A=Person` and `B=Company`, and a relation `worksFor`. Such relations are usually stored in relational databases as a collection of pairs, `(person, company)`. This is like 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 we have a binary relation 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 example above, we would have expressions like `Person worksFor Company`.

In most 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, this includes binary relations. For instance, this expression: `Person worksFor Company` is legitimate in Scala.

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

Binary Relation on a Single Type

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

Definition 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 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 it is just the equality relation.

Example 20. 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 21. 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. It is also both symmetric and antisymmetric.

Definition 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 $x < y$ 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

Example 22

Equivalence Relation

Definitin Equivalence relation is a binary relation R defined on a type A such that it is symmectir, reflexive, and transitive.

A natural case of equivalence is equality, and there are other examples. For instance, in Java (and Scala) one can define `equals` methodf, which tells whether one value can be replaced with another, and to make sure the code behaves properly, this method should be symmetric, reflexive, and transitive. It is not an equality that you define by this method, it is an equivalence relation.Fth

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: $Op : A \times A \to A$.

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

Chapter 2. Abstractions of Algebra

Introduction

You encountered monoids many times in your life and programming practice; but you might not have recognized them. Here they are.

1. Integers with addition. We know that
   - $(a + b) + c = a + (b + c)$
   - $0 + n = n + 0 = n$
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, like in

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

5. Sets with intersection, like in

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

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 monoid.

Formally

First, let me remind you 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 means in a type theory; discussing type theories is not the topic of this discussion.

Now I will give a strict formal definition of monoid.

Definition. Monoid Given a type T, a binary operation $Op : T \times T \to T$, and an instance $Z : T$, with the properties that will be 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*: $(a \ Op \ b) \ Op \ c = a \ Op \ (b \ Op \ c)$

Example 1 $(String, ++, "")$; concatenation is associative, and the empty string is neutral.

Example 2 $(Boolean, \&, true)$; conjunction is associative, and true is neutral.

Example 3 $(\mathbb{N}, max, 0)$ – take natural numbers, and the operation of taking maximum of two numbers, and 0 as a neutral element. We have a monoid.

Example 4 Given a set A, the set of all functions $f : A \to A$ is a monoid, if we use composition as the operation, add the identity function as the neutral element. We have associativity.

Counterexample 5 $(\mathbb{N}, min, ???)$ – if we take natural numbers, and the operation of taking minimum of two numbers, we have almost a monoid (it is called a semigroup); but where do we take a neutral element? There is no such thing here. 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, average is not: $avg(10, avg(30, 50)) \neq avg(avg(10, 30), 50)$. 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 is the type of data, A, $(a, (b, c))$ is not the same as $((a, b), c)$. (With one possible exception of 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 take 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 concatenation.

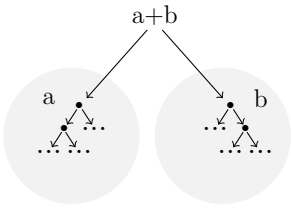


Figure 1: *Binary operation on trees*

This operation is not associative:

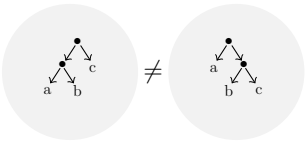


Figure 2: *No associativity*

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?" At least here I have a positive answer to these questions: 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 we have a *fold* notation for this expression:

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

Here we do all calculations sequentially, because we rely on results of the previous calculation. But if we have associativity, we can regroup the calculation and run it in parallel, like in the following picture:

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

Mappings Between Monoids

We have defined monoids; but it is not enough. It is important to study relations between monoids. If we had just 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, mappings between monoids have also to preserve operations.

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

Definition. Monoid Functions For two monoids, (A, Op_A, Z_A) and (B, Op_B, Z_B), we define a *function from one monoid to another* 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 Exponent function; it maps sums to products,

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

Example 10 Length of a string:

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

Example 11 Sum of a list of integers:

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

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

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

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

Free Monoids

If you do not see enough monoids around, you can build one out of anything you have. Just having any A, we can build a monoid from it. In mathematics this monoid is known as A^*, and is called *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,

$X = 1 + A \times X$

You 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... well, not so long ago extraction of square root of -1 was not allowed either.

Now we can; and we get the type of lists, which can also be defined (in Haskell) as

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

(In Haskell colon character, `':'`, means appending of an element to a list, and `Nil` means an empty list.)

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

What's important about lists (aka Kleene stars, aka free monoids) is their universality among other monoids. To define a monoid function 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). More, any such function from A to B gives a monoid function.

Let's see how it works, in terms of lists.

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

$f'(Nil) = Z_B$

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

(Here Nil means an empty list, and $::$ means appending an element to a list.)

Do you see that f' is a monoid function? Preserving neutral element - check. Preserving composition... we can prove it too; except that proving anything is not in our plans. You can just try to see that it is true.

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

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; we just have defined a type consisting of seven possible values).

Now we define the following function, in Scala:

```
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.

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.

Deterministic State Machine

How is a deterministic machine usually defined? We have:

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

For a transition function $f : A \times S \rightarrow S$ we can take its curried version,

$f' : A \rightarrow (S \rightarrow S)$. This function maps our alphabet to a monoid that was described above, a monoid of endomorphisms on S.

Words over alphabet A form A^*, or $List[A]$, the free monoid we already saw above; and we have $f'' : A^* \rightarrow (S \rightarrow S)$, a function between two monoids - and that's what a state machine amounts to.

Example 14 Take input alphabet to be $\{'a', 'b'\}$, and the machine should check that we have the same number of $'a$'s and $'b$'s in the input. As a state space we take `Int`, all integers. The traditional representation of 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 on elements of alphabet; now 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]`. But we have a monoid (of endomorphisms), so there is no need to keep a list of functions, we can just fold them together. For folding, since we are in Scala, we use Scala fold.

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

```
def f2(input:String) = (id /: input) (_ andThen f1(_))
```

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

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

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

Commutative Monoids

You have probably noticed that $'+'$ behaves very differently for strings and for numbers; for numbers, $a + b = b + a$; for strings - almost never. So, to be specific, we have to define what we

are talking about:

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

This definition does not involve any monoids and is good for just any binary operation. But if we have 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*" $\neq$ "*world*" + "*hello*"

We do not need to define a special kind of functions for commutative monoids; commutativity is the property of 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 $a1\ Op\ a2 = a2\ Op\ a1$, then $f(a1)\ Op\ f(a2) = f(a2)\ Op\ f(a1)$, so we will have some commutativity in B; but that's it.

How about free monoids, can they be commutative? The short answer is: No. But we can introduce a new kind of free monoids, *free commutative monoid*.

How can we get one? We start with a free monoid (on a type A), that is, a monoid $List[A]$. To make it commutative, we have to introduce a special kind of equality, where two lists differing only in order of elements, are the same. The easiest way, in terms of code, is to have "sorted lists". In Java/Scala world, this monoid is called `SortedList[A]`.

For example, if we have a list of characters: $\backprime abracadabra''$. Forget about the concatenation order; we sort the list, obtaining $\backprime aaaaabbcdrr''$.

We can build it for any type `A`. But note, concatenation will be different from list concatenation: after concatenating two lists we will have to sort the new list.

So, maybe we should not focus on lists, and look for a better representation? All we need is the number of occurrences for each element of the alphabet. This structure reminds a set, but we need to count, how many times each element is there. It is called *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, say, all of the birds, we don't need to know on which day they were appended to the multiset.

Now, what is the binary operation that makes multisets a monoid? It is multiset union; we join two multisets, adding the counts. The empty multiset will be our neutral element.

Do you see that this operation of joining multisets is commutative?

How About Sets?

A regular set is very similar to a multiset, except that we do not count occurrences; an element is either there or not. The main distinction actually 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 following property:

$x\ Op\ x = x$ is called *idempotent*.

In the monoid of sets with 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 $(Bool, \wedge, true)$. We will focus more on logic later on.

Removing constraints

A monoid has two properties: associativity and 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 monoid, we have a *semigroup*. Examples are plenty. First, any monoid is of course a semigroup. Here is a non-trivial one:

Example 16 Take a set of all integers above 17, and addition: $(\{n : Int \mid n > 17\}, +)$. We would have had a monoid if we had 0 here; but we don't; and so all we have is a semigroup.

Notice, we still have associativity in semigroups. But what if we drop associativity too? We get...

Magmas

Magma is just a binary operation. You may think that this does not make much sense; but it actually does, a lot.

Example 17 Binary Tree is magma. Indeed, given two binary trees, we can build a tree out of them by attaching a new root.

Example 18 The operation of forming a pair gives us magma: if we have a proper common type for the arguments of tuple.

Chapter 3. Partial Orders, Graphs, DAGs

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

Partial Order

Partial Order is also known as poset. Partial order means *Partially Ordered Set*. The word "set" should not confuse us; we are not talking about sets; we are programmers, and we deal with type. The politically correct term would still be "partial order".

Definitions

Definition Given a type A, a *partial order* on A is any transitive antisymmetric binary relation on A. The relation is usually denoted as $a < b$: it is the same as saying that the pair (a, b) belongs to this relation.

We see a strict order here; "antisymmetric" means that we, for instance, cannot have $a < a$. This definition introduces certain difficulties if we are not in a Boolean world of double negation. We can bypass this by introducing a better definition: non-strict order, a relation $' \leqslant'$, which is loosely defined as $a \leqslant b \ \equiv \ (a = b) \vee (a < b)$.

Alternative Definition Given a type A, a *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 form of definition is that we don't involve here the rule of excluded third.

Further on we will be using this second definition; the reader just better be aware that opinions on this topic vary.

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

Definition A partial order $(A, \leqslant)$ is called *discrete* if the relation $\leqslant$ only consists of the diagonal: $x \leqslant x$ for all x. (Check out chapter 1 for the role of 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. We can compare c only to itself.

Example 3 Take 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, we need transitivity; from $a \leqslant b$ and $b \leqslant c$ it should follow that $a \leqslant c$. In our case it would be enough, to have $a \leqslant c$, then we would have a legitimate partial order.

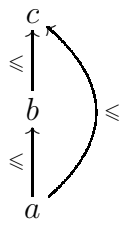


Figure 3: *A partial order with all elements comparable*

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.

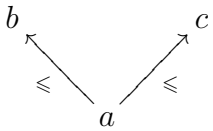


Figure 4: *A partial order where two elements are not comparable*

Example 4 Strings, with alphabetic order (assuming, we know how to compare any two characters; in real life the order of characters depends on a) language, b) year; c) country; d) prevailing

current point of view of local linguists). This example basically shows that an order is not necessarily coming from above, but is added separately, and is not must dependent on the "type of data".

Note also that the order defined here for strings is *partial*. If we require that either $a \leqslant b$ or $b \leqslant a$, such an order is called *total*, or *linear*. In *Set Theory* one of the axioms is equivalent to having a linear order for every set. The axiom 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, which may be not exist for all pairs.

Definition 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 have to always exist. For example, in $\{a, b, c\}$, with order defined as $a \leqslant b$ and $a \leqslant c$, we can see that $a = glb(b, c)$, but there is no $lub(b, c)$.

For example, in $\{a, b, c\}$, if $a \leqslant b$ and $a \leqslant c$, we can see that $a = glb(b, c)$, but there is no $lub(b, c)$.

Caveat In Chapter 2 you saw that for a partial order $(\mathbb{N}, \leqslant)$, the function *lub* gives us a monoid, $(\mathbb{N}, lub, 0)$. But in a general case, neither *glb* nor *lub* necessarily give a monoid on a partial order. First of all, neutral elements are not guaranteed, and more, these operations, *glb* and *lub*, don't even have to be associative. Look at this picture:

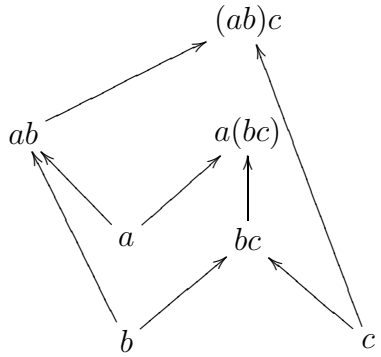


Figure 5: *A partial order where glb is not associative:* $glb(glb(a, b), c) \neq glb(a, glb(b, c))$

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 has to exist for a given partial order. Take natural numbers, $\mathbb{N}$. Bottom exists, it is number 0; but there is no top.

Trees

As a programmer, you are familiar with trees as data structures. Mathematically, these definitions are either not generic enough, or too specific. We will define a tree in a way that may involve continuous versions. For instance, if we take $\mathbb{R}$, with numbers as nodes, there's no "parent node" for a given number.

If we will start, defining a tree, with partial order, we will have to come up with something that does not require a unique parent node. One of the solutions consists of requiring that all nodes above any given node are comparable.

Definition A *tree* is a special kind of partial order. It has a top, and for each x the collection of elements above this x, $\{y \mid y \geqslant x\}$, is a linear order. The top of a tree is called *root*.

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 this 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.

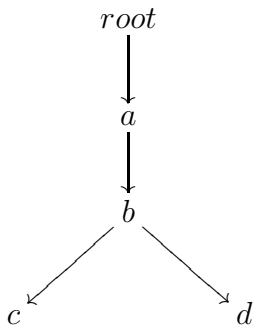


Figure 7: *A tree with four nodes*

Example 6

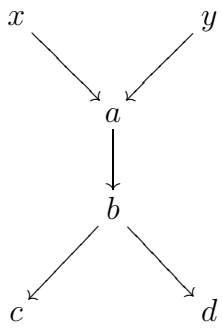


Figure 8: *Not a tree*

Counterexample 7 In this case neither x nor y is a root.

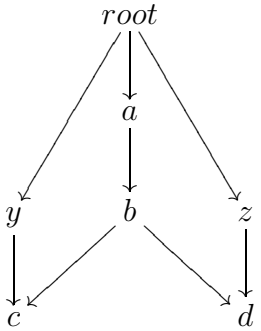


Figure 9: *Not a tree*

Counterexample 8 In this case c has incomparable ancestors, y and b. So $\{x|x > a\}$ is not a linear order.

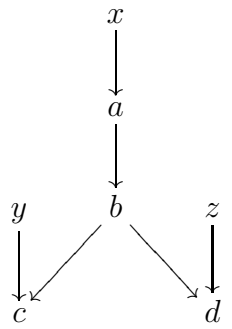


Figure 10: *Not a tree*

Counterexample 9 In this case c has incomparable ancestors, y and b; and x is not a root either.

Functions of Partial Orders

Definition 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? It does. 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 includes a segment of natural numbers into another, mapping numbers to themselves. f preserves the order, but it preserves neither top nor bottom. Top of $[2, 4]$ is 4, and bottom of $[2, 4]$ is 2; neither is a top or a bottom in $[0, 5]$.

Remember that 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 now you are 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. But database people will tell you "no way, 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 such a representation of relation: kind of relation, timestamps, and the like. It is not a binary relation anymore. All this turns our partial order into something different; we will get back to it some time later.

Epi, Mono, Isomorphisms As usual, we can define a monomorphism, an epimorphism, an isomorphism for functions between partial orders. Mono and epi are defined as before, on 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 relation, $a \leqslant b$. And take another partial order, $Y = (\{a, b\}, \{\})$. It has the same two elements that are not comparable. You can see that 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 $a \leqslant b$; but it is not so.

Graphs

Everybody in software industry knows what a graph is; however, definitions vary. Here are some of the definitions; we will choose one, but have to be aware of existence of others. We'll walk through three such definitions.

Undirected Graphs

Definition 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 (named *edge*) is between two nodes. For every pair of node there is at most one connection.

For each collections 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 dates from year 1878; but the whole idea is much older. While here we are not going deep into the study of such graphs, we will list some interesting problems related to such graphs.

Eulerian Path Leonard Euler was a great German Mathematician born in Königsberg, Prussia; and he found this problem interesting: there are 7 bridges in Königsberg; can one take a walk crossing each bridge exactly once?

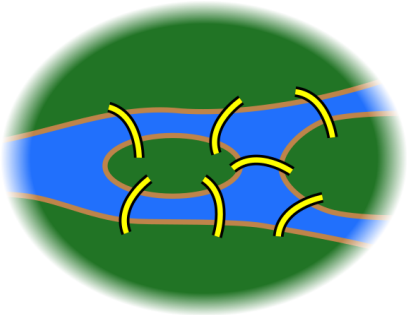


Figure 11: *Königsberg*

Before working on that problem, look at a simpler one: draw an envelope without lifting pen. The solution is below:

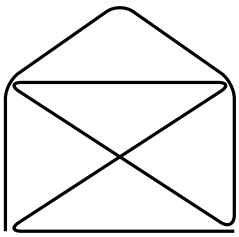


Figure 12: *Königsberg*

Drawing envelope without lifting pen.

Why this solution? Notice, if a node has 1 edge, we have to either start or finish there; if it has two edges, we can just draw through; and if we add two more edges, we can do it again. 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. Like in this envelope picture.

But in Königsberg we are out of luck: every node (every island) has an odd number of edges (bridges). Our path cannot have more than two ends, though. So, no solution. That's of course assuming that the Earth is flat and there is no way to bypass river Pregolya.

Chinese Postman Problem A postman has to deliver mail to all nodes on a graph, and for some reason he (or she) has to 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; but if not (say, this is a Königsberg postman), something should be done. We ascribe length to each edge; and the problems turns into "find the shortest path" problem.

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

Planar Graphs Let's start with a practical problem. We have 3 cottages standing nearby, and 3 utility companies (electricity, gas, water); is there a way we can provide electricity, gas and water to these cottages so that none of the lines crosses another? Here is the picture (this specific graph is called $K_{3,3}$):

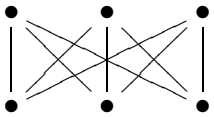


Figure 13: $K_{3,3}$

It turns out that no, it is impossible, unless we go to other dimensions (3 is enough, and electric wires can go in the air.

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

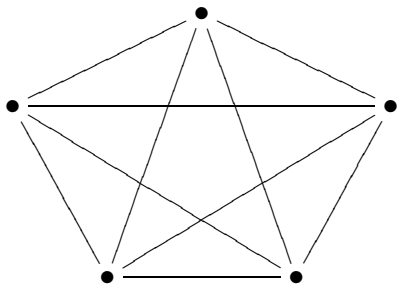


Figure 14: K_5

But how about 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 them, the problem cannot be solved (obviously); but if it does not, the graph is planar.

Directed Graphs

Definition 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 $(A, \leqslant)$; 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 Directed graph can have cycles. Note that in this case we do not have partial order:

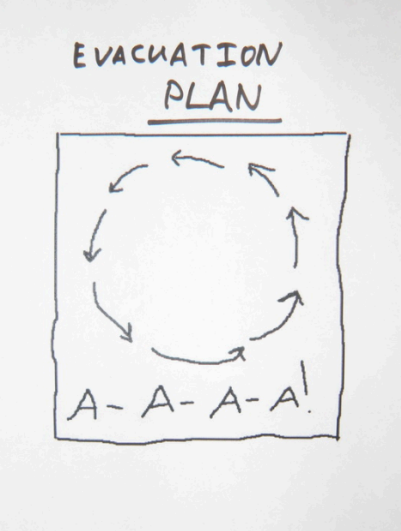


Figure 15: *Directed graph with cycles*

Example 15 A directory hierarchy in a computer is an oriented 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 we have links in our file system, we can have any graph structure. In any case, we don't generally have a tree.

Definition *DAG*, *directed acyclic graph*, is a directed graph that has no cycles. We probably had to define cycle in a graph, but the reader can guess what it is.

DAGs are a popular tool for representing a variety of programming ideas; you can find discussions of problems with DAG on stackoverflow, and a tool for drawing DAGs on dagitty.com.

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

Example 16 Java Memory Model has a well-defined "happened-before" relation. It is a partial order, and directed graph.

Example 17 Git is (one of the) best version control systems, as of 2014; and it uses DAGs to represent the life of objects (mostly we are talking about files). The history of a file may branch, merge, but never go back.

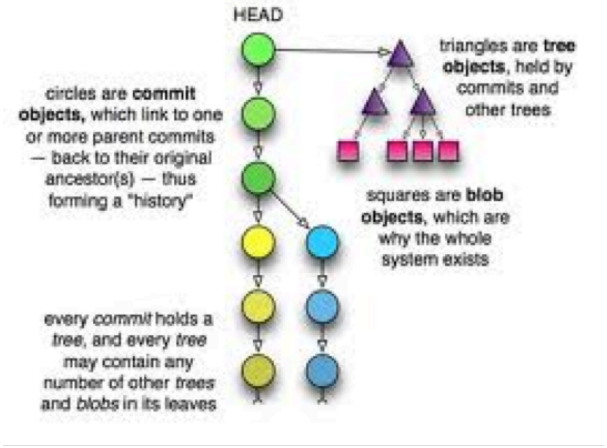


Figure 16: *Git version tree*

Directed Multigraphs

Definition 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, machines.

Example 18 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 symbols of input alphabet. In UML it is called "State Machine Diagram":

A directed graph is a graph ("directed multigraph") too, with not more than one edge from node to node. Hence, a partial order is a special case of a graph.

If you remember, 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. But since, in graphs, we can have more than one node, we can imagine, given a collection of types, e.g. $\{Int, String, Char, Bool, Double\}$, a graph that has these types as nodes, and all functions between these types as edges.

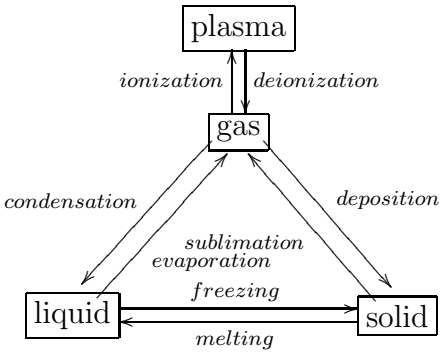


Figure 17: *Matter state machine diagram*

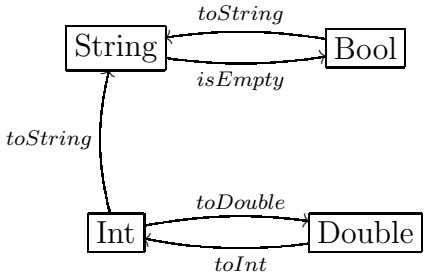


Figure 18: *Sample graph of Scala types*

Chapter 4. Boolean Logic

"Logic" is an immense area of human knowledge; I have no plans to even mention all the aspects, and focus here only on simple notions and language, reasonable enough to be used in our daily practice as software developers. Lawyers or philosophers or physicists or linguists will probably need a very different kind of logic; we just won't even look in those directions.

The Language of First Order Logic

Before talking about logic, even vaguely, we introduce a language that we will use throughout this text. It is neither loose nor strict, but somewhere in the middle.

Definition 1 A first-order language consists of the following:

- *Names*, which are supposed to denote objects in some kind of universe (like "John" denotes a certain human being in a small world where it does it uniquely, or 10 denotes a number, if we know exactly which numbers we are talking about);
- *Functions*, which are used to form terms out of names and other terms; a function can have *infix notation*, like $2 + 3$; a function takes its arguments and has a result value;

- *Terms*, which are expressions built out of names and functions;
- *n-ary relations*, which denote the fact that some terms have certain properties see examples below);
- *formulas*, which are expressions built from n-ary relations applied to terms.

We could express all this formally, in Backus Normal Form, but probably there is no need.

Summing up, terms are either built from names and functions, using parentheses, or are formulas that use terms; and that's it so far.

Example 1

- $sin(ln(2.718285)) < cos(exp(0.001))$
- $phoneNumber(John) =$" 3141592654"
- $Alaska$

Note that "first order language" is not something universal; on the contrary, you can just come up with a bunch of names, functions, relations: and voilà, you have a language.

Example 2 Take integer numbers, arithmetic operations on them, comparisons and equality relations.

Example 3 Planar geometry. We will need numbers, 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 center at P and radius R
- $center(C)$ – this term denotes a point that is 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 circleC_1
- $isBetween(P_1, P_2, P_3)$ – points P_1, P_2,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. We have nodes and edges, and the only relation:

$isBetween(Node_1, Node_2, Edge_3)$.

Logical Operations

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

Negation

Having 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

Having 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

Having 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

Out of atomic formulas we build *sentences*, using negation, conjunction and disjunction; to avoid ambiguity, we also use parentheses.

Example 6 $((x < 7) \vee ((y < x) \wedge \neg isEmpty(''helloworld'')))$

Properties of Operations

Describing these properties, we use symbol $\equiv$, which vaguely means that the expressions on the left and on the right are equivalent. The exact meaning of such equivalence 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 saw negation, conjunction and disjunction; can we have other operations?

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

We can start with zero arity, that is, with *constants*, and remember the two constants that were 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 we will need both. 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 just 4: here is the table.

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

For binary operations there is more combinations (any idea how many?); we will not list them all here, 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

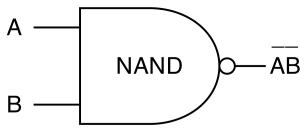
You see conjunction and disjunction; implication can be defined as $\neg x \vee y$; equivalence can be defined as $(x \wedge y) \vee (\neg x \wedge \neg y)$.

Actually, any formula has a standard representation via negation, conjunction and disjunction. More, disjunction can be expressed via negation and conjunction: $x \vee y \equiv \neg(\neg x \wedge \neg y)$. So it's enough to have just negation and conjunction; the rest follows.

Or we could have negation and disjunction, and express conjunction as $x \wedge y \equiv \neg(\neg x \vee \neg y)$. Again, two operations is enough.

Can we go further? Yes. There are two operations, each of which can be used to represent all other operations. One of them is called *Peirce Arrow*, or *NOR* (because it is equivalent to $\neg(x \vee y)$); the other is called *Sheffer Stroke*, or *NAND* (because it is equivalent to $\neg(x \wedge y)$).

Imagine you have to use just one circuit to build a logical schema; which one would you use? You have two choices, either use NAND, or NOR.



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

Figure 19: *NAND*

Proving Something

Premises and Conclusions

When we have first-order language, we can connect our formulas (see definition), calling some of them *premises* and some – *conclusions*. Informally, you list premises, and then come up with a conclusion, or go in the opposite direction: conclusions, because we have premise1, premise2, etc. Nobody said we have to do it right.

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

Some may agree with this, 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 men are mortal". We may argue that, judging by the name, Pavlova must be a female; I will also add that Pavlova is a cat. The fact that she is dead by now does not make her a man.

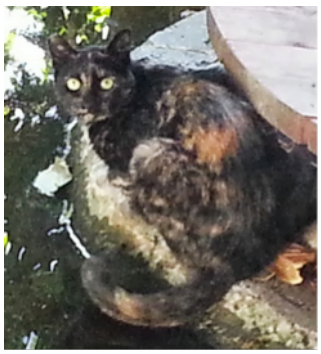


Figure 20: *Pavlova, a cat*

Argument

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

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

Valid and Sound Arguments

As you see from examples above, some conclusions make sense, some don't; also some premises make sense, and some don't. Let's disambiguate these situations.

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

Note that we have just introduced the word "true" in our discourse. This is a little bit unusual; but we are not quite formal here.

You can check by yourself whether each of the examples above is valid or sound or neither or both.

Premises are called *inconsistent* if they contradict each other, that is, you can deduce ⊥ (that is, *false*) out of them. Remember, in this case the argument is not sound (wrong 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 applying *rules*; we will walk through such rules, without questioning them (questioning them means proving theorems, and it 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; this latter shape will be explained later.

Elimination Rule Also known as *Indiscernibility of Identicals*, *Substitution Principle*, and *Identity Elimination*.

The rule is this:

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

For example,

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

Introduction Rule Also known as *Reflexivity of Identity.*

The rule is this:

$$P \vdash x = x$$

Have you noticed that we are using variables already? Strictly speaking, we write this rule, but we mean anything can be substituting x.

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

Neat, right? It seems like the properties come out of nothing. Just out of substitution, which turns out to be a very powerful rule. We will add more rules now.

Negation Elimination

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

Informally, it means that if we have "non not P", we can say that P holds. This may sound obvious, but look at it like this. We could not prove P. We only could prove that assuming "not P" leads to contradiction. Does it give us P? Probably not, generally speaking. E.g. Scott Peterson could not prove he did nod do it, and hence was found guilty. That's what they do in classical logic.

Conjunction Elimination

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

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

Disjunction Elimination

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

The meaning of this 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 as 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. Good for proving existence of supernatural creatures and entities, as well as their non-existence.

Proof by Contradiction The trick 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.

Assume it is rational, $\sqrt{2} = p/q$, where p and q are natural numbers, mutually prime (have 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*something. But if 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, and so p and q do have a common divisor (=2), oops. Contradiction! Our assumption was wrong.

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

This can be extended to a list of $P_1, P_2, ..., P_n$ and 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 the case when d is not rational, we are free to take $d^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}*\sqrt{2}} = \sqrt{2}^2 = 2$, and it is rational, produced from two irrationals.

See, we have no clue here, and we don't even care if d is rational or not. Rational or not, we prove our point.

Simplification Expressions

When we have a long expression, involving our three operations, $\wedge, \vee, \neg$, it may be hard to understand its meaning. E.g. even this one: $(A \vee B) \wedge C \wedge (\neg(\neg B \wedge \neg A) \vee B)$ may confuse a regular human. Fortunately, our rules above help us in refactoring such expressions into a standard form, which is usually simpler.

Negative Normal Form

The first such refactoring consists of pushing negation as deep as possible, right in front of atomic formulas (that is, variable names). The result of such normalization is called *Negative Normal Form*, aka *NNF*.

Every time we have two negations, we remove them, by Double Negation rule. Every time we have a negation before a disjunction or a conjunction, we can apply a De Morgan Law. This way we can do, for example, the following transformation:

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

Disjunctive Normal Form

After applying Negative Normalization, we have 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 disjunctiion: $(A \vee B) \wedge (C \vee D) \equiv (A \wedge C) \vee (A \wedge D) \vee (B \wedge C) \vee (B \wedge D)$. This way any complex expression be 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)
$(\neg A \vee (B \wedge C)) \ \wedge \ (B \vee (A \wedge C)) \equiv$ (distibuting over disjunction)
$(\neg A \wedge (B \vee (A \wedge C))) \ \vee \ (B \wedge C \wedge (B \vee (A \wedge C))) \equiv$ (distibuting 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 part of three chapters dedicated to logic; next we will see how we can do without Booleanness (that is, without the rule of double negation)…; then we cover quantifiers.

Chapter 5. Logic Does Not Have To Be Boolean

What Non-Booleanness Means

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

For example, for decades mathematicians were looking for an answer whether there is an intermediate set size between countable and continuum; and then it turned out that if we assume there is such a size, then it exists; if we assume there is none, there is none; it is not a theorem, it is an axiom. And an axiom is not something that is always true; it is something that introduces a constraint in a theory. If we could prove that an axiom is true, we would not have needed it.

We may have more than two logical constants, but it does not make our logic non-boolean. Double negation rule may still hold.

Example 1. Boolean logic, but not 2-valued Take bytes and their operations: bitwise conjunction, bitwise disjunction, and negation. We have 256 different values; but we know that double negation is identity. So this 256-valued logic is Boolean.

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

Dropping Booleanness

We want to remove the rule saying that either P or $\neg P$ holds; formally, $\vdash P \vee \neg P$. Removing the rule means that we will not rely on it anymore, although it may still hold 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 have to define it differently. And here is the solution:

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

Informally, saying that R can be deduced from a conjunction $P \wedge Q$ is the same as saying that we can deduce the implication $Q \rightarrow R$ from P. Imagine Q is $\top$. Then, from the formula above, we see that $Q \rightarrow R$ is the same as R. On the other hand, if Q is closer to the bottom, the

situation may change. Say, if Q is "below" R, the implication $Q \to R$ is true. We can say that implication defined like this shows the level at which R depends on Q.

So, we were able to define implication via conjunction, and we do not need negation for it.

Also, does not it remind you currying? Given a function $f(P, Q) : R$, we transform it to a function $f_c(P) : Q \to R$.

Now that we have implication defined, we can define negation as $\neg P = P \to \bot$. It has properties that make it very similar to classical negation, but double negation law does not generally hold anymore. In the table below are the properties that still hold when implication is defined like above:

Statement	**Its 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 not as strong as 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 does not hold, then neither of P and Q holds
$\neg P \land \neg Q \vdash \neg(P \lor Q)$	If neither P nor Q holds, their disjunction does not hold either

But some statements do not hold in this kind of logic; look at the following table:

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, it does not mean one of them is always wrong

Example 2. Three logical values ("ternary logic") Take three logical values, $\top$, $\bot$, ?; and define operations on them:

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

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 that disjunction and conjunction are associative, and that distribution laws hold for them.

This is the simplest intuitionistic logic, and it is a good tool to test our statements.

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

We will have a closer look at the idea of "between" later on; now another example.

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:

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

- $a \vee b = lub(a, b)$

Implication in this example is defined, as always, via the equivalence: $(x \wedge y) \leqslant z \equiv x \leqslant (y \to z)$, that is, in our case, for any $y \leqslant z$, $y \to z$ is 1(aka $\top$). This is because $x \wedge y \leqslant z)$, and for any $y > z$, $y \to z$ is z(because in this case $(x \wedge y \leqslant z) \equiv (x \leqslant z)$.

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

We Have a Partial Order

When we take all available logical values, we can introduce a partial order on them, by the following rule: $p \leqslant q \equiv ((p \wedge q) = p)$. Conjunction here plays the role of $glb(p, q)$. Why is it a partial order? We need antisymmetry and transitivity.

Antisymmetry is easy; we need $p \leqslant q, q \leqslant p \vdash p = q$. Why is it so? 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 we see the partial order for the ternary logic we discussed above.

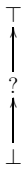


Figure 21: *Partial order of 3-valued logic*

Finding Good Partial orders

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

You see that there is no unique closest elements below both d and e: neither a nor b nor c satisfy the obvious requirements of *glb* being unique and being the closest possible. We need *glb* and *lub*, since they are our conjunction and disjunction operations.

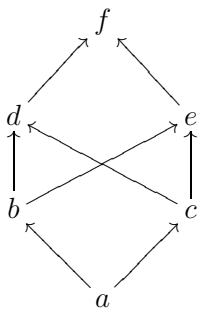


Figure 22: *Partial order with no glb/lub*

Definition A partial order where every two elements have *glb* and *lub* defined is called *lattice*.

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

But that's not enough: we also need $\top$ and $\bot$, neutral elements for conjunction and disjunction, that is, from partial order point of view, *top* and *bottom* elements.

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

We are almost there; the only thing is, we need to have implication $a \to b$ operation to be defined for all a, b. And this is what is called Heyting Algebra:

Definition 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 *Heyting Algebra*.

This was the last property required for intuitionistic logic; and you already saw that a Boolean logic is a special case of intuitionistic one.

Example 4 What if we take ternary logic from the example above and add one more intermediate value, like in the picture:

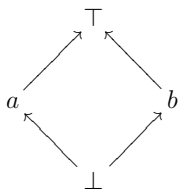


Figure 23: *Sample Heyting algebra that is actually Boolean*

By our definition of implication, for any x saying that $x \leqslant (a \to b)$ is equivalent to saying that $x \wedge a \leqslant b$. We have only two values with the property $x \leqslant b$: these are b and $\bot$; only one of

them, $\bot$, can be represented as $x \wedge a$; so having $x \leqslant (a \to b)$ is equivalent to having $x \wedge a = \bot$. The biggest such element is b.

Similarly, by definition, for any x saying that $x \leqslant (a \to \bot)$ is equivalent to saying that $x \wedge a \leqslant \bot$. We have just one value y with the property $y \leqslant \bot$: it is $y = \bot$; so having $x \leqslant (a \to \bot)$ is equivalent to having $x \wedge a = \bot$. The biggest such element is b.

Have you noticed? $b = \neg a$; and similarly, $a = \neg b$. We have a Boolean logic here. Of course its size is 4; it must be a power of 2.

Example 5

Remember, we saw this rule: $\neg(P \wedge Q) \vdash \neg P \vee \neg Q$ that does not generally hold. Can we demonstrate it on a partial order?

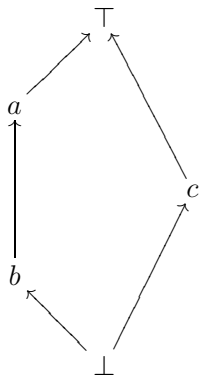


Figure 24: *Sample Heyting algebra*

In this lattice, we see that $a \wedge c = b \wedge c = \bot$ and 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 $\top \vdash c$ does not hold, we have our example… or is it rather a counterexample?

More Operations?

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

Proof in Intuitionistic Logic

Informally, a proof in intuitionistic logic consists of providing "an evidence". But we have all the rules described in the previous part; and out of those rules, only one is not used in intuitionism; it is Negation Elimination rule, $\neg\neg P \vee P$, and it does not apply.

Chapter 6. Quantifiers

What Are They

You are already familiar with predicates. A predicate is a judgment about an element of a collection; whether its value is true or false (or something else), depends on the element. Quantifiers are used to express judgments about the whole collection of elements.

Example 1 "All men are mortal". Here we say nothing about any particular man (and we say nothing about women, or any other possible kind of humans), but what we say applies to any man, due to the word "all".

So from here we normally 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". Here we are talking about men in general again; but we cannot say for sure whether this or that man is or ever will be mugged in the streets of New York; this definitely does not apply to those who are not in New York. What's more important, while this statement may be generally correct, we may never manage to find such a man for an interview.

These two examples mention the two quantifiers we are going to talk about, existential and universal.

Universal Quantifier

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

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

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

In Java/Scala

In Scala, having a `c:Collection` (actually, `c:TraversableLike`) 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`.

In Java before version 8 there was no such thing; so, if you are still with older versions, you can start using Bolts library, FunctionalJ, or Google Guava library; each of these can handle quantifiers. Starting with Java 8, you can use `Stream` and its method `allMatch`.

Existential Quantifier

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

We write it as $\exists x\ P(x)$; it is pronounced as "there exists an x, such that $P(x)$"

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

In Java/Scala

In Scala, having a collection `c` (actually, `TraversableLike`) 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`.

In Java before version 8 you can use Bolts, FunctionalJ, or Google Guava to provide similar functionality. With Java 8 you can use `anyMatch`.

The Case of Boolean Logic

Quantifiers and Logical Connectives

The predicate you have 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, an with quantifiers inside. Interesting, 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 maybe a continuum of values and trying to figure out whether we can ever

prove anything, we take just two values in the range. If a statement does not hold for a choice of two, it won't hold for a bigger collection, in general settings.

Also, to be on the safe side, we have to check if our statement holds on an empty collection; we will see further why so.

Conjunction and Existential Quantifier Can we check how $\exists x\ (P(x) \wedge Q(x))$ and $(\exists x\ P(x)) \wedge (\exists x\ Q(x))$ are related?

Suppose we have just 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 (see the end of chapter 4), we will have

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

You see that the second statement follows from the first, but not the first 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 holds, we found an x such that both $P(x)$ and $Q(x)$ hold; 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 $(\forall x\ P(x)) \vee (\forall x\ Q(x))$.

If, again, we have a range 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 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, deducing $\forall x\ (P(x) \vee Q(x))$ from each one, come to the same conclusion.

Negation and Universal Quantifier What happens if we write $\forall x\ (\neg P(x))$? Is it related to $\forall x\ (P(x))$?

Let's start with the case of x_0 and x_1. The statement is equivalent to $\neg\ P(x_0)\ \wedge\ \neg\ P(x_1)$, which is the same as $\neg\ (P(x_0)\ \vee\ P(x_1))$, that is, $\neg\ (\exists\ x\ P(x))$.

The same holds for an arbitrary amount of values. In plain English it can be rephrased as "The statement 'for all x, $\neg P(x)$' holds if and only if there is no such x that $P(x)$".

Negation and Existential Quantifier Now what about $\exists x(\neg P(x))$?

Again, take the case of just two elements, x_0 and x_1. In this case, the statement 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 there is an x that does not satisfy P, then we know that not every x satisfies P. But what if we go in the opposite direction, and from a general fact that not every x satisfies P try to deduce the existence of such an x?

Here is an example. Let's start enumerating sets of integers. Of course some of them can be easily enumerated. But definitely not all of them, right? So there must exist a set of integers that cannot be enumerated. Which one is it? (Suppose we found one, let's call it XXX; we then can introduce another enumeration, starting with this XXX, thus beating the assumption.)

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 logic usually does not make any such assumptions.

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 $\exists y \exists x P(x, y)$.

But how about $\forall x\ \exists y\ P(x, y)$ vs $\exists y\ \forall x\ P(x, y)$? Let's show an informal example. Say, $P(x, y)$ stands for "x is a Facebook friend of y". The first one, $\forall x\ \exists y\ P(x, y)$, 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 is probably the invisible friend, or rather a Big Brother.

When Our Logic Is Non-Boolean (Intuitionistic)

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

The rest of the features are the same as in Boolean logic; so we will look closer into how quantifiers interact with implication and negation. Note that implication is a binary operation, and the two parameters, P and Q, in $P \to Q$, usually behave very differently.

Implication and Universal Quantifier

So, what do we say about the following three cases?

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

Informally, assume again that we have just two values, x_0 and x_1. Then our three cases turn into these:

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. $(P(x_0) \to Q(x_0)) \wedge (P(x_1) \to Q(x_1))$

Let's cover them one by one.

The case of $\forall x\,(P \to Q(x))$ This case is pretty obvious; it is equivalent to $P \to (Q(x_0) \wedge Q(x_1))$, which gives us a hint that, most probably, $\forall x\ (P \to Q(x))$ is the same as $P \to (\forall x\ Q(x))$.

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

The case of $\forall x\ (P(x) \to Q)$ This case, if we have just two values in the universe, can be transformed into $(P(x_0) \vee P(x_1)) \to Q$.

The argument does not look as trivial as case 1. The superficial explanation is this. Suppose Q follows both from $P(x_0)$ and from $P(x_1)$; then, having $P(x_0) \vee P(x_1)$, whichever of these two holds, we have Q.

Proving it in the opposite direction is trickier. Assume we have $(P(x_0) \lor P(x_1)) \to Q$, and assume $P(x_0)$ holds. Then $P(x_0) \lor P(x_1)$ holds, and so Q holds. The same way we deduce Q from $P(x_1)$; so $(P(x_0) \to Q) \land (P(x_1) \to Q)$ is true.

Now we have a strong suspicion (or hope) that $\forall x\ (P(x) \to Q)$ is equivalent to $(\exists x\ (P(x)) \to Q$.

While the rule works in a populated universe, now suppose we have no values (our universe is empty); so there are no x. The statement $\forall x\ (P(x) \to Q)$ then holds, whatever be our P and Q. How about $(\exists x\ P(x)) \to Q$? Oh, this is always true: since there is no x, $\exists x\ P(x)$ is false, and any Q follows from it. So we are good in an empty universe.

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

The case of $\forall x\ (P(x) \to Q(x))$ This third case is not equivalent with $(\forall x\ (P(x)) \to (\forall x\ Q(x))$, but still we have some connections. First, if we have $\forall x\ (P(x) \to Q(x))$, and $\forall x\ P(x)$ holds, then $\forall x\ Q(x)$ also holds. You can check it by using our two-valued example. Also, if we have $\forall x\ (P(x) \to Q(x))$, and $\exists x\ P(x)$ holds, then $\exists x Q(x)$ also holds. This is more or less obvious; and you can check it using the two-valued example.

Summing Up

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

Implication and Existential Quantifier

Similarly to the previous part, consider three cases:

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

In the domain with just two values, it will look like this:

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

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

For case 1, we can actually demonstrate that $\exists x\ (P \to Q(x)) \vdash P \to (\exists x\ Q(x))$ right away. Assume $\exists x\ (P \to Q(x))$, and that P holds. 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 it does not work in the opposite direction? it is 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 it give us? Nothing; say, if we have $P(x_0) \to Q)$ and $P(x_1)$ – so what? It is not enough. On the other hand, if we have $\exists x(P(x) \to Q)$ and $\forall x P(x)$, then Q is true.

So we have this: $\exists x(P(x) \to Q) \vdash (\forall x P(x)) \to Q$

And as to case 3, it gives nothing; if for some x we have $P(x) \to Q(x)$, so what? Having one such x, without knowing if $P(x)$ holds, gives us nothing.

Summing Up

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

Negation and Quantifiers in Intuitionistic Logic

In intuitionistic logic, $\neg P$ is the same as $P \to \bot$; so all that we talked about in the previous two sections applies, but not more. 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 see, we have no rules that would magically state existence of an entity without an explicit demonstration how to reach that entity.

Chapter 7. Models and Theories

In this chapter we discuss theories, models, and relations between them. This is a very 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 very 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 to define theories in terms of models. Here are examples. People talk about geometry, and suddenly you hear about "a set of all points such that..." But wait, 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.

Or, one defines a monoid as a set of elements... wait, can we define a monoid without *a* Set Theory? More; if everything is defined using sets, how then can we define a Set Theory without basing it on a Set Theory? Something wrong with this approach. A monoid based on a set is actually a model of a monoid, a model in the Set Theory (where the underlying set belongs).

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

Theories

Definition 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: 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 this provides us with a "first order language" (see Chapter 4). Now in addition to a language, a theory also may have *axioms*, which are formulas in the given language. In the case 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 in a theory use identifiers; each identifier has a type. The fact that an identifier x has type T is denoted like this: $x : T$.

You are already familiar with some examples of theories, but it makes sense to rephrase them in this new aspect.

Example 1. Monoid. A monoid theory (T, Op, Z) has just 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**
  $\forall a, b,\ lt(a, b)\ \wedge\ lt(b, a) \rightarrow a = b$
- **transitivity**
  $\forall a, b, c,\ lt(a, b)\ \wedge\ lt(b, c) \rightarrow lt(a, c)$

Example 3. Euclidean Geometry (a segment). There's a large variety of definitions of Euclidean Geometry as a theory; here we introduce a definition of a small portion of it, without going deep into tricky details.

Euclidean Geometry has the following types:

- P – points
- L – lines
- C – circles

It has 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 not every two points give a unique line; if the two points are equal, they don't. We could ask then, where do the lines come from, if there's no function to build a line? The answer is, lines and points are independent from each other. Lines and points 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**
  $\forall p : P, a : L,\ \neg isOn(p, a) \rightarrow \exists b : L\ isOn(p, b) \wedge \forall q : P,\ isOn(q, b) \rightarrow \neg isOn(q, a)$

There are many more axioms; building a full geometry theory is not a goal here.

We saw in two examples already that we have to define equality, `=`, and specify its properties in axioms. Instead we can refactor our examples a little bit, and extract the general idea:

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

Example 4. Peano Arithmetic *Peano Arithmetic* is an equational theory that has just one type, and two operations, one nullary, 0, and one unary, S (called "next" or "succ"). Axioms 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 (in addition to `=`) relational symbol, $\in$. It has over a dozen axioms; here are some:

- **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 we add one more axiom to the one we had. 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
- $\forall a, b, c, d\ a \neq b \wedge a \neq c \wedge b \neq c \rightarrow d = a \vee d = b \vee d = c$ – no fourth value

Dealing with Theories

When we define a theory, we only define types, functions, and relations. If we don't provide any axioms at all, our theory is a free theory. If we have axioms, on the other hand, we can try, using first-order logic deduction rules, to deduce more propositions out of the existing axioms. These new propositions are called *theorems*. A theorem is never something given to us magically from above, but just a consequence of axioms of a certain theory; a theorem belongs to that theory and makes no sense outside the theory. If you see a theorem, and you are told that the theorem is always true, you should immediately get suspicious. If a theorem is a property of a certain theory, will it still hold 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 monoid's axioms and the properties of equality relation.

You've noticed above that in some cases we can take a theory and add one more axiom. In other cases it seems impossible.

Definition A theory is called *complete* if every statement in terms of this theory can be provedF to be either true or false.

One may expect that most popular theories are complete. We study number theory, we study geometry, and we expect that any statement in this theory can be either proved or disproved. It can be added as an axiom, though.

We will start with examples.

The Example 7 from the previous section, where we define "a theory of a three-element set" is complete. Mostly it is because we don't have much to talk about: no functions, no relations except equality. But the theory from Example 6 is of course not complete. It says we have at least three values; and there may be four. Or five. Or three; so that the statement saying that 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 it that any theorem of geometry can be proved or disproved. Unfortunately, it's not so. Our expectations that a certain theory is complete, may lead to pretty wrong conclusions.

Example 9. Peano Arithmetic is not Complete Having defined natural numbers via 0 and *next*, we can easily define addition and multiplication in this theory; with some efforts we can introduce negative and rational numbers. But 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, finding 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, it is called Goodstein's theorem.

Goodstein's Theorem Take 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. Repeating this process, we eventually come up with a representation of our number n as the sum of towers (powers of powers...) of 2. As an 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, and then subtract 1 (and then represent the result again as the sum of towers, this time of 3). E.g., for $2^{2^{2^1}+1} + 2^{2^1+1} + 2^1$ we will produce $3^{3^{3^1}+1} + 3^{3^1+1} + 2 = 22876792455044$.

Repeating this operation (replacing 3 with 4, etc) will, as it was proved by the theorem, eventually produce 0. But the intermediate numbers are going to be huge.

This theorem, that the sequence eventually produces 0, can be proved, but not in Peano Arithmetic. So the question is, if the theorem is formulated in Peano arithmetic, but cannot be proved there, how do we know it is true? We don't. The proof belongs to a so-called second-order arithmetic, which involves not only numbers, but also predicates. Or, alternatively, we can talk about numbers and "sets of" numbers. Of course no "actual" sets are involved; it's just predicates that we are talking about. Second-order arithmetic is a very powerful theory, but it is just a theory. By tweaking it in the right places, we can get a theory where Goodstein's Theorem is not true. This reminds Robert Sheckley's novel "Mindswap", except that in the novel it is the reality that varies; here we only vary theories, the products of our mind.

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 probably itself); but since it contains Peano Arithmetic, it is also incomplete. A popular example is having (or not having) a set of the size that is bigger than countable and smaller than continuum.

One would expect the answer to this question to be definite, either/or. A set either exists or not. That's what one would expect if we were dealing with a "reality". Set theory is not a reality of any kind; it is just a theory. And it was discovered in particular that there's no answer to this question.

The existence of such a set is just another axiom; the negation of the existence of such a set is an axiom as good as the one that states its existence. The issue is known under the name of Continuum Hypothesis.

What's interesting, an ISO standard (ISO/IEC 13568:2002) specifies a version of Set Theory as the so-called *Z-notation*. Whether, according to ISO, the intermediate set exists or not, it is hard to tell; this axiom is not included in the definition of *Z-notation*. Sinse *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 Theories

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

Definition. Algebraic Theory An *(finitary) algebraic theory* is an equational theory that consists of *types*, *functions*, and *equations*. A function has a name and a signature: the list of "argument types" and the "result type". 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 we have a binary relation '=', but that's the only relation, and it is only used in axioms that are equations.

Example 11. Algebraic Theories. If we return to the examples we saw before, we can easily classify the theories into Algebraic and not. First, *Monoids* is definitely an algebraic theory, it has a couple of functional symbols (of arity 0 and 2), and two axioms connecting them. *Partial Order*, on the other hand, is definitely not an algebraic theory: we have a binary relation, not a function; we could throw in a *Bool* type for the results of comparison, but it would not save us from the axioms of *Partial Order* that 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 "three or more" theory into an algebraic form by introducing three nullary operations, giving us a, b, c.

This won't help us though, since the axiom of them 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. Looking back at the definition, we see that an algebraic theory is just a special case of a geometric theory. Some theories are not algebraic, and to disambiguate, we mention the fact that it is a geometric theory. For example, Euclidean Geometry is geometric, and Set Theory is geometric, and Peano Arithmetic is geometric.

Models

We define models of theories in this section. The definition is not universal, we only limit ourselves with modeling 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 let's start with a simple case.

Definition A *model* of an algebraic theory A in a Set Theory $\mathcal{S}ets$ 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.

If we have a model, all the theorems of the underlying theory are valid in this model. But the opposite is not necessarily true - that if for all models in sets some property holds, then it is true as a theorem in the theory. It is not necessarily so; see, for example, Goodstein Theorem.

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 monoid in $\mathcal{S}ets$.

Note, any such structure is a model of 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 monoid.

Note also that the theorem about the uniqueness of neutral element holds in any model of monoid.

Definition A *model* of a (geometric) theory A in a Set Theory $\mathcal{S}ets$ 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 3-element Set. To provide a model of this theory we will need any set consisting of exactly 3 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; e.g. von Neuman model of Set Theory is a good example. Anything that can be modeled in a Set Theory, can be modeled in that theory: modeling is transitive.

Conclusion

This chapter only touched the surface of Theories and Models. There are more than two approaches to all this; the one here is closer to Lawvere's categorical approach. We could not go deeper here because doing it would require the knowledge of topos theory. Topos theory is based on category theory, and so there is a couple of steps before we can study model theory in full depth.

Chapter 8. Category: a Multi-tiered Monoid

Monoid of Functions

Of all monoids, probably the most interesting is the monoid of functions $X \to X$, where the binary operation is defined as composition of functions, and the identity, id_X takes the role of the 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 only a certain kind of functions;

then we 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 has to be included too.

Actually, any monoid can be represented as a monoid of functions; you've probably guessed how. Take a monoid (T, Z, Op) ; and use T as the domain of functions. Now, how do we represent elements of T as functions on T? By multiplication! Every element t of T turns into a function, $f(t)$, that maps every other element s to $f(t)(s) = t\ Op\ s$. Obviously, 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 it is easy to check that it does.

Of course not every function defined on the collection of monoid values comes from the monoid. Look at this example:

Example 1 Take natural numbers (zero included) with addition; they are a monoid. Any natural number n gives us a function $x \mapsto n+x$; but not every function on natural numbers is an addition of a constant.

More Than One Domain

Let's throw in one more object, Y; and have functions of four kinds: $X \to X$, $X \to Y$, $Y \to X$, and $Y \to Y$. We are not in a monoid anymore; not every function can be composed with every other function, so $`Op`$ is now a partial operation. More, we have two distinct neutral elements: id_X, and id_Y.

The picture does not get more complicated if we add more domains and functions between them.

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 either.

So, now you got the idea, and let's elaborate. We need functions and a collection of objects that serve as domains/codomains for functions. There can be no functions (except identities) or no objects (hence no functions at all), or just one object, in which case we have a monoid; or can be more than one object, then composition is partial. Still we need neutral elements (identity functions) and associativity of composition.

In the example above, $a \leqslant b$ was not a function at all; it's 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 synonym is "morphism"; the term is more traditional, and traditionally it scares software engineers, so we won't use it. But they are just synonyms.

Definition 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 $id_Y \circ f = f = f \circ id_X$.

Note that arrows here are pretty abstract; they are not necessarily something that take "elements" (whatever this term may mean) of one object and produce elements of another. All we know about an arrow is its signature: $f : X \to Y$.

Examples

Example 2 Here we will see how a database schema can be viewed as a category. Let's start with definitions.

```
create type rels as enum ('spouse', 'child', 'partner');
create type jobs as enum ('ceo', 'cto', 'eng', 'sales');
create table Person(id bigint, name varchar(80), primary key (id));
create table Company(id bigint, name varchar(80), primary key (id));

create table Rel(from bigint, to bigint, kind rels,
  constraint p1_fk foreign key (from references Person(id)),
  constraint p2_fk  foreign key (to references Person(id)));

create table Job(company bigint, employee bigint, position jobs,
  constraint c_fk foreign key (comp references Company(id)),
  constraint p_fk foreign key (pers references Person(id)));
```

Traditionally we represent a conceptual model of this schema like this, with so-called "foreign keys" as references to other chunks of data:

This follows the tradition of representing all the relations in a relational database as if they were collections of "objects" referencing other objects. If we realize that these "foreign keys" are actually just arrows that allow us to navigate, we get this modified model:

It is a graph; and if we throw in identity arrows, we see that we have a category.

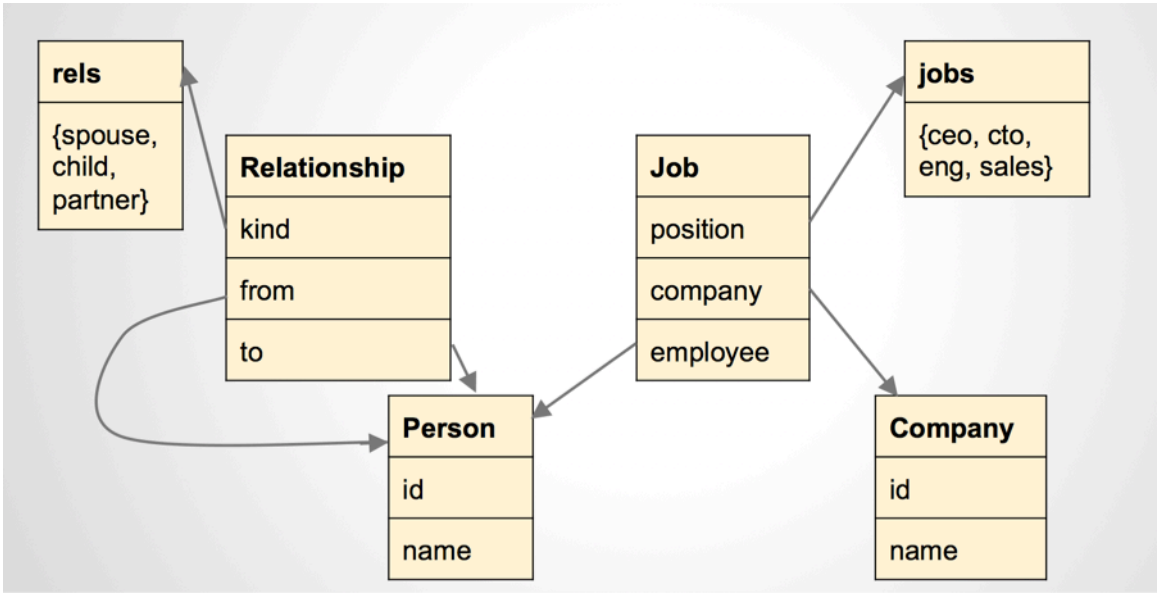


Figure 25: *Sample database*

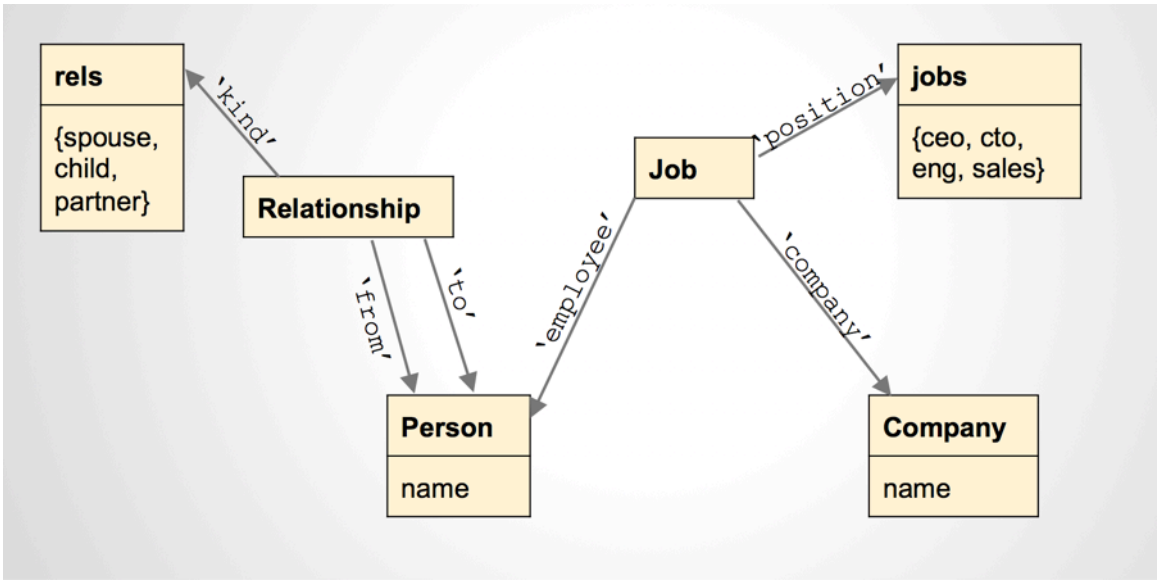


Figure 26: *Sample database, categorically*

This specific example does not have any composition to be defined. To make sure composition is available, we need to take as arrows all possible paths. The trick applies to any graph… by a graph I mean "directed multigraph".

Not all data in a table are foreign keys; other data belong to certain types and can be considered as arrows from a table type to a specific value type. E.g. `Company.name` is an arrow from `Company` to `String` (also known under a strange name of `VARCHAR2`) type; so it makes sense to add to the schema these types as object of the category.

Example 3 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.

Pet Database gives us a chance to look into a specific example of such a graph.

Example 4

```
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)));
```

Again, the "more" conceptual model looks like this, with an example of composition:

Example 5. Empty category An empty category does not have objects, it does not have arrows; it is just empty.

Example 6 Category $\mathbb{1}$ It has one object and one arrow (identity).

Example 7 Categories $\mathbb{1}+\mathbb{1}+...+\mathbb{1}$ – each of these is a discrete category, consisting of n objects and n arrows (identities). Since neither of these arrows have common domain/codomain, there is nothing to compose, except identities with themselves.

This kind of category can represent sets; a segment of Set Theory can be looked at as a part of Category Theory. Of course most of the axioms of Set Theory are missing, so there is not much

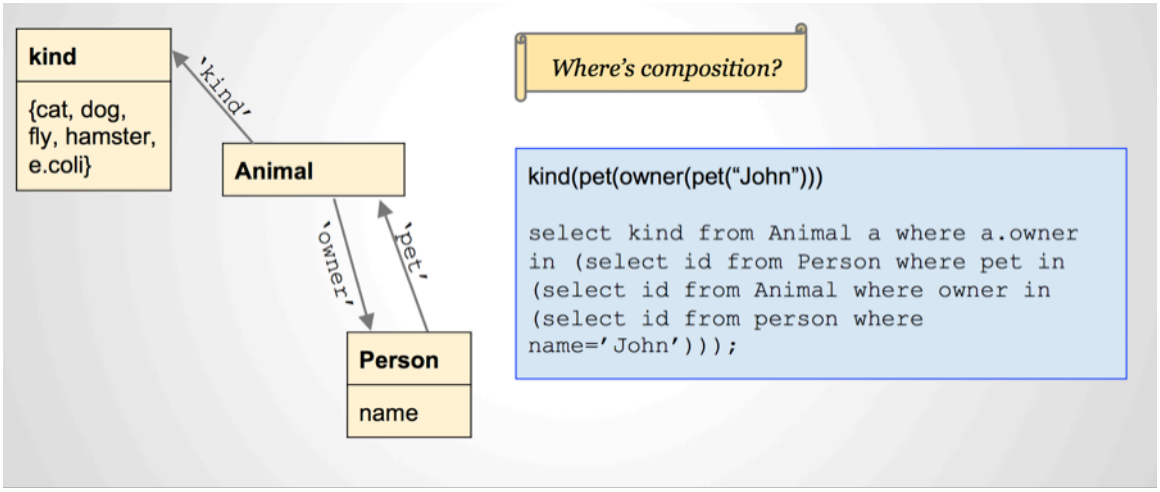


Figure 27: *Animal kingdom, limited*

we can do with such sets. Cannot form a powerset, for instance, or a subset (comprehension does not exist in categories).

Example 8. Category $\mathbb{2}$ It has two objects and three arrows; it 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, so it is $0 \to 1$, but it does not matter much. The "true nature" of the objects does not matter; it is like counting, 1+2=3, be it apples or oranges.

Example 9. Category $\mathbb{3}$ It has three objects and six arrows; like this:

$* \to * \to *$

Now we have a non-trivial composition; but the schema above does not reflect it. It will be better to have a bigger picture:

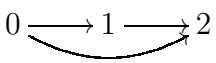


Figure 28: *Category* $\mathbb{3}$

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, $0, 1, 2$, are arbitrary.

Example 10. Category $\mathbb{4}$ It has 4 objects and 10 arrows; look at the picture:

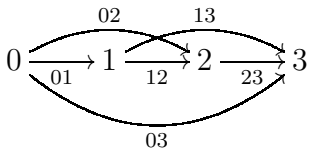


Figure 29: *Category 4 (Identities not shown).*

Now we have enough arrows to involve associativity. Since 03 is unique, it has no choice but to be equal to both $23 \circ 02$ and $13 \circ 01$

Example 11. 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 12. Category of integer numbers, ordered The category is called $\mathbb{Z}$; its objects are integer numbers, and it is a partial order.

Example 13. Category of real numbers, ordered The category is called $\mathbb{R}$; its objects are integer numbers, and it is a partial order.

Example 14. Monoids Any monoid is a category. Just one object x (of abstract nature); all elements of the monoid go as arrows $x \to x$, and we have identity arrow and composition associativity.

Example 15. Category of sets and functions If, given some Set Theory (there is 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 for using as a base for a lot of mathematics on it.

Example 16. Category of sets and partial functions Similar to the previous example, we take sets as objects; but for arrows we 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 be even empty, so that the partial function is not defined at all!

Z-notation uses this arrow symbol: $\nrightarrow$ to denote a partial function.

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

Composition of partial functions is defined like this: if we have $f : X \nrightarrow Y$ 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 $x \in XY_0$ to $g(f(x))$.

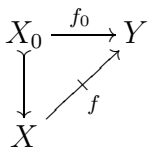


Figure 30: *Partial function*

Associativity follows; and if we take 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.

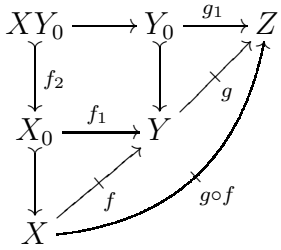


Figure 31: *Composition of partial functions*

So in result we have a category, and it is called $\mathcal{S}et_{Part}$.

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 everybody is used to it; but we don't have to; we can treat them properly as arrows in a category of partial functions.

In Scala, partial functions are a part of 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) > ε => 1/d
}
```

How can we use this function? One method is checking whether the function is defined on a value, like

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

Example 17. Category of sets and their binary relations The category is called $\mathcal{R}el$; its objects are plain sets, and its arrows are binary relations, also known as "many-to-many" relations. Now, do you know how to compose binary relations? It is actually not hard. 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, $\Delta \subset X = \{(x, x) \mid x \in Y\}$, is obviously the identity.

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

Example 18. Vector Spaces and Linear Transformations The category is called $\mathcal{F}inVect$; 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 transformation is a transformation represented by the product of two matrices. The unit $n \times n$ matrix represents the identity arrow on an n-dimensional vector space.

Chapter 9. Working with a Category

Arrows in a Category

We defined category as consisting of objects and arrows; an 'arrow' is an abstract notion reminding functions, but not necessarily. They don't 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 little amount of data? Turned yet that yes, we can talk a lot. But remember, objects too 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 consists of elements.

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

Monomorphism

Definition An arrow $f : A \to B$ is called 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 look equivalent to the definition we had for sets:

An arrow $f : A \to B$ is called an *injection* if from $f(a_1) = f(a_2)$ it follows that $a_1 = a_2$.

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, 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 we have an injection.
- Now suppose we have an injection; then 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 lots of 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 we, on most occasions, can only say that an object A is a subobject of an object B if there is a known monomorphism from A to B.

Epimorphism

If you remember the definition of surjection (epimorphism) in sets, we had to provide, for each $b : B$, an $a : A$ such that $f(a) = b$. This kind of definition is totally unacceptable if we don't have enough elements; so we will have to come up with something different.

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

This does not look any close to the old set-theoretic definition. But see, 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 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 there is a b that is not equal to $f(a)$ for some a, we can introduce a function that is $false$ on this specific b and $true$ on every value $f(a)$. Such a function, composed with f, is equal to constant $true$ composed with f. If these two functions are equal, no such b exists!

The advantage of our definition is that we do not rely on the existence of anything; it is just a universal property, similar to the one used in the definition of monomorphism.

Again, in partial orders every arrow is an epimorphism, just 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 An arrow $f : A \to B$ is called an *isomorphism* if it has an inverse, $g : B \to A$, that is such that $g \circ f = id_A$, and $f \circ g = id_B$.

Every isomorphism is both an epi and a mono; but if we have an arrow that is both an epi and a mono, it does not mean it is an isomorphism. Remember partial order? Every arrow 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 Two objects, X and Y, in a category $\mathcal{C}$, are called *isomorphic* if there is an isomorphism between them. This relation is denoted as $X \cong Y$. Note that this is 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.

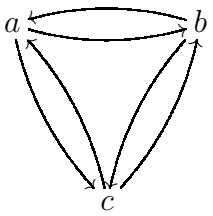


Figure 32: *Category with three isomorphic objects*

Example 1 In the category of Example 1 objects a, b, c are all isomorphic; but they are definitely not equal to each other.

Example 2 Category of sets. Every two single-element sets (singletons) are isomorphic.

Initial and Terminal Objects

If you look at category $\mathbb{3}$:

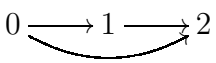


Figure 33: *Category* $\mathbb{3}$

you will notice that the object 0 has a special feature: it has exactly one arrow from it to each object (including itself); and the object 2 has the opposite property: it has exactly one arrow coming to it from each object.

In $\mathcal{S}et$ we have a similar couple of objects (that is, sets): the empty set and a singleton set. An empty set uniquely maps into any set, and any set has a unique function to a singleton. Of course a singleton set is not unique, but they are all isomorphic.

Definition An object T of category $\mathcal{C}$ is called *terminal object* if for any object X there is a unique arrow $f : X \to T$.

Note that because of uniqueness, there is just one arrow $T \to T$; so it must be an identity.

We can easily see that every 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 are identities, right? So these arrows are isomorphisms, and the objects T_1 and T_2 are isomorphic.

Definition An object I of category $\mathcal{C}$ is called *initial object* if for any object X there is a unique arrow $f : I \to X$.

Similarly, an initial object has only one arrow to itself, an identity; and every two initial objects are isomorphic.

Example 3. $\mathcal{S}ets$**.** As it was noted before, in a category of sets, 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 it is included into any set. For the singleton, we can define an arrow from any set to the singleton, and there is exactly one way to do it.

Example 4. Monoids and their functions A unit monoid, 1, the one that has 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. But how about an initial monoid? Any monoid should have a neutral element, meaning that there is no such thing as an empty monoid. More, there is always

a unique function from 1 to M, for every monoid M: it 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. Category $\mathbb{1} + \mathbb{1}$. In this discrete category no object can be initial or terminal, just because there is not enough arrows.

Example 6. 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 has a terminal object; another has an initial object:

Figure 34: *Two Partial Orders.*

Example 7. Scala Scala programming language has a pretty rich type system. In particular, it has `Unit` and `Nothing` types, that play the roles of terminal and initial objects. `Nothing` can be cast to any type; and this is all you can do with `Nothing`, since it has no values. So, by the definition, we have an initial object. `Unit` is the default return value of a function, an analog of `void` in Java. Since every type can be cast to `Unit`, and, (if we happily ignore side effects) there is just one function that returns `Unit` (the one that does not do anything), `Unit` is a terminal object.

Chapter 10. Manipulating Objects in a Category

Product of Two Objects

Category of sets is always a good prototype for generalization. If we have two sets, we can build a Cartesian product out of them. It is a set of pairs, $\{(x, y) \mid x \in X, y \in Y\}$. The operation is not associative: $\{((x, y), z)\}$ is not the same as $\{(x, (y, z))\}$, but 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, we have an *up-to-an-isomorphism* associativity. Similarly, if we build a product 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 literally generalized for an arbitrary category: in an arbitrary category objects do not necessarily consist of elements. But we can generalize elements, replacing them with arrows.

In this specific case we can say this: a product of two sets, X and Y, is such a set that any function $f : Z \to X \times Y$ is in one-to-one correspondence with pairs of functions $(f_X : Z \to X, f_Y : Z \to Y)$, so that $f(z) = (f_X(z), f_Y(z))$.

Why is it an equivalent definition? First, if we take $Z = \{\cdot\}$, we see that for this specific kind of Z the definitions are equivalent. How about an arbitrary Z? Remember, any such Z is a set, and consists of points, representable as $\{\cdot\} \to Z$. If our definition of product holds for points, it will hold for any element of Z, that is, it will hold for the whole Z.

Product in a Category

Now we can try to define a product in general settings, via arrows and universal properties.

Definition Given a category $\mathcal{C}$ and two objects in it, X and Y, their *product* is defined as an object $X \times Y$, together with two arrows, $p_X : X \times Y \to X$, $p_Y : X \times Y \to Y$, 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 $h : Z \to X \times Y$ such that $f = p_X \circ h$ and $g = p_Y \circ h$. Graphically, it looks like this:

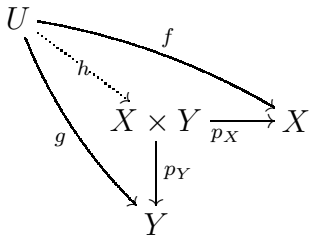


Figure 35: *Cartesian product*

We will use the term "product" instead of "Cartesian product"; it's the same thing.

If we are in a category of sets, this is exactly the definition given in the beginning of this section. But now we are not restricted by objects being sets. Also, we don't even assume the existence of products; in some cases they may not exist, 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 arrow h defined by (f, g). This means that a product may not be unique; but all of them are isomorphic.

For example, if we take $Y \times X$, and two arrows, (p_X, p_Y), we will get a unique arrow *swap* from $Y \times X$ to $X \times Y$ such that the $p_X \circ swap = p_Y$ and $p_Y \circ swap = p_X$; it is obvious that swap is inverse to itself, so the two products, $X \times Y$ and $Y \times X$, are isomorphic.

Similarly we can demonstrate that 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 an impression that they are epimorphisms. 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.

Let's walk through some examples.

Example 1. Tuples In Programming Languages Given two types, `T` and `U`, their product type is the type of *tuple*, `(T,U)`. In Scala, alternatively, you can 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. Data Structures We don't have to limit ourselves by unnamed tuples. Any data structure, e.g.

```
struct S {
  Int id;
  String name;
  Date date_of_birth;
}
```

can be thought of as a product type. We have three base types here, `Int`, `String`, `Date`, and projections from `S` to these types; the projections are named `id`, `name`, `date_of_birth`.

Why is it a product? Given a type `T`, and three functions, `i: T => Int`, `n: T => String`, `d: T => Date` (using Scala notation here) 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 gives us the original functions.

Example 3. Cross Join in SQL Take a relational database; it 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 we do not add any conditions, all pairs of table rows are included. Categorically, every pair of references to rows in `table1` and `table2` can give us a reference to a row in this product.

Example 4 Categories $\mathbb{1} + \mathbb{1} + ... + \mathbb{1}$ – the only product that exists here is a product of an object with itself; and it is the same object.

Example 5 Category $\mathbb{2}$. Here we have two objects, 0 and 1, so we need to figure out what are 0×0, 0×1, and 1×1. The first one, 0×0, obviously amounts to 0: it has two "projections" to itself, and since there is no other arrow into 0, that's it. A little bit more elaboration leads us to the conclusions that 0×1 is 0 and 1×1 is 1.

Example 6 Take any partial order P as a category. Take two objects, X and Y, in this category. Suppose a product, $X \times Y$, exists. Having 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, in a partial order, that $Z \leqslant X$ and $Z \leqslant Y$; and, by the definition of product, every such Z should satisfy $Z \leqslant X \times Y$. This is exactly the definition of greatest lower bound of X and Y.

Example 7 Given any set A, take as a partial order its lattice of subsets, $P(A)$. We already know (from the previous example) that in such a category $g.l.b.(X, Y)$ plays the role of $X \times Y$. But what is it for two subsets? It is their intersection, $X \cap Y$. This is not the classical product of two sets, but this is exactly the greatest lower bound of the two. It is the product of X and Y in the category $P(A)$.

Example 8 Take a monoid M as a category. We have only one object, so its product with itself, if exists, 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 element of the monoid); and 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 just one element). No other monoid, viewed as a category, can have a product.

Example 9 Take a Heyting algebra, H. Being a partial order, it is a category. In such a category, $a \wedge b$ plays the role of Cartesian product. Why so? First, we have $a \wedge b \leqslant a$ and $a \wedge b \leqslant a$ - 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 Cartesian product.

Neutral Element for Cartesian Product

We already observed that Cartesian product is, up to an isomorphism, associative. To make it look more like a monoid, we need a neutral element; 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 be not unique; but all such instances are isomorphic.

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 it is enough to have $g : Z \to X$ 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 $f : Z \to 1$. So X is a valid instance of $1 \times X$ (remember, a product is defined up to an isomorphism).

Now that we have enough examples and illustrations of product, let's see if we can generalize set union to arbitrary objects of an arbitrary category.

Sum of Two Objects

Some programming languages have union types, to reflect the idea that values may belong to either one type or another. E.g. in Haskell this is implemented like this:

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

This whole expression, for those who are not familiar with Haskell, says that, given two types, `a` and `b`, we define a *data type* `Either a b`, and instances of this type are represented either with a constructor `Left a`, or with a constructor `Right b`. `Left a` references a value of type `a`, and has a marker `Left`; similarly `Right b` references a value of `b`, and has a marker `Right`. The same idea is expressed more verbosely in Scala as

```
sealed 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]
```

The word `sealed` means that we cannot have implementations of `Either` outside the scope of this code.

If we translate all 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 saying whether we are dealing with "left copy" or "right copy". Actually, we could express this idea using Cartesian product: by multiplying A by a set $\{0, 1\}$. We will 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.

Now the problem is, how do we 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 "on the right" to both X and Y, same way as their product is closest to X and Y "on the left" . Here "left" and "right" means that arrows on the diagrams are usually shown as going left-to-right.

So, we are coming to the following

Definition Given a category $\mathcal{C}$ and two objects in it, X and Y, their *Sum* is an object denoted as $X + Y$, together with two arrows, $i_X : X \to X + Y$, $i_Y : Y \to X + Y$, 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:

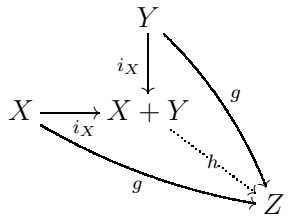


Figure 36: *Sum*

Sum is also known as *disjoint sum*, or *disjoint union*, or just *union*. Note that, depending on the category, the union does not have to be *actually* disjoint (see example 11).

Example 10 Take two sets, X and Y, with no common elements. Their union, in a category of sets, has this universal property as in the definition above. Why so? 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 a sum of X and Y; we just need to provide inclusions i_X and i_Y.

Example 11 Take a set A, and its lattice of subsets, $P(A)$. 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 12 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 word "disjoint" has no meaning in this specific case.

Monoidal Properties of Sum

Due to the universality, similarly to Cartesian product, we can demonstrate that the sum of two objects does not depend on their order; and that sum is associative: both properties up to an isomorphism: $X + Y \cong Y + X$ and $(X + Y) + Z \cong X + (Y + Z)$.

To have a monoid, up to an isomorphism, we need a neutral element. An initial object is such an element. The argument is similar (dual, actually) to that for Cartesian product, and I won't repeat it here.

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. So we will start with equalizers.

Equalizer

Definition Given a category $\mathcal{C}$ and two parallel arrows, $f : X \to Y$ and $g : X \to Y$, we say that an arrow $h : Z \to X$ *equalizes* f and g if $f \circ h = g \circ h$.

Out of all such equalizing arrows for f and g one (up to an isomorphism, of course) has a universal property, and is called an equalizer of f and g.

Definition 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 picture illustrates it:

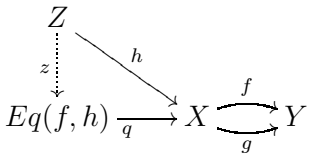


Figure 37: *Equalizer*

Example 13. Equalizer in Sets This is the simplest case: given two functions, f and g, their equalizer is $\{x \mid f(x) = g(x)\}$. You can easily check that this subset of X satisfies the definition.

Example 14. Fixpoint If g is identity, then $Eq(f, id_X) = \{x \mid f(x) = x\}$ (that's in sets), and it is clear that it is exactly the definition of fixpoints of f. We either consider $Eq(f, id_X)$ a set of fixpoints of f, or define 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 it does not, the generalized fixpoint is an empty set.

Example 15. Partial Order If we have a partial order, every two parallel arrows are equal; so equalizing them does not make much sense: it is always X with its identity arrow.

Example 16. 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 Russia database. Then the following SQL query: `select * from users where id = boss;` amounts to finding all fixpoints, that is, all the people that are their own bosses. And the result is an equalizer, as mentioned above.

Note that an equalizer is also a monomorphism. You can try and prove it by yourself; it is not hard.

Coequalizer

Dually to the notion of equalizing and equalizer, we can introduce *coequalizing* and *coequalizer*.

Definition 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 (strictly speaking, it is called *couniversal*) among all arrows coequalizing f and g.

Generally speaking, building a coequalizer may take more computational efforts. E.g. if we talk about sets, a coequalizer can be built as a *factorset* of Y, defined by an equivalence relation $f(x) \cong g(x)$. To check if $y_a \cong y_1$, one has to 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; we won't focus on them either.

Pullback

We have now two essential structures, product and equalizer, that can help build other universal structures. The first one is *pullback*, which encapsulates common features of a product and an equalizer. Take a look:

This is a pullback diagram for (f, g). We have two arrows, $f : X \to Z$ and $g : Y \to Z$; and we build an object called *pullback*, denoted as $X \times_Z Y$, (you can read it as "X cross Y over Z"), together with two *projections*, (p, q), having two properties:

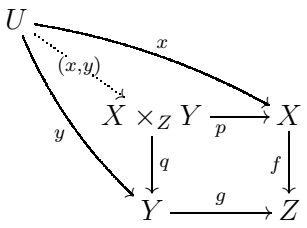


Figure 38: *Pullback*

- it 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 as we saw in the beginning of this chapter), and
- any other (U, x, y) such that $g \circ y = f \circ x$, can be split through $X \times_Z Y$ (see the diagram).

The definition may look over-complicated; but you'll immediately get the idea, if you have not yet, from the following database example:

Example 18. Databases `select * from tableX, tableY where tableX.f = tableY.g;`

This SQL join query conveys the essence of pullback: take all the data from two tables such that certain values match.

```
select * from user, company where user.address = company.address;
```

This query lists all the people that have companies in their garage, together with the company information. And it is a pullback.

Example 18. Sets Having two arrows, $f : X \to Z$ and $g : Y \to Z$, we can express their pullback as $\{(x, y) \mid x \in X \wedge y \in Y \wedge f(x) = g(y)\}$.

Example 19. Partial Order In a partial order there is at most one arrow between two objects; so a pullback here is the same as a product, that is, the greatest lower bound, if exists, is also a pullback.

Properties of 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:

We can also express product as a pullback; just take $Z = 1$, then arrows f and g become irrelevant in forming the pullback:

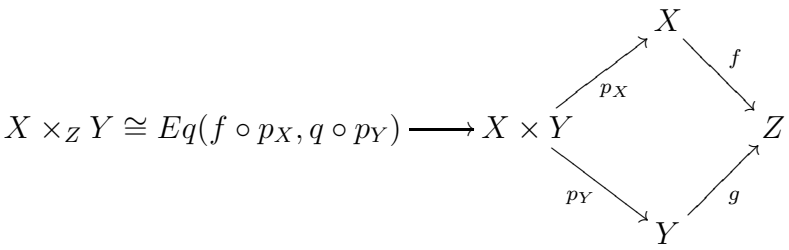


Figure 39: *Pullback is an equalizer.*

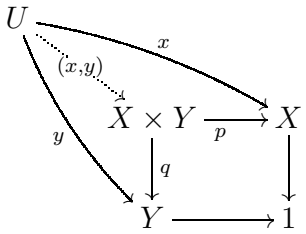


Figure 40: *Product is a pullback.*

Expressing an equalizer via pullback is less obvious, but is still doable. Look at the following diagram:

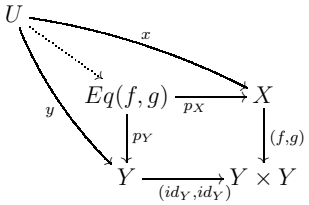


Figure 41: *Equalizer is a pullback.*

What do we see here? We have two arrows, $f : X \to Y$ and $g : X \to Y$, and we build a product $Y \times Y$, so the pair can be expressed as one arrow $(f, g) : X \to Y \times Y$. We also use another arrow (known as *diagonal*): $(id_Y, id_Y) : Y \to Y \times Y$. Having a pair (x, y) equalizing these two is the same as saying that $f \circ x = y$ and $g \circ x = y$, that is, equalizing f and g.

Pushout

Similarly to pullbacks, we can define pushouts, by reversing all the arrows.

This construction can be expressed, similar to pullbacks, via sums and coequalizers. And, since coequalizers are hard to calculate, pushouts are hard to calculate too, with the exception of the case of partial orders, where pushout is the same as union, and is the lowest upper bound of X and Y.

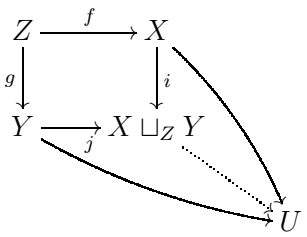


Figure 42: *Pushout*

Chapter 11. Relations Between Categories

So far we've been looking at individual categories as if they were isolated galaxies: interesting structures inside, no connections between them. In this chapter we will investigate how categories relate to each other, so that we can actually have a category of all categories, a thing that is impossible for sets, for instance.

To build a category of categories, we need to introduce arrows between categories, their composition, and identity arrows on categories (this one is the easiest).

Let's start with two predecessors of categories: monoids and graphs.

In $\mathcal{G}rph$, a 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 $\mathcal{M}on$, a category of all monoids, arrows are monoid functions - the functions that preserve the neutral element and the binary operation.

Since a category is both a graph and a generalization of 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.

Functor

Definition Given two categories, $\mathcal{A}$ and $\mathcal{B}$, a *functor* $f : \mathcal{A} \to \mathcal{B}$ consists of the following components:

- F_0 that maps objects of $\mathcal{A}$ to objects of $\mathcal{B}$;
- F_1 that 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, we can safely (and for simplicity) omit the subscripts and further denote both with just one letter F. Also, application of a functor, according to a tradition of certain programming languages,

will be denoted not as $F(X)$ but as $F[X]$; you will see that, although unusual, it does make 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 functor also requires that for an arrow $f : X \to Y$

we should have $F[f] : F[X] \to F[Y]$. But this actually just follows from our definition above: $F[f] = F[f \circ id_X] = F[f] \circ id_F[X]$ — this 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 word to denote any arrow that acts on other arrows. This is wrong. A mapping from one arrow to another is called an *operator* in mathematics (e.g. 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, actually, an *endofunctor*, since its domain and codomain are equal.

Before showing examples, we will wrap it up with showing that we have a category: a category of categories, called $\mathcal{C}at$. Categories are objects in $\mathcal{C}at$, and functors are arrows. Composition of two functors is easy to define: apply one, then apply another. It is associative just 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 is okay; we are not dealing with sets, we do not have set-theoretic axioms, specifically, the Comprehension Axiom that would allow us to build a category of all barbers that don't shave themselves.

Examples of Functors

Example 1. `List[+T]` This is the most popular functor in programming. We will not discuss its features, just look at it from a categorical point of view. In an unspecified programming language (as long as it is Scala) we have its types as objects of the category, and single-argument functions as arrows of the category. Now, given a type `T`, we can produce a type `List[T]`; so we have a functor defined on objects. How 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 really 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`.

If, instead of a regular category $\mathcal{S}cala$, we use the category (the partial order, rather) where only subtypings are allowed as arrows, we don't have to introduce map function; the language can provide us with a feature that from $isSubtype : T \to U$ we have $isSubtype : List[T] \to List[U]$. This feature of a paramerized type is denoted in Scala by having the `+` sign, like in the title of this example (namely, `List[+T]`). More on this later.

Counterexample 2. Set[T] In Scala, as well as in Java, and probably in some other languages, `Set[T]` is a parameterized type representing "a set of values of type T". But mapping just types is not enough; we need to map functions, and here we have 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` on elements of the original set, by building an image of function `f`. This is not very efficient, since the new set should be materialized right away. If it remained virtual (lazy), like in the case of `List`, we would have for every call of `set.contains(x)` method scan through the whole original set and compare the result of function application with the value provided.

A different solution exists; but it requires some new notions, to be introduced later on.

Example 3. A functor from category $\mathbb{1}$ $\mathbb{1}$ is a category consisting of one object and its identity arrow. How does a functor from $\mathbb{1}$ to a category $\mathcal{C}$ look like? To specify object mapping, we need to select an object in $\mathcal{C}$, and that's all that we need. 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 $\mathcal{A}$. How is it? We map every objects of $\mathcal{A}$ to x, and every arrow to id_x. All the properties of a functor are satisfied, are not they? You can check it yourself.

Example 5. A functor from 2 to $\mathcal{S}et$ If you remember, 2 is a category 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 a category of sets. To define a functor F from 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 $F[1]$.

For the reasons of beauty, we better rename $F[0]$ to F_0, $F[1]$ to F_1, and $F[01]$ to F_{01}. We see that every functor from 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 $F_{01} = f$.

$$F[0] \xrightarrow{F[01]} F[1]$$

Figure 43: *Functor from* $\mathbb{2}$ *to* $\mathcal{S}et$

Actually, the fact that we are dealing with category $\mathcal{S}et$ is irrelevant. The same argument would work for any category $\mathcal{C}$: functors $\mathbb{2} \to \mathcal{C}$ are arrows in $\mathcal{C}$.

Example 6. A functor from $\mathcal{S}et$ to $\mathbb{2}$ Now we will try to figure out what we can have here. Two obvious functors are constants: map all objects of $\mathcal{S}et$ to object 0 of $\mathbb{2}$, and all arrows to id_0, and similarly, map all objects of $\mathcal{S}et$ to object 1 of $\mathbb{2}$, and all arrows to id_1.

Except for these two obvious functors, there must be others that cover both 0 and 1. Note that the empty set $\emptyset$ is a subset of every set, so if a functor F maps it to 1, every other set should map to 1. Meaning, we will have the constant functor we talked about above. Similarly with singletons, which are terminal objects, either they map to 1 or every set maps to 0. We have mappings for $\emptyset$ and singletons. Every nonempty set S has an element, and so there is an arrow from singleton to S; so S should also map to 1. As you see, all nonempty sets should map to 1. We have exactly three functors from $\mathcal{S}et$ to $\mathbb{2}$.

Example 7. Product with an object Given a category that has products, and an object A in , 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 figured out already how it does it.

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, which consists 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 $f : X \to Y$, $f^A : X^A \to Y^A$. We can define it element-wise. Given an $x_a \in X^A$, that is, $x_a : A \to X$, we produce $y_a : A \to Y$ by defining it as $y_a = x_a \circ f$.

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 preserve multiplication (acting as composition) and the neutral element (acting as identity arrow).

Example 10. Partial Order A partial order is also a category; and a functor between two such categories is a partial order function that preserves order (it is called *monotone function*).

Example 11. Integers to Reals The regular inclusion of $\mathbb{Z}$, the partial order of integer numbers, into $\mathbb{R}$, the partial order of real numbers, does preserve the order, so it is a functor, from categorical point of view.

Example 12. 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 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, and, since it preserves composition and identities, we 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, equalizers in $\mathcal{C}at$, and whether it has initial and terminal objects. Let's walk through all these.

Initial Category

This is a category that has a unique functor to any (other) category. Of course such a category cannot have objects; if it did, we could apply constant functor to it, for each object in a target category, and have more than one such functor, generally speaking. So 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 $\mathcal{C} \to \mathbb{1}$. So you see that we have a terminal object in $\mathcal{C}at$.

Product of Two Categories

This structure can be built similar to what we have in $\mathcal{S}et$. 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 very fact that math allows us to form such pairs is beyond the scope of this text, of course; this can be done internally if we are within a certain domain. As arrows, 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 see that we have a category, we provide identities $id_\$X, Y)\} = (id_X, id_Y)$, and verify that they are neutral re: composition; and that the composition is associative.

Does this category satisfy the universal property in $\mathcal{C}at$? It can be proved component-wise that it does.

Note that if we have $\mathcal{C} \times \mathcal{D}$, there are two projection functors, $\mathcal{C} \times \mathcal{D} \to \mathcal{C}$ and $\mathcal{C} \times \mathcal{D} \to \mathcal{D}$; they map objects (x, y) to x and y, and the same happens to arrows. There is also a diagonal functor $\Delta : \mathcal{C} \to \mathcal{C} \times \ \mathcal{C}$ that takes an object x of $\mathcal{C}$ to $(x, x) \in \mathcal{C} \times \mathcal{C}$ and an $f : x \to y$ to $(f, f) : (x, x) \to (y, y)$.

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 get a new category, $\mathcal{C} + \mathcal{D}$. We already saw examples of this: 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 of them being that equality for objects is not defined in categories, only isomorphisms; so we would have to define everything up to an isomorphism, which gets us into higher-order categories. Higher-order categories are not a part of this book.

Reversing the Arrows

Remember that arrows in a category have, in general, nothing to do with something that takes an argument and returns a value; they are just formal abstractions. So, given a category $\mathcal{C}$, nothing can stop us from producing another category out of it, by reversing the direction of all arrows.

Definition. Given a category $\mathcal{C}$, its *opposite*, or *dual*, $\mathcal{C}^{op}$, is a category with the same objects and the same arrows, but the direction of arrows is reverted. If, in $\mathcal{C}$, $f : X \to Y$, in $\mathcal{C}^{op}$ we have $f : Y \to X$.

Note that having any knowledge about arrow makes no sense here; these are just symbols. Composition is defined in the opposite direction too, so $(f \circ g)^{op} = g^{op} \circ f^{op}$. The fact that it is a category can be easily proved (you can do it as an exercise).

For some categories opposite is isomorphic to the original category (e.g. $\mathbb{1}$, $\mathbb{2}$, $\mathbb{3}$…); even $\mathcal{R}el$, a category of sets and their binary relations, is symmetrical relative to this operation; for others it is not trivial at all. For instance, $\mathcal{S}et^{op}$ is the category of *Complete Atomic Boolean Algebras*.

Omitting exact definitions, we can intuitively look into it like this: given a set, we have its characteristic function, a predicate that is true only on members of the set. Now, if we have an arrow $f : X \to Y$ on sets, and for set X we have a predicate p_X, and for set Y we have a predicate p_Y, we can map p_Y to a predicate on X by defining $f(p_Y)(x) = p_Y(f(x))$. This way, for each map between sets we have a map between predicates. We can view sets of such predicates, for each given set, as objects of the category; and we, under certain assumptions, may think of such predicates as being the same as the underlying sets.

In programming languages this operation is equivalent to defining sets 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, one "characteristic" predicate for each element of the original set.

Contravariant Functor

Frequently the functors we've been discussing so far are called *covariant* functors, due to their actions on arrows that map domain to domain and codomain to codomain. Another kind of functor, the one that maps domain of an arrow to codomain, and 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 $\mathcal{C} \to \mathcal{D}^{op}$).

But since variance plays an important role in computer science, we must spend some time discussing it.

Example 1. Map[_,T] If, in Scala, we fix the second argument of the parameterized type `Map`, we have this feature that for an arrow `f: X => Y`, we can produce an arrow `Map[Y,T] => Map[X,T]`. This mapping preserves identities and composition; so we have a contravariant functor. In Scala, contravariance is denoted using minus: `Map[-X,+Y]` is the signature of this type.

Variance in Programming Languages

Usually, in languages allowing subtyping (e.g. in Scala) parameterized classes, if they happen to be functors, get their variance marker not because they behave covariantly or contravariantly on arbitrary arrows, but only on inclusions ("subtyping") of types into other types. So that, e.g., if `A<:B` and `X<:Y` (this is a notation for the compiler's ability to subtype one into another), we have `Map[B,X]<:Map[A,X]` and `Map[A,X]<:Map[A,Y]`. We are obviously dealing with a category where types are objects, and the relations of subtyping are arrows. This is a partial order, so things are easier than with generic arrows.

Chapter 12. Relations Between Functors

Natural Transformations

We were gradually climbing up the hiearchy of relations. Objects and arrows constitute a category. Then we studied relations between categories, which are, in the simplest case, represented by functors. Now is time to study relations between functors; this will also help us get a fresher look at constructs within categories.

For 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 practically are dealing with the same category $\mathcal{B}$, we just treat its objects as functors, and arrows between objects as arrows from one functor to another.

Now we expand 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 example with $\mathcal{A} = \mathbb{1}$, 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 $f_1 : x_1 \to y_1$; and we need the following diagram to be a commutative square (that is, $y_{01} \circ f_0 = f_1 \circ x_{01}$):

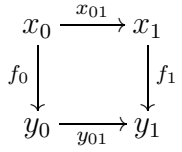


Figure 44: *Arrow between two arrows*

Is it really a category, the collection of functors from $\mathbb{2}$ to $\mathcal{B}$, and pairs of arrows, (f_0, f_1)?

To have a category, we need to define identity arrows, and, for each appropriate pair of arrows, their composition that is associative.

An identity on x consists of (id_{x_0}, id_{x_1}); and we define composition component-wise $(f_0,\ f_1) \circ (g_0,\ g_1) \equiv (f_0 \circ g_0,\ f_1 \circ g_1)$.

The two examples above will help us define arrows between functors $\mathcal{A} \to \mathcal{B}$ in a general case.

$$\begin{array}{ccc} F[x] & \xrightarrow{F[h]} & F[y] \\ {\scriptstyle t_x}\downarrow & & \downarrow{\scriptstyle t_y} \\ G[x] & \xrightarrow[G[h]]{} & G[y] \end{array}$$

Figure 45: *Natural transformation*

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 $t_x \circ F[h] = G[h] \circ t_y$.

There is a nice way to draw a natural transformation as 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 46: *Natural transformation notation*

We will need more examples, to illustrate how a natural transformation may look like in different circumstances.

Example 1. Flatten a List

`flatten: List[List[T]] => List[T]` (this is Scala code)

The action is pretty simple and familiar: we take a list of lists and flatten it, concatenating its elements to produce one single list of values of type `T`. We have to check that this action is a natural transformation.

First, for a natural transformation we need two functors. We have `List[_]` and `List[List[_]]`; are these two functors? `List[_]` is known to be a functor, we discussed it in the previous chapter. It maps a type `T` to the type of lists of `T`. If we compose this

functor with itself, we now get a functor again; this functor, `List[List[_]]`, maps each type `T` to the type "list of lists of `T`".

`List[List[T]]` and it is a functor because it is a composition of functors.

So, we have `flatten` defined on each type of form `List[List[T]]`. Is it a natural transformation? We need to check this: given an arrow `f: T => U`, it should make the square commute (See Figure 4).

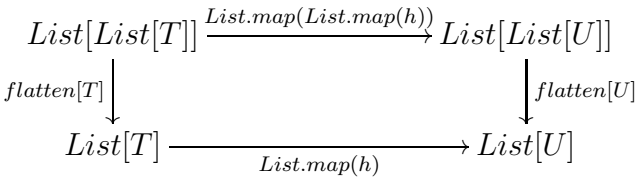


Figure 47: *`List.flatten` is a natural transformation.*

Is it true? One path in the square consists of first flattening, then mapping; the other one is first mapping, then flattening. You can check it out, using the definitions of `flatten` and `map`, that it is the same thing. So (I hope your results are positive, as are mine) we have 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; the functor on the right is `List[_]`. Is it a natural transformation?

For this to be a natural transformation, we need, given a function `f: T => U`, to have `f(t)::Nil = (t::Nil).map(f)`, which is obvious (I hope).

Example 3. Arrows are Natural Transformations

Recapping the beginning of this chapter, 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 let's 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?

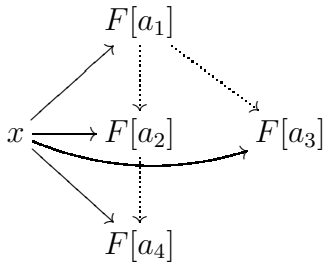


Figure 48: *Cone*

We will need, for each object a in $\mathcal{A}$, an arrow $f_a : x \to F[a]$ such that it is compatible with all $F[a] \to F[b]$, like in the following diagram:

This diagram is called a *cone*. There is a similar diagram, when we take a natural transformation from a functor F to a constant functor; it is called a *cocone*.

Now that we know that all functors from $\mathcal{A}$ to $\mathcal{B}$ 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 instance, $\mathcal{A}^{\mathbb{1}+\mathbb{1}}$ is equivalent to $\mathcal{A} \times \mathcal{A}$.

Adjoint Functors

We have covered relations between functors with the same domain and codomain, that is, pointing in the same direction.

Another case is when we have a pair of parallel functors pointing in the opposite directions. A spooky entanglement had been observed between such functors; and this section is dedicated to this case.

Example 5. Galois Connection

Let's start with a simple case, two partial orders, say, $\mathbb{Z}$ and $\mathbb{R}$. We have an inclusion $Include : \mathbb{Z} \to \mathbb{R}$, and another functor, integral part, $Int : \mathbb{R} \to \mathbb{Z}$. Have you noticed that they are related? For each $x \in \mathbb{R}$ and $n \in \mathbb{Z}$, $x < n$ is equivalent to $[x] < n$; so we have 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 between two partial orders is called Galois connection.

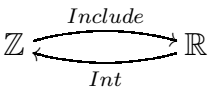


Figure 49: *Galois connection between ℤ and ℝ*

Example 6. Partial Functions

Now a richer example. If we take two familiar categories, regular sets with functions, $\mathcal{S}et$, and sets with partial functions, $\mathcal{S}et_{\mathcal{P}art}$, we can try to build a similar related pair of functors.

The category $\mathcal{S}et_{\mathcal{P}art}$ consists of sets as objects and partial functions as arrows:

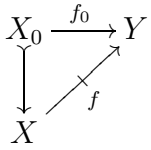


Figure 50: *Partial function*

There is an obvious inclusion functor from $\mathcal{S}et$ to $\mathcal{S}et_{\mathcal{P}art}$ 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, $\mathcal{S}et_{\mathcal{P}art} \rightarrow \mathcal{S}et$; how can we do it?

Here is the idea. Fix a singleton in $\mathcal{S}et$, call it $1 = \{0\}$. For every set X in $\mathcal{S}et_{\mathcal{P}art}$ choose $X + 1$ in $\mathcal{S}et$, that is, the same set with one appended element. This is a disjoint union, so it is irrelevant whether X did contain 1 as a subset; 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 $\mathcal{S}et_{\mathcal{P}art}$, that is, for a partial function $X \nrightarrow Y$, represented by $X_0 \rightarrow Y$ where $X_0 \subset X$, define a function $Option[f]$ in $\mathcal{S}et$ like this:

$Option[f](0) = 0$;
$Option[f](x) = if(x \in X_0)\ then\ f(x)\ else\ 0$.

We need to check that we have a functor, that is, identity and composition are preserved. Leaving it for the reader to enjoy.

With these two functors, $Include$ and $Option$, the picture is similar to what we had in Example 5: arrows $f : Include[X] \nrightarrow Y$ are in one-to-one correspondence with functions $g : X \rightarrow Option[Y]$.

How does it happen?

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 one-to-one correspondence with a function $X \to Y + 1$.

Similarly, a function $g : X \to Y + 1$ is uniquely defined by $g0 : X_0 \to Y$ for some $X_0 \subset X$.

Now we have a Galois connection again:

$$Set \underset{Option}{\overset{Include}{\rightleftarrows}} Set_{Part}$$

Figure 51: *Galois connection between $\mathcal{S}et_{\mathcal{P}art}$ and $\mathcal{S}et$*

We can actually model all this in Scala; 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 $\mathcal{S}et$; what would be a generic case, though, it is hard to define at our current level.

Example 7. Binary Relations

In chapter 7 we met the category $\mathcal{R}el$, and in chapter 11 we saw a functor embedding $\mathcal{S}et$ into $\mathcal{R}el$. Here we will build a functor in the opposite direction. First, we introduce one more Z-notation: $\leftrightarrow$ for binary relation, e.g. $R : X \leftrightarrow Y$. And 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 in $\mathcal{R}el$, that is, each set, to $Pow[X]$ as an object in $\mathcal{S}et$. As to the mapping of functions, given a relation $R : X \leftrightarrow Y$, we build a function $Pow[f] : Pow[X] \to Pow[Y]$ by defining $Pow[f](X_1) = \{y \mid \exists x : xRy\}$. 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: $R = \{(x, y) \mid y \in p_R(x)\}$. So we have the same kind of connection between the two.

By the way, the latter construction, lifting of a relation to a function, can be illustrated in SQL:

```
select y from MyTable where x=?;
```

We have a binary relation `MyTable`, and we "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 being *left adjoint* and G being *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 $F \dashv G$.

Now you can see that the three examples above are examples of adjoint functors; more, every Galois connection is actually just an adjunction.

Specifically, in the first example, $Include \dashv Int$; in the second example, $Include \dashv Option$, and in the third example, $Include \dashv P$.

Let's have more examples, with the functors that you are already familiar with.

Example 8. Yoneda Lemma

We take the category of sets, $\mathcal{S}et$, one (arbitrary) set $A \in \mathcal{S}et$, and two functors, $- \times A$ and $-^A$. The first functor takes a set X and returns a set $X \times A$; the second functor takes a set Y and returns a set of functions $A \to Y$, that is, Y^A.

Yoneda lemma states that the first functor is left adjoint to the second functor:

$(- \times A) \vdash (-)^A$.

The adjunction is exactly what currying and uncurrying (see Ch.1) gives us.

For an $f : X \times A \to Y$, currying gives us $curry(f) : X \to Y^A$; for an $g : X \to Y^A$, uncurrying gives us $uncurry(g) : X \times A \to Y$.

Example 9. 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 a part of an adjoint pair, a left adjoint. Let's see what it would mean in practice. Take two objects, one, Z, in $\mathcal{A}$ and another in $\mathcal{A} \times \mathcal{A}$. The object in $\mathcal{A} \times \mathcal{A}$ is a pair, and we will denote it as (X, Y).

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

obviously the familiar Cartesian product functor, $\Pi : \mathcal{A} \times \mathcal{A} \ \rightarrow \ \mathcal{A}$: having an arrow $Z \rightarrow X \times Y$ is the same as having a pair of arrows, $(Z \rightarrow X,\ Z \rightarrow Y)$.

As a result, we have $\Pi \dashv \Delta$. There's no guarantee, though, that such a functor, Π, that is, a Cartesian product, exists for category $\mathcal{A}$.

Alternative Definition of Adjoint Functors

Another way to declare that two functors, $F : \ \mathcal{A} \rightarrow \mathcal{B}$ and $G : \ \mathcal{B} \rightarrow \mathcal{A}$, are adjoint, is to have a couple of natural transformations, unit $\eta : id_{\mathcal{B}} \rightarrow GF$ and counit $\epsilon : FG \rightarrow id_{\mathcal{A}}$; these two transformations must have the following properties (see Figure 9):

$$F[X] \xrightarrow{F[\eta_X]} FGF[X] \xrightarrow{\epsilon_{F[X]}} F[X], \quad F[X] \xrightarrow{id_{F[X]}} F[X]$$

$$G[Y] \xrightarrow{\eta_{G[Y]}} GFG[Y] \xrightarrow{G[\epsilon_Y]} G[Y], \quad G[Y] \xrightarrow{id_{G[Y]}} G[Y]$$

Figure 52: *Properties of Unit and Counit*

Why is this equivalent? See, having an adjoint pair $F \dashv G$, we take $Id : F[X] \rightarrow F[X]$ and reflect it, using our adjunction, to $\eta_X : X \rightarrow GF[X]$; similarly, taking $Id : G[Y] \rightarrow G[Y]$, we use the adjunction and reflect it to $\epsilon_Y : FG[Y] \rightarrow Y$. Properties follow.

On the other hand, if we have unit η and counit ϵ, then every function $F[X] \rightarrow Y$ gives us $GF[X] \rightarrow G[Y]$, and if we compose it with the unit, we have $X \rightarrow GF[X] \rightarrow G[Y]$, which is the function from X to $G[Y]$ we were looking for. Similarly, we can go in the opposite direction.

One can ask, can a functor have both left and right adjoint? Yes, it happens; but it is beyond the scope of this book. Another question: can a functor have two left adjoints, or two right adjoints? The answer is - kind of. Because you can easily check that if F has, say, a right adjoint G_1 and another right adjoint G_2, there is an isomorphism between the two. So, adjunction is defined up to an isomorphism, like many things in category theory.

Now that we have an alternative definition of adjunction, with unit and counit, how do these unit and counit look like for the adjoint pairs that we already saw?

Example 10. Unit and counit for partial functions adjunction.

As you saw above, to build a unit: $X \to X + 1$ we need to take id_X as an arrow in $\mathcal{S}et_{\mathcal{P}art}$, and reflect it back to $\mathcal{S}et$; it gives an inclusion of X into $X + 1$: and this is our unit η. Regarding counit ϵ, it is just an identity arrow in $\mathcal{S}et_{\mathcal{P}art}$.

Example 11. Unit and counit for $\mathcal{S}et$ and $\mathcal{R}el$ adjunction.

Again, given a set X, start with a binary relation $x == x$(that's how identity function looks like 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. Counit ϵ again is just an identity.

Example 12. Unit and counit for Cartesian product adjunction.

In Example 4 we had this adjunction. Now, if we have a category $\mathcal{A}$, and an object X in $\mathcal{A}$, this object maps to (X, X); reflecting identity on (X, X) back to $\mathcal{A}$, we get $X \to X \times X$, and this function is obviously the diagonal, $(id_X, id_X) : X \to X \times X$. Counit for this adjunction will look like this: given a pair (X, Y) in $\mathcal{A} \times \mathcal{A}$, it maps to $X \times Y$ in $\mathcal{A}$, so when we get it back to $\mathcal{A} \times \mathcal{A}$, we have $(X \times Y, X \times Y)$, and the counit function $(X \times Y, X \times Y) \to (X, Y)$ consists of a pair of projections, (p_X, p_Y).

Limits

In Example 4 we had 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 like this:

Functor $Const : \mathcal{B} \to \mathcal{B}^{\mathcal{A}}$ may have an adjoint functor $Lim : \mathcal{B}^{\mathcal{A}} \to \mathcal{B}$, such that $Const \dashv Lim$.

For such an adjoint to exist, we need a one-to-one correspondence between cones (see Example 4) $x \to F$ and regular arrows $x \to Lim[F]$.

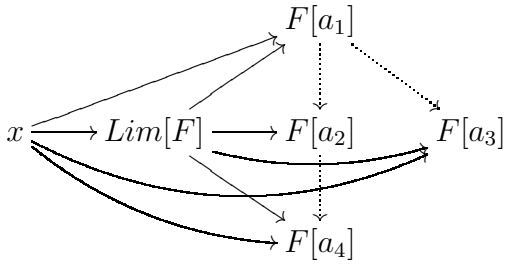


Figure 53: *Limit*

Example 13. Cartesian product as a limit

You might have noticed that if category $\mathcal{A}$ is $\mathbb{1} + \mathbb{1}$, the diagram is exactly the diagram for Cartesian product: that's because $\mathcal{B}^{\mathbb{1}+\mathbb{1}}$ is $\mathcal{B} \times \mathcal{B}$.

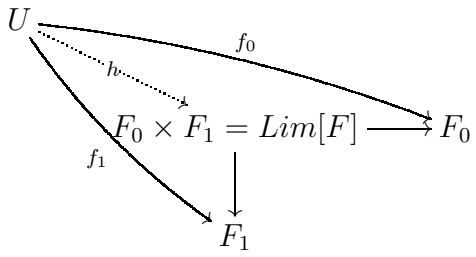


Figure 54: *Cartesian product as a limit*

Example 14. Terminal as a limit

Terminal object can be represented as a limit of an empty diagram, that is, a functor $\mathbb{0} \to \mathcal{C}$. What does it consist of? It consists of nothing; so its limit must be an object to which there is a unique arrow from any other object

$$U \dashrightarrow 1 = Lim[T]$$

Figure 55: *Terminal object as a limit*

Example 15. 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 "Parallel Pair" must be a good choice.

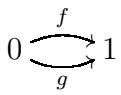


Figure 56: *Category "Parallel Pair"*

A functor F from "Parallel Pair" 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 exactly an equalizer of the parallel pair (F_0, F_1).

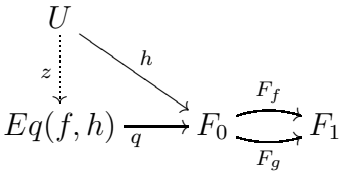


Figure 57: *Equalizer as a limit*

Conclusion

This ends chapter 12, and congratulations! This was the hardest part of the book; the rest will be much easier.

Chapter 13. Cartesian Closed Categories

Introduction

Out of all adjoint functor pairs, one pair is specifically useful in computing: it is a generalization of the adjunction described by Yoneda Lemma (see ch.12). The computing interpretation of Yoneda lemma is known as *currying-uncurrying*.

Of course, the lemma in ch.12 was only formulated for $\mathcal{S}et$; we need to have at least a Cartesian product defined for every pair of objects. Here's a formal definition:

Definition. Cartesian Closed Category

A category $\mathcal{C}$ is called Cartesian closed, 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* $\_^A$ $\_ \times A \vdash \_^A$;

We could equivalently replace the requirement of finite products with the requirement of having a terminal object and products for any two objects.

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's an object, and its internal structure and relations with other objects depend on the category $\mathcal{C}$.

If we talk about $\mathcal{S}et$ category, then A^B is the set of functions from B to A, by Yoneda lemma.

How does it look like in other categories? If we take $\mathcal{F}inVect$, the category of finite-dimensional vector spaces and linear transformations (one can think of them as just matrices), then A^B is the vector space of transformations (matrices) $B \to A$; it dimension is $dim(A)^{dim(B)}$.

Now an example of a Cartesian closed category that has nothing to do with $\mathcal{S}et$ - a Heyting algebra H (see ch.5). We know what 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 $x \leqslant y \to (a \wedge x) \leqslant (b \wedge y)$. Similarly, implication, $a \to \_$, is an endofunctor in our category H. (Please note that the character "$\to$" used here has two meanings,, one for implication inside H, another for a functor defined on H, that is, an arrow in $\mathcal{C}at$.

These two functors are adjoint, $(\_ \wedge a) \vdash (a \to \_)$.

We also need a terminal object; $\bot$ is one.

So, as you see, any Heyting algebra H is a Cartesian closed category.

Features of CCC

"CCC" is a popular abbreviation for Cartesian closed categories.

Properties of product

- 1 is neutral for product: $A \times 1 \cong A$ (the relation $\cong$ means 'isomorphic'). Why? Because Cartesian product is defined up to an isomorphism, and A is as good candidate for $A \times 1$ as any other.
- $A \times B \cong B \times A$. Why? Because *swap* that maps $A \times B$ to $B \times A$ is an isomorphism: $B \times A$ is a good a product as $A \times B$, and all products are isomorphic.
- Product is 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 another.

Properties of exponential

The following properties do not need 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 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 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 products properties:

- 0 is neutral for sum: $A + 0 \cong A$.
- $A + B \cong B + A$.
- Sum is associative, up to an isomorphism: $A + (B + C)$ is isomorphic to $(A + B) + C$ - for the same reason, each one can replace another.

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 $(f, g) : U \to (A^B \times A^C)$. But such a pair corresponds to a pair $(f' : U \times B \to A, g' : U \times C \to A)$, which, by the definition of 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 Curry-Howard-Lambek correspondence, they are equivalent to typed lambda calculus. That is, each typed lambda calculus defines a CCC, and each CCC defines a typed lambda calculus; objects in a CCC correspond to types in lambda. Lambda calculus is not a part of this book, though, and so there will be no further discussion of Curry-Howard-Lambek correspondence.

Chapter 14. Monads

Main Ideas

We 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 has a `map(f):F[T]=>F[U]`, defined 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 `(f compose g): A=>C`.

A monad, in a programming language, 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 the type `T`); second, the function of forming a singleton list, `u(t) = List(t)`, is the unit transformation, and third, the function `flatten:List[List[T]] => List[T]` is the multiplication.

A monad has to 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 we are dealing with `List[T]`), or first build a singleton, and then map it via `f`. This means that unit `u` is a natural transformation (see Ch. 12).
- For an `f:T=>U`, we can provide two paths from `F[F[T]]` to `F[U]`. The first path consists of first 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]`; the composition is

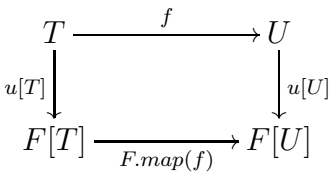


Figure 58: *Monad unit is a natural transformation*

`m[U] compose F[U].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; this means that `m` is a natural transformation

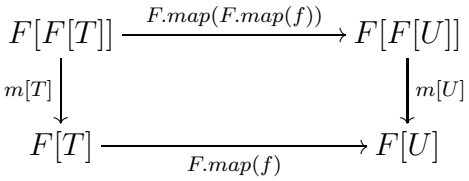


Figure 59: *Multiplication is a natural transformation*

- `u` is neutral with regards to multiplication:
  `m[T] compose map(u[T]) == m[T] compose u[F[T]] == id[F[T]]`.

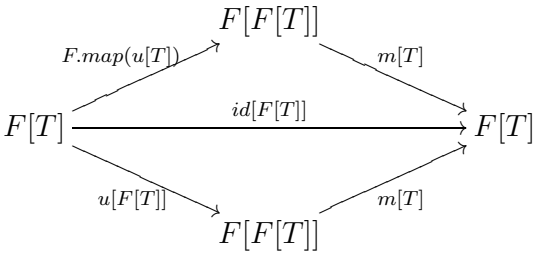


Figure 60: *u is a unit for multiplication m*

- Multiplication is associative:

These properties may be hard to assert for every specific monad, but if we get back to the example of lists, they are obvious. For instance, take associativity: we have a `list` of `list`s of `list`s; and the property says that when we flatten it to a `list`, we can go either way, first flatten externally, then again, or do flatten internally, on each element, and then flatten the result. Look at the following example: the `list` of `list`s of `list`s

```
List(List(List("a11", "a12"), List("a21", "a22")),
List(List("b11", "b12"), List("b21", "b22")))
```

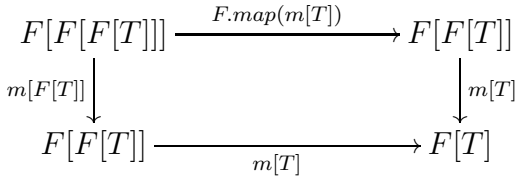


Figure 61: *Associativity of multiplication*

can be first transformed to the list of four lists of strings:

```
List(List("a11", "a12"), List("a21", "a22"),
     List("b11", "b12"), List("b21", "b22"))
```

and then to the list of eight strings:

```
List("a11", "a12", "a21", "a22", "b11", "b12", "b21", "b22")
```

or, the flattening can be done inside first, via

```
List(List("a11", "a12", "a21", "a22"),
List("b11", "b12", "b21", "b22"))
```

The axiom of associativity says that it does not matter.

Formal Details

Definition

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 is explained in the diagrams below.

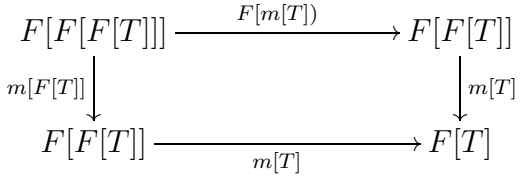


Figure 62: *Associativity of multiplication*

Examples of Monads

Example 2. Identity Functor. This is the simplest imaginable monad: identity functor; so both unit and multiplication are identity natural transformations $id_{Id_C} : Id_C \to Id_C$.

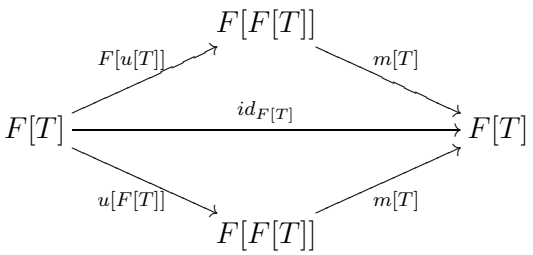


Figure 63: *Neutrality of unit*

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?

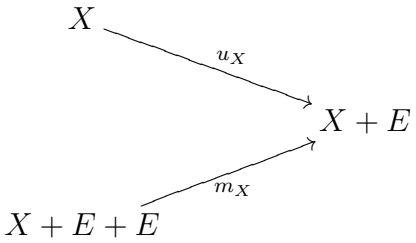


Figure 64: *Monad "$\_ + E$"*

Start with multiplication. We will need $X + A + A \to X + A$ for all X; this is obviously the same as having an arrow $A + A \to A$, which is defined on each component A as its identity. So we have multiplication; but how about unit? We will need $X \to X + A$ for each X ; 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 very different kinds of associativities.

This monad is also known as *Exception Monad* - meaning that E stands for an object of exceptions.

As a special case, of such a monad, take $A = 1$; we add a terminal object, also known as "point". What we get, a functor $Z \mapsto Z + 1$, is a known *Maybe* (also known as *Option*) monad.

Example 4. Functor $X \times A$. In a category that has Cartesian products, a functor $X \mapsto X \times A$ may have a monoidal structure, but for that we would need to define a unit $X \to X \times A$ for every X, and a multiplication $X \times A \times A \to X \times A$.

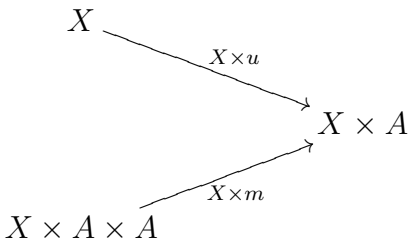


Figure 65: *Monad "_ × A"*

In the case 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 exactly that (A, u, m) is a monoid in our category.

Example 5. Functor `List`. 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 that we can take a list `L`, turn it into a list of singletons and then flatten, or we make a singleton list consisting of just `L`, and then flatten; in either case we get the same `L`.

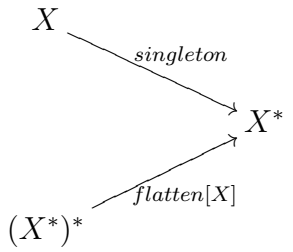


Figure 66: *Monad $List$ ("$X \mapsto X^*$")*

Categorically, a list functor (if exists), is defined like this: For a given object X, build $X \times X$, $A \times X \times X$, 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 disjoint union. So far we did not talk about infinite sums; assume they exist (they don't have to).

Informally (or rather over-formally) such a sum can be represented as $List[X] = 1/(1-X)$; since division is not known yet to the humans 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 $List[X] = 1 + X \times List[X]$.

Now it is not just legal, it also looks like a pretty elegant way to define a list: it is either an empty list, or an element of X followed by some list. Alternatively, we can look at it also as a fixed point Z of a functor $1 + X \times Z$. Again, its existence is not guaranteed. Do we need to describe multiplication and unit for this kind of functor? It's an exercise.

Example 6. Functor Pow**.** In a $\mathcal{S}et$ category, define the powerset endofunctor $Pow : X \mapsto Pow[X]$. It is not very trivial how does it map arrows. Given a function $f : X \to Y$, we need to define properly 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 $\bigcup : Pow[Pow[X]] \to Pow[X]$ is the multiplication for this monad.

Every Adjunction Gives a Monad

In Chapter 12 we introduced adjunctions - 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 $X \to Y$. Alternatively, a adjunction can be defined via two natural transformations, unit $\eta : id_{\mathcal{B}} \to GF$ and counit $\epsilon : FG \to id_{\mathcal{A}}$. These two transformations allow us to define a monad strutcure for the functor $M = GF$.

How does it happen? The unit u is the same as η; the multiplication $m[X] : GFGF[X] \to GF[X]$ is defined as $G[\epsilon[F[X]]]$.

We now can get back to examples of adjunctions in Chapter 12, and see what kind of monads they produce.

Example 7. A Monad for Partial Functions. For two categories, $\mathcal{S}et$ and $\mathcal{S}et_{\mathcal{P}art}$, we already defined the inclusion $\mathcal{S}et \hookrightarrow \mathcal{S}et_{\mathcal{P}art}$, and its right adjoint is the functor $Option : \mathcal{S}et_{\mathcal{P}art} \to \mathcal{S}et$ functor. The composition of the inclusion with this $Option$ functor gives us the familiar functor under the same name, $Option : \mathcal{S}et \to \mathcal{S}et$. It is known to be a monad; and the monadic structure is defined by the η and ϵ of the adjunction above.

Example 8. A Monad for Binary Relations. In example 7 of chapter 12 we see 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. The adjunction produces the same monad as described in example 5.

Example 9. Square Functor. Looking at example 9 of chapter 12, we see another adjunction, $\Pi \dashv \Delta$, where $\Delta[X] = X \times X$. The same 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 have then 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.

Summarizing, the multiplication takes two external components of $X \times X \times X \times X$ and ignores the two internal ones.

Example 10. Power of X. If we are in a Bicartesian Closed Category, the previous example would rather look like this: 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 $X^4 \to X^2$.

Now we dont have to limit ourselves by 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 Cartesian product, and the fact that 1 is a unit for product, 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^{Delta_Z}; and $X \to X^Z$ is just an adjoint to the projection $X \times Z \to Z$.

Example 11. Visitor Pattern is a Monad This example is due to Kris Nuttycombe, see https://logji.blogspot.com/2009/12/reader-monad-for-visitor-pattern.html

```
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 `VX` is a functor that has a property of a monad. `map` provides the functoriality of this parametric type.

`pure` plays the role of monadic unit, since it maps `T` to `V[T]`; and specifying `flatMap` gives us an alternative to specifying monadic multiplications, which can be defined as `flatten[T] = flatMap[V[T]](identity[V[T]])`.

Conclusion

You can experiment with a bunch of monads you know from programming, and try to figure out what kind of adjunction it is actually coming from. Or what adjunction can come from this monad. Note that there's not just one adjunction available; there's a whole category of adjoint pairs of functors for each monad.

Chapter 15. Monads: Algebras and Kleisli

While monads are already an interesting and useful concept (we'll get to the usefulness later in this chapter), they also give rise to other interesting concepts, *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, then action again, is 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 67: *Algebra action and monad multiplication*

This definition may look somewhat complicated, but it is actually not. Here's 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 68: *Free Algebra $M[X]$*

The fact that u is neutral for m, and that m is associative gives us the required properties.

Example 2. Algebra over $Option$ monad. As you saw before, $Option$ 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 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 languages, an instance of type `X`.

The choice is arbitrary, and, in the world of programming, opinions vary. On one hand, in languages that have `null`, we can just use `null` as a special instance of type `X`. On the other hand, Java philosophy suggests to use a so-called "null object", this is a "pattern" in Java. Again, the choice is arbitrary; and that's all we need for an algebra over *Option* monad to be properly defined.

The Java tradition to have a special "null value" for each used type make the whole programming style "algebraic" - having an algebra for each type. This helps Java programmers to deal with the fact of life that very rarely a function is total. To make a function total, they lift it, using a default value when there's no value.

If you see, for instance, that a program which could not detect the user name assumes that the user name is an empty string, or another program, unable to access a bank website, decides that your bank balance is `$0.00`, this is a typical abuse of algebras over *Option*.

Example 3. Algebra over $List$ monad. To have such an algebra with an underlying object X, we an arrow $fold : List[X] \to X$, compatible with singleton 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 components, and then flatten.

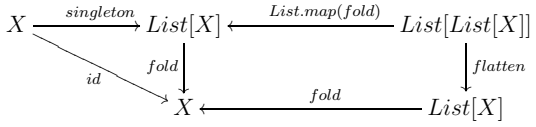


Figure 69: *Algebra over $List$ Monad*

We can prove that this feature is provided if we have an associative binary operation $op : X \times X \to X$; and we will also need an arrow from the empty list, $u : 1 \to X$. All this taken together gives us a monoid. It is the same thing, having a monoid or having an algebra over $List$ functor. Note that lists are also monoids, so they are also algebras over $List$ 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 be an algebra over this monad? We would need an action $X \times A \to X$ having the appropriate compatibility with monoidal operations. This is called an action of monoids. If we were in $\mathcal{S}ets$, we could talk about sets with the action of A over them, which amounts to monoid arrows $A \to X^X$; we 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. For that, 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:

$$\begin{array}{ccc} M[X] & \xrightarrow{\alpha_X} & X \\ {\scriptstyle M[f]}\downarrow & & \downarrow{\scriptstyle f} \\ M[Y] & \xrightarrow[\alpha_Y]{} & Y \end{array}$$

Figure 70: *Arrows between algebras*

Since the arrows are defined on the underlying objects, we can go ahead and see that a) identity arrows remain algebra arrows, and b) a composition of two algebra arrows is an algebra arrow. So we have a category; 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 $\mathcal{C}$. So we have a natural mapping of algebras to theier 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 a Functor

Out of all algebras that we can find for a given monad M, one kind is especially the easiest to have (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. 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: $Free \dashv Forget$.

When we have, in $\mathcal{C}^M$, an algebra arrow $f : Free[X] \to (M[Y] \to Y)$, it gives us a plain arrow $M[X] \to Y$ in $\mathcal{C}$, which, composed with the unit $u_X : X \to M[X]$, gives us an arrow $f' : X \to Y$.

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.

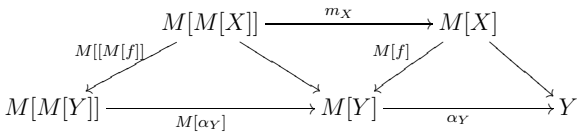


Figure 71: *Lifting an Arrow to an Algebra Arrow*

Although we already have some monads, it may make sense to demonstrate this adjunction in a couple of examples.

Example 5. *Option* **Monad and its Algebra Adjunction** As we saw before, algebras over *Option* monad in a category $\mathcal{C}$ are objects X with a point $1 \to X$; arrows between such algebras should be preserving these points.

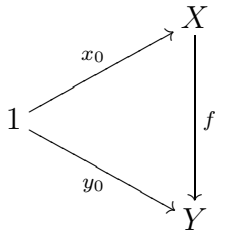


Figure 72: *An Arrow in $\mathcal{C}^{Option}$ Category*

In this specific case the forgetful functor will just 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

You saw above that a monad is produced by an adjoint pair; one of the candidates for such a pair is $Free \vdash Forget$, but there are others with the same effect. Here's one.

Definition. Kleisli Category

Given a monad M in a category $\mathcal{C}$, we can define it via *Kleisly Category* for that monad, denoted as $\mathcal{C}_M$. This category is build from $\mathcal{C}$ and M 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: $f : X \to M[Y]$ - this arrow is considered to be an arrow $f :\to Y$ in $\mathcal{C}_M$.

We should ask, if this is a category, where's the composition, and where's the identity?

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.

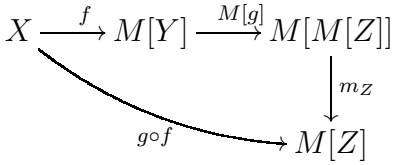


Figure 73: *Composition in a Kleisli Category*

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.

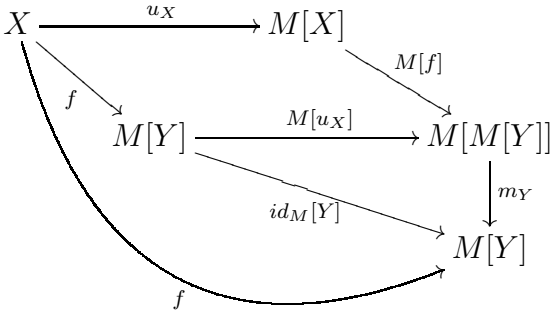


Figure 74: *Kleisli Identity is Neutral (on the left)*

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. It will be an exercise to prove that u_X is also the right neutral element for composition.

As you 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 have to check the functoriality, of course, but it's 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 $m_Y \circ M[f] : M[X] \to M[M[Y]] \to M[Y]$.

You see that 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 $\mathcal{S}et$. The 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 reflects better the facts of life - that not every function is total. In Scala, the code that works with the Kleisli category for `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]`. All three functions don't necessarily produce a good result, but they are combined to produce a reasonable answer in case of success, and no answer in case of any failure. That's how partial functions are supposed 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, $\mathcal{S}ets_{Pow}$, is an arrow $X \to Pow[Y]$, that is, a binary relation on $X \times Y$. Of course we have to prove at least 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 it 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 `String`s, the side effect can be thought of as logging.

Plurality of Adjunctions

We just saw that 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 one is an initial object, and another is a terminal object.

Bibliography

James L. Hein, *Discrete Structures, Logic, and Computability*, Jones & Bartlett Learning, Fourth edition, 2015.

Saunders Mac Lane, *Categories for the Working Mathematician*, Springer, Second edition, 1998.

Bartosz Milewski, *Category Theory for Programmers*, https://github.com/hmemcpy/milewski-ctfp-pdf/releases/download/v0.6.0/category-theory-for-programmers.pdf, 2018.

Benjamin C. Pierce, *Basic Category Theory for Computer Scientists (Foundations of Computing)* The MIT Press; 1 edition, 1991