Categories and Functors
(Lecture Notes for Midlands Graduate School, 2013)

Uday S. Reddy
The University of Birmingham

April 7, 2013

Contents

Chapter 1

Categories

A category is a universe of types.

The view point we take in this notes is that category theory is a "type theory." What is a "type?" Whenever we write a notation like $f : A \to B$, the A and the B are "types." And, f is a "function" of some kind, which "goes" from A to B. So, $A \to B$ is also a "type," but a slightly different kind of type from A and B. Call it as a "function type" for the time being.

The main point of writing these types, from a structural point of view, is that they allow us to talk about when we can plug functions together. If $f : A \to B$ and $g : B \to C$ then we can plug them together to form $g \circ f : A \to C$. The types serve as the shapes of the plugs so that we can only plug functions together when it makes sense to do so.

A traditional mathematician uses a notation like $f : A \to B$ in many different contexts. See the table below for some examples:

Subject	types	functions
Set theory	sets	set-theoretic functions
Group theory	groups	group homomorphisms
Linear algebra	vector spaces	linear transformations
Topology	topological spaces	continuous functions
Order theory	partially ordered sets	monotone functions
Domain theory	complete partially ordered sets	continuous functions

But a traditional mathematician might think that he is reusing the function notation in different contexts just for convenience; there is nothing fundamentally common to all these contexts. Category theory disagrees. It shows what is common to all these contexts and provides an axiomatic theory, so that we now have a single mathematical theory that forms the foundation for every kind of type that one might possibly envisage.

The "kind of type" just mentioned is referred to by the name *category*. The individual types of that kind are referred to as *objects*. This is just to avoid confusion with other possible meanings of "type" in mathematics, but feel free to think of "object" as being synonymous with "type." Nothing will ever go wrong. The functions that go between types are called *morphisms* or *arrows* (again just to avoid confusion with other possible meanings of "function" in mathematics, but feel free to think of "morphism" as being synonymous with "function." Nothing will ever go wrong.) Every kind of type mentioned in the above table is a distinct category. So, there is a category of sets, a category of groups and so on.

The plugging together of morphisms/functions is provided by postulating an operation called *composition*, which must be associative. We also postulate a family of units for composition, called *identity morphisms*, one for each type, $\mathrm{id}_A : A \to A$. These are units for composition

in the sense that $f \circ \mathrm{id}_A = f$ and $\mathrm{id}_B \circ f = f$ for every $f : A \to B$. That completes the axiomatization of categories.

One might think that identity morphisms form an afterthought, added just for convenience. But they are really central to category theory, as we will see in Chapter 4.

Notational issue. One of the irritations with mathematical notation is that functions run right to left: $g \circ f$ means doing f first and g later. When we do a set-theoretic theory, this might have some rationale, because we end up writing $g(f(x))$. But, in pure category theory, there is no need to stick to the backwards notation. So, we often use $f; g$ as composition in the "sequential" or "diagrammatic" order. It means the same as $g \circ f$ in the backwards notation. People also talk about composing functions on the "left" or "right," which are tied to the backwards notation. We will say "post-composing" and "pre-composing" instead. So, $f; g$ is obtained by "post-composing" g to f, or, equivalently "pre-composing" f to g. This terminilogy is neutral to the notation used.

1.1 Categories with and without "elements"

Prior to the advent of category theory, set theory was regarded as the canonical type theory. Sets are characterized by their *elements*. Functions between sets $f : A \to B$ must specify some way to associate, for each element of $x \in A$, an element of $f(x) \in B$. In contrast, in category theory, types are not presumed to have anything called "elements." (Specific categories might indeed have elements in their designated types, but the axiomatic theory of categories does not assume such things.) This makes the the type theory represented by categories immediately more general.

For an immediate example, we might consider a category where the objects are pairs of sets (A_1, A_2). A morphism between such objects might again be pairs $(f_1, f_2) : (A_1, A_2) \to (B_1, B_2)$ where each $f_i : A_i \to B_i$ is a function. Notice that the objects in this category *do not have "elements."* Since each object consists of *two* sets, which set is it that we are speaking of when we talk about elements? The question doesn't make sense. On the other hand, this structure is perfectly fine as a category. In essence, *category theory liberates type theory from the talk of elements.*

Definition 1 (Product category) If $\mathcal{C}_1$ and $\mathcal{C}_2$ are categories, their product category $\mathcal{C}_1 \times \mathcal{C}_2$ is defined as follows:

- Its objects are pairs of objects (A_1, A_2), where A_1 and A_2 are objects of $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively.
- Its morphisms are pairs of morphisms $(f_1, f_2) : (A_1, A_2) \to (B_1, B_2)$ where $f_1 : A_1 \to B_1$ is a morphism in $\mathcal{C}_1$ and $f_2 : A_2 \to B_2$ is a morphism in $\mathcal{C}_2$.
- The composition $(g_1, g_2) \circ (f_1, f_2)$ is defined as the morphism $(g_1 \circ f_1,\ g_2 \circ f_2)$ where $g_1 \circ f_1$ is composition in $\mathcal{C}_1$ and $g_2 \circ f_2$ is composition in $\mathcal{C}_2$.

Exercise 1 (basic) Verify that $\mathcal{C}_1 \times \mathcal{C}_2$ is a category, i.e., satisfies all the axioms of categories. What are its identity arrows?

Exercise 2 (basic) Define the concept of a "singleton" category $\mathbf{1}$, which can serve as the unit for the product construction above.

Exercise 3 (medium) If $\{\mathcal{C}_i\}_{i\in I}$ is an I-indexed family of categories (where I is a set), define its product category $\prod_{i\in I}\mathcal{C}_i$ and verify that it is a category. Can you think of other generalizations of the product category concept?

While category theory does not insist that objects should have elements, it still supports them whenever possible to do so. Here is the idea. Consider the category **Set**. It has a terminal object 1, the singleton set with a unique element $*$, which has the following special property: for any set A, the elements of A are one-to-one with functions of type $1 \to A$. (An element $x \in A$ corresponds to the constant function $\bar{x} : 1 \to A$ give by $\bar{x}(*) = x$ and every function $1 \to A$ is a function of this kind!) If $f : A \to B$ is a set-theoretic function, it maps elements $x \in A$ to unique elements $f(x) \in B$. We can represent this idea without the talk of elements: whenever $e : 1 \to A$ is a morphism, $f \circ e : 1 \to B$ is a morphism. If e corresponds to a conventional element $x \in A$, then $f \circ e$ corresponds to the conventional element $f(x) \in B$. So, we can represent the idea of "elements" in category theory without explicitly postulating them in the theory.

Definition 2 (Global elements) If a category $\mathcal{C}$ has a terminal object 1, then morphisms of type $1 \to A$ are called the *global elements* of A (also called the *points* of A).

Definition 3 (Generator) An object K in a category $\mathcal{C}$ is called a *generator* if, for all pairs of morphisms $f, g : A \to B$ between arbitrary objects A and B, $f = g$ iff $\forall e : K \to A.\, f \circ e = g \circ e$.

Definition 4 (Well-pointed category) A category $\mathcal{C}$ is said to be *well-pointed* if it has a terminal object 1, which is also a generator.

A well-pointed category is "set-like" even if its objects don't have a conventional notion of elements. For example, the category **Set** $\times$ **Set** has the terminal object $(1, 1)$ which is also the generator for the category. So, we can regard pairs of morphisms $(e_1, e_2) : (1, 1) \to (A_1, A_2)$ as the "elements" of (A_1, A_2).

Exercise 4 (basic) Verify that $(1, 1)$ is a generator in **Set** $\times$ **Set**.

Exercise 5 (medium) Consider the category **Rel**, the category whose objects are sets and morphisms $r : A \to B$ are binary relations $r \subseteq A \times B$. The composition of relations $r : A \to B$ and $s : B \to C$ is the usual relational composition and we have identity relations. Show that **Rel** is not well-pointed.

Exercise 6 (challenging) Which of the categories mentiond in the table at the beginning of this section have generators? Which are well-pointed?

One of the significant generalizations we obtain from category theory is that there are categories that are not well-pointed. Such categories turn out to be quite essential in programming language semantics.

Milner proved the so-called "context lemma" for the programming language PCF which says that two closed terms M_1 and M_2 of type $\tau_1 \to \tau_2 \to \cdots \to \tau_k \to \mathbf{int}$ are observationally equivalent if and only if, for all closed terms of $P_1, \ldots, P_k$ of types $\tau_1 \ldots, \tau_k$ respectively, the terms $M_1 P_1 \cdots P_k$ and $M_2 P_1 \cdots P_k$ have the same behaviour (either they both diverge or both evaluate to the same integer). Let us restate the property in a slightly more abstract form. Use the notation $\cong_\tau$ to mean observational equivalence in type τ. Then the context lemma says:

$$M_1 \cong_{\tau\to\sigma} M_2 \iff \forall \text{closed terms } P : \tau.\, M_1 P \cong_\sigma M_2 P$$

Now consider what this means denotationally. In a fully abstract model for PCF, two terms would have the same meaning if and only if they are observationally equivalent in PCF. The meaning of a closed term of type A would be a function of type $1 \to A$, where 1 is the terminal object in the fully abstract model. So, Milner's context lemma says essentially that a fully abstract model of PCF must be a well-pointed category.

On the other hand, if we extend PCF with programming features of computational interest, then the context lemma fails. For example, PCF + catch, which extends PCF by adding control operations called "catch" and "throw", does not satisfy the context lemma. Idealized Algol, which can be thought of as PCF + comm, adding a type of imperative commands, does not satisfy the context lemma. So, the fully abstract models of such computational languages cannot be well-pointed categories. The more general type theory offered by category theory is necessary to study the semantics of such languages.

Another data point is the simply typed lambda calculus itself. The simply typed lambda calculus can be regarded as an equational theory with the three equivalences α, β and η. Then we can ask the question whether the equational theory is complete for a certain class of models for the simply typed lambda calculus. Various notions of set-theoretic models have been proposed such as Henkin models and applicative structures. They correspond to well-pointed categories from our point of view. It turns out the equational theory of the simply typed lambda calculus is incomplete for all such classes of models. On the other hand, if we include non-well-pointed categories in the class of models, then the equational theory is *complete*. So, non-well-pointed categories are necessary for understanding something as foundational as the simply typed lambda calculus.

1.2 Categories as graphs

The preceding discussion shows that it is not a good idea to think of the types in a category as sets having elements. An alternative view is to think of a category as a directed graph, with the objects serving as the nodes of the graph and the morphism as the aarrows in the graph. Note that, whenever $f : A \to B$ and $g : B \to C$ are arrows, there is another arrow $g \circ f : A \to C$ that directly goes directly from A to C skipping over B. This makes a category into a very dense graph. So, it is better to think of only certain arrows of the category as forming the graph, and all the paths in the graph as being implicitly arrows.

A useful construction that arises from this view that of the *opposite category* (also called the *dual category*). In graph-theoretic view, the dual category is obtained by simply reversing the direction of all the arrows.

Definition 5 (dual category) If $\mathcal{C}$ is a category, $\mathcal{C}^{\mathrm{op}}$ is the category defined as follows:

- The objects are the same as those of $\mathcal{C}$.
- The morphisms are the reversed versions of the arrows of $\mathcal{C}$, i.e., for each arrow $f : A \to B$, there is an arrow $f^\sharp : B \to A$ in $\mathcal{C}^{\mathrm{op}}$.
- The composition of arrows $g^\sharp \circ f^\sharp$ in $\mathcal{C}^{\mathrm{op}}$ is nothing but $(f \circ g)^\sharp$.

Exercise 7 (basic) Verify that the dual category is a category. What are its identity arrows?

The availability of dual categories gives category theory an elegant notion of symmetry. Various concepts in category theory thus come in twins: product-coproduct, limit-colimit, initial-terminal objects etc. The coproduct in a category $\mathcal{C}$ is nothing but the product in $\mathcal{C}^{\mathrm{op}}$.

1.3 Categories as generalized monoids

There is another perspective on categories that is sometimes useful. A *monoid* is an algebraic structure that is similar to categories in many ways. A monoid is a triple $A = (A, \cdot, 1_A)$ where

- A is a set,
- "$\cdot$" is an associative binary operation on A, typically called "multiplication," and
- 1_A is a unit element in A that satisfies $a \cdot 1_A = a = 1_A \cdot a$.

Instances of monoids abound. In particular, the set of functions $[X \to X]$ on a set X forms a monoid with composition as the multiplication and the identity function as the unit. In any category $\mathcal{C}$, the set of morphisms $X \to X$, called "endomorphisms" of X, is a monoid in a similar way.

A monoid A may be thought of as a very simple category that has a single object, denoted $\star$, and all the elements of A becoming morphisms of type $\star \to \star$. Composition of these morphisms is the multiplication in the monoid and the identity morphism of $\star$ is the unit element. In this way, a monoid becomes a very simple case of a category.

The idea of categories may now be thought of as a generalization of this concept, where we have morphisms not only from an object to itself, but between different objects as well. These morphisms still have notion of "multiplication," if only when the types match, i.e., we can multiply b and a only when the target type of a and the source type of b are the same.

This idea leads to a project called "categorification," whereby familiar algebraic structures are viewed as categories. It sometimes leads to new insights which cannot be seen in the original algebraic setting.

1.4 Categorical logic

Recall the Curry-Howard isomorphism which says that there is a one-to-one correspondence between programs (terms in a type theory) and proofs in a constructive logic. Categories provide another formulation to add to this correspondence, so that we may now regard both programs and proofs as the arrows of a category.

This view point allows us to give a presentation of category theory as a logical formalism, which is referred to as *categorical logic*.[1]

The presence of identity arrows the operation of composition are presented as deduction rules:

$$\frac{}{\mathrm{id}_A : A \to A} \qquad \frac{f : A \to B \quad g : B \to C}{g \circ f : A \to C}$$

Note that it is entirely straightforward to read these rules as proof rules: id_A is a proof that A implies A; if f proves that A implies B and g proves that B implies C then $g \circ f$ is a proof that A implies C. The rule for id_A is called the "axiom" rule in proof theory and the rule for composition is called the "cut rule."

But, we also have equational axioms for categories. Do they have a proof-theoretic reading? Indeed, they do. They correspond to proof reduction or proof equivalence.

$$\frac{\dfrac{}{\mathrm{id}_A : A \to A} \quad f : A \to B}{f \circ \mathrm{id}_A : A \to B} \quad = \quad f : A \to B$$

[1] An excellent introduction to this topic is the text, Lambek and Scott: *Introduction to Higher Order Categorical Logic*, Cambridge University Press, 1986.

The morphism $f \circ \mathrm{id}_A$ represents a needlessly long proof, and we can reduce it by eliminating the unnecessary steps. The other equations can also be viewed in the same way.

Let us look at the correspondence with programming languages. A morphism is in an arbitrary category is similar to a term with a single free variable. The term formation rules for such terms are as follows:

$$\frac{}{x : A \vdash x : A} \qquad \frac{x : A \vdash M_x : B \quad y : B \vdash N_y : C}{x : A \vdash N_y[y := M_x] : C}$$

The rule on the left, called the *Variable* rule, allows us to treat a free variable as a term. The rule on the right is *Substitution*, which allows us to substitute a free variable y in a term by a complex term M_x. (We are assuming that the notation M_x, which would be normally written as $M[x]$, captures all the free occurrences of x in M.) These constructions satisfy the laws:

$$\begin{array}{rcl} y[y := M_x] & = & M_x \\ N[y := x] & = & N_x \\ (P_z[z := N_y])[y := M_x] & = & P_z[z := N_y[y := M_x]] \end{array}$$

It is clear that the Variable rule corresponds to the identity arrows, the Substitution rule to the composition of arrows, and three laws are the same as the axioms of categories. This allows us to interpret programming languages in categories with the required structure.

But this only works for terms with single free variables, you wonder. What of terms with multiple free variables? There are two possibilities. There is a natural generalization of normal category theory to *multicategories*, where the arrows have types of the form $A_1, \ldots, A_k \to B$, i.e., the source of each arrow is a sequence of objects instead of being a single object. They allow one to interpret terms with multiple free variables in a direct way. The second possibility is to use a product type constructor so that a typing context with multiple variables $x_1 : A_1, \ldots, x_k : A_k$ is treated as a single free variable of a product type $x : A_1 \times \cdots \times A_k$. The latter approach is more commonly followed. [2]

[2] For a full worked-out treatment of this aspect, see Gunter, *Semantics of Programmign Languages*, MIT Press; Lambek and Scott, *op cit.*

Chapter 2

Constructions in Categories

Since we maintain that category theory is a "type theory," we should be able to define within the theory type constructors that we are familiar with: products, sum types, function spaces, and perhaps other fancy types for data structures such as lists, trees etc. Moreover, since category theory is a *general* type theory, once we define these concepts, they will become applicable at once to *all* type theories such as the type theory of sets, that of groups, vector spaces, topological spaces, posets, cpo's etc. and for any other type theories that we might invent in future. All of this can be done. But we will only make a beginning in this chapter.

2.1 Warm-up definition: Product of sets

To give a gentle introduction to how these definitions go, we first examine how the definition of products works in the familiar set theory. We are told in high school that, for any two sets A and B, there is another set called their product, $A \times B$. We are also told that the elements of $A \times B$ are the so-called "ordered pairs," which are written as (x, y), where x represents an element of A and y an element of B.

That was presumably good enough for high school. But, if we look at the issue seriously, we discover that the high school teacher never told us what an "ordered pair" really is. Introducing a notation for writing ordered pairs is not the same as giving a definition! However, the notation was cleverly chosen, and it packs a load of information. Let us unravel it.

In the very idea of writing a notation such as (x, y), we discover the following features:

- From any ordered pair $p \in A \times B$, we can extract its first component, the x, which is an element of A, and the y, which is an element of B. Let us formalize this by saying that we have two functions:
$$\begin{aligned} \text{fst} &: A \times B \to A \\ \text{snd} &: A \times B \to B \end{aligned}$$
which allow us to extract the two components.

- Given two elements $x \in A$ and $y \in B$, there is one and only one ordered pair $p \in A \times B$ so that $\text{fst}(p) = x$ and $\text{snd}(p) = y$. (That is the essence of writing the notation such as (x, y). If there could be two ordered pairs that have x and y as their components, then which of them would have been called (x, y)? It wouldn't make sense.)

So, a formal statement of what $A \times B$ is would go like this:

there are functions $\mathrm{fst} : A \times B \to A$ *and* $\mathrm{snd} : A \times B \to B$ *satisfying this condition: for any given elements* $x \in A$ *and* $y \in B$*, there is a unique element* $p \in A \times B$ *such that* $\mathrm{fst}(p) = x$ *and* $\mathrm{snd}(p) = y$*.*

This unique element p *is written using the notation* (x, y)*.*

That is quite a complicated statement, with two levels of nested conditions and a universal quantifier and an existential quantifier. Our high school teacher was certainly wise enough not to lumber us with it at an early age!

But, if we think about it, we realize that all type constructors have to be essentially defined by statements like this. Since the type constructor builds a new type from the old, we will need to know how to extract information of old types from it. This is done by the fst and snd functions above. Secondly, we will need a way to build things of the new type. This is specified by the "for all ... there is unique ..." statement. These two constructions have to be *mutually inverse*.

Let us use the notation (x, y) for the unique element p of $A \times B$. Then the two conditions that it satisfies can be written as equations:

$$(\beta) \qquad \mathrm{fst}(x, y) = x \qquad \mathrm{snd}(x, y) = y$$

To express the uniqueness condition, we might say that, if there were some other pair p which had the same components as $\mathrm{fst}(x, y)$ and $\mathrm{snd}(x, y)$ then it has to be equal to (x, y). We can express this by an equation too:

$$(\eta) \qquad (\mathrm{fst}(x, y), \mathrm{snd}(x, y)) = (x, y)$$

The first batch of equations are called the "β laws" for products. The second equation is called the "η law" for products. Almost all type constructors come with these two batches of equational laws.

2.2 Type constructions in categories

To generalize the above warm-up exercise for categories, we must remember that, in categories, we don't assume that there are "elements" in types. Their role is taken over by morphisms. So, instead of talking about an element $x \in A$, we talk about a morphism that leads to A such as $f : X \to A$. With that generalization, the categorical definitions work very much in the same spirit.

One other subtlety is that we postulated two functions "fst" and "snd". But, obviously, we want to define the product $A \times B$ for every pair of types A and B. So, each such pair of types will need to have its own instances of "fst" and "snd". So, we will treat these functions *generically* in the following.

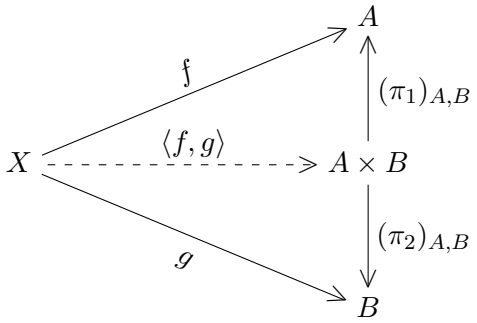
Products

A product of two objects A and B in a category $\mathcal{C}$ is an object (which we write as $A \times B$ for the time being) equipped with two morphisms:

$$(\pi_1)_{A,B} : A \times B \to A \qquad (\pi_2)_{A,B} : A \times B \to B$$

satisfying this condition: given any two morphisms $f : X \to A$ and $g : X \to B$, there is a unique morphism $h : X \to A \times B$ such that $(\pi_1)_{A,B} \circ h = f$ and $(\pi_2)_{A,B} \circ h = g$. This unique morphism h is written as $\langle f, g \rangle$.

We can express this diagrammatically as follows:



The two triangles must commute. They express the β laws. The dashed arrow labelled $\langle f, g\rangle$ expresses the condition is that it is a *unique* arrow that makes the diagram commute. Dashed arrows in our diagrams will *always* have this meaning.

We can express the various equational properties required in this definition as β and η laws:

$$\begin{array}{ll} (\beta) & \pi_1 \circ \langle f, g\rangle = f \\ & \pi_2 \circ \langle f, g\rangle = g \\ (\eta) & \langle \pi_1 \circ h,\, \pi_2 \circ h\rangle = h \end{array}$$

where $f : X \to A$, $g : X \to B$ and $h : X \to A \times B$. The β laws are evident from the definition. The η law is equivalent to the requirement that $\langle f, g\rangle$ is a *unique* arrow satisfying the β laws.

Uniqueness of products

One subtlety in the above definition is that we said "*a* product," not "the product." Can there be several products for A and B? Indeed so. In fact, if you can find one object $A \times B$ that qualifies as a product, you can immediately find another one, $B \times A$, that also qualifies as a product. To prove that $B \times A$ is a product of A and B, we need to first equip it with two projection morphisms. Let them be:

$$(\pi'_1)_{A,B} = (\pi_2)_{A,B} \qquad (\pi'_2)_{A,B} = (\pi_1)_{A,B}$$

Secondly, given $f : X \to A$ and $g : X \to B$, we need to show that there is a unique arrow of type $X \to B \times A$ such that the two equational conditions hold. Denote it by $\langle f, g\rangle'$ and define it as $\langle f, g\rangle' = swap_{A,B} \circ \langle f, g\rangle$ where $swap_{A,B} : A \times B \to B \times A$ is $\langle (\pi_2)_{A,B}, (\pi_1)_{A,B}\rangle$. It is easy to verify that $B \times A$ satisfies the conditions needed for a product.

However, notice that $A \times B$ and $B \times A$ are isomorphic. The isomorphism is $swap_{A,B} : A \times B \to B \times A$, which has an inverse $swap_{B,A} : B \times A \to A \times B$. By generalizing this example, it is possible to show that any two objects that qualify as a product of $A \times B$ will be necessarily isomorphic. So, we often say that the product is "unique up to isomorphism."

All constructions in category theory determine objects "up to isomorphism." That is in fact a fundamental aspect of the theory. It means that all types are "abstract." The way in which the material of the object is represented does not matter. Multiple representations that have the same external behaviour as seen from the outside are all equally good.

Another subtlety in saying "a product" is that nothing guarantees that there will be an object that qualifies as a product. Some categories have products and some don't. However, if a category has products, it must satisfy the definition given above.

Example 6 (Products in Set) The products in the category **Set** are the familiar set-theoretic products. Since our definition is a direct generalization of the set products, this fact should be more or less obvious.

Example 7 (Products in Poset) As another example, let us examine the products in the category **Poset**. The objects in **Poset** are partially ordered sets, i.e., sets with a partial order given for their elements, of the form $\mathbf{A} = (A, \sqsubseteq_A)$. Such posets arise almost everywhere in the theory of computer science in trying to model *partially defined* elements. For example, the set of partial functions $[\mathbb{N} \rightharpoonup \mathbb{N}]$ is a poset with the partial order given by

$$s \sqsubseteq t \iff (\forall n \in \mathbb{N}.\, s \text{ defined for } n \implies t \text{ defined for } n \wedge t(n) = s(n))$$

A morphism $f : \mathbf{A} \to \mathbf{B}$ in **Poset** is a "monotone" function, i.e., a function on the underlying sets A and B such that $x \sqsubseteq_A y \implies f(x) \sqsubseteq_B f(y)$. Now, what should be the product of $\mathbf{A} \times \mathbf{B}$? A little thought reveals that the underlying set of $\mathbf{A} \times \mathbf{B}$ should be the product of their underlying sets $A \times B$. The partial order on $\mathbf{A} \times \mathbf{B}$ should be chosen in such a way that π_1 and π_2 are monotone. The only possibility is to define:

$$(x, y) \sqsubseteq_{A \times B} (x', y') \iff x \sqsubseteq_A x' \wedge y \sqsubseteq_B y'$$

This makes π_1 and π_2 monotone. Next, we need to verify that $\langle f, g \rangle : \mathbf{X} \to \mathbf{A} \times \mathbf{B}$ is monotone whenever f and g are monotone, and also that the β and η laws hold for these particular choices of π_1, π_2 and $\langle f, g \rangle$. We leave these verifications to the reader.

Functor action for products

If $f : A \to A'$ and $g : B \to B'$ are morphisms, is there one of type $A \times B \to A' \times B'$? Indeed, there is, and we can derive it as follows:

$$\dfrac{\dfrac{\overline{\pi_1 : A \times B \to A} \quad f : A \to A'}{f \circ \pi_1 : A \times B \to A'} \qquad \dfrac{\overline{\pi_2 : A \times B \to B} \quad g : B \to B'}{g \circ \pi_2 : A \times B \to B'}}{\langle f \circ \pi_1,\, g \circ \pi_2 \rangle : A \times B \to A' \times B'}$$

Note that this construction allows us to "lift" morphisms on component types $f : A \to A'$ and $g : B \to B'$ to a morphism on the constructed type $A \times B \to A' \times B'$. It is the canonical operation on morphisms associated with the product type constructor. We denote it by $f \times g$.

We are deliberately overloading the notation here. The same symbol "$\times$" is used for the type constructor as well as the corresponding operation on morphisms that "lifts" the arrows on component types to an arrow on the constructed types. We will see in Chapter 3 that this is a general phenomenon in category theory. For now, the reader may enjoy verifying that the $\times$ operator on morphisms

- preserves composition of morphisms:

$$(f_2 \circ f_1) \times (g_2 \circ g_1) = (f_2 \times g_2) \circ (f_1 \times g_1)$$

- preserves identity morphisms:

$$\mathrm{id}_A \times \mathrm{id}_B = \mathrm{id}_{A \times B}$$

Terminal object

The definition of products can be generalized from *binary* products discussed above to arbitrary (finite or infinite) products. Given a family of objects $\{A_i\}_{i \in I}$, we can define the notion of $\Pi_{i \in I} A_i$. (Cf. Exercise 9.) Whether they exist or not in a particular category is a different question.

The case of the "nullary" product, i.e., product of an empty family of objects, has a special name, viz., the *terminal object*. It is denoted $\mathbf{1}$ and is unique up to isomorphism. The definition is as follows.

> *The terminal object in a category $\mathcal{C}$ is an object $\mathbf{1}$ such that there is a unique arrow $!_X : X \to \mathbf{1}$ from any object X of $\mathcal{C}$.*

For example, the terminal object in the category **Set** is any one-element set. We often write the lone element of $\mathbf{1}$ as "$\star$". In the category **Poset**, the terminal object is similarly a one-element poset. The only possibility for its partial order is to define $\star \sqsubseteq_{\mathbf{1}} \star$.

In programming languages, the type corresponding to the terminal object is often called the *unit* type. There is only an η law for the terminal object, which reads (somewhat strangely):

$$(\eta) \qquad !_X \;=\; h$$

for all $h : X \to \mathbf{1}$. There are no β laws because the terminal object does not have any "component" types.

Exercise 8 (basic) Verify that the definition of products for posets satisfies the categorical definition.

Exercise 9 (insightful) Generalize the definition of products from binary products to finite products, i.e., given a finite family of objects $\{A_i\}_{i \in I}$, give a definition for their product object $\prod_{i \in I} A_i$.

Exercise 10 (medium) Check if the category **Rel** has products (binary products as well as the terminal object). What are they? How about the category **Pfn**, whose objects are sets and morphisms are partial functions?

Exercise 11 (basic) Consider the category $\mathbf{Poset}_\perp$ of partially ordered sets that have least elements (which are denoted $\perp_A$). They are called "pointed posets." The morphisms of pointed posets are monotone functions that preserve the least elements, i.e., $f(\perp_A) = \perp_B$ for functions $f : A \to B$. Such functions are often called "strict" functions. Check if this category has binary products and a terminal object).

Exercise 12 (basic) Consider a variant of the above category **PPoset**, where the objects are pointed posets, but the morphisms are *all* monotone functions. (They need not preserve the least elements). Check if this category has products.

Exercise 13 (medium) Prove that the product of A and B is unique up to isomorphism.

Exercise 14 (insightful) Prove that the functor action on morphisms, $f \times g$, preserves the identity arrows and composition of arrows.

Exercise 15 (medium) Prove that the categorical product construction is commutative and associative (up to isomorphism):

$$\begin{aligned} A \times B &\cong B \times A \\ A \times (B \times C) &\cong (A \times B) \times C \end{aligned}$$

Prove also that the terminal object is its unit:

$$A \times \mathbf{1} \;\cong\; A \;\cong\; \mathbf{1} \times A$$

Exercise 16 (medium) Argue that, if a category $\mathcal{C}$ has binary products and a terminal object, then it has all finite products.

Exercise 17 (challenging) Which categories mentioned in Chapter 1 have products? What are they?

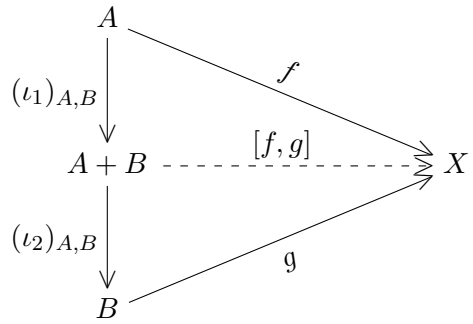
Coproducts (sum types)

You would recall from Chapter 1 that categories have duals. For every category $\mathcal{C}$ there is a dual category $\mathcal{C}^{\text{op}}$ which has the same objects and arrows as $\mathcal{C}$, but its arrow go in the opposite direction. So, we can consider the product concept in $\mathcal{C}^{\text{op}}$ and reflect the concept back to $\mathcal{C}$. The result is called *coproduct*. Here is its explicit definition.

A coproduct of two objects A and B in a category $\mathcal{C}$ is an object $A + B$ equipped with two morphisms:

$$(\iota_1)_{A,B} : A \to A + B \qquad\qquad (\iota_2)_{A,B} : B \to A + B$$

satisfying this condition: given any two morphisms $f : A \to X$ and $g : B \to X$, there is a unique morphism $h : A + B \to X$ such that $h \circ (\iota_1)_{A,B} = f$ and $h \circ (\iota_2)_{A,B} = g$. This unique morphism h is written as $[f, g]$. Diagrammatically:



Once again, we can express the equational properties required in this definition as β and η laws:

$$\begin{array}{ll} (\beta) & [f, g] \circ \iota_1 = f \\ & [f, g] \circ \iota_2 = g \\ (\eta) & [h \circ \iota_1,\ h \circ \iota_2] = h \end{array}$$

where $f : A \to X$, $g : A \to X$ and $h : A + B \to h$.

The coproduct in the category **Set** is called the "disjoint union." It means the "union" of A and B but treated in such a way that the elements of A and B are considered different from each other (the "disjointness"). More concretely, define:

$$\begin{array}{l} A + B = \{\, (1, x) \mid x \in A \,\} \cup \{\, (2, y) \mid y \in B \,\} \\ \iota_1(x) = (1, x) \\ \iota_2(y) = (2, y) \\ [f, g](z) = \begin{cases} f(x), & \text{if } z = (1, x) \\ g(y), & \text{if } z = (2, y) \end{cases} \end{array}$$

The functions ι_1 and ι_2 are called "injections." They allow us to treat an element of A (or B) as an element of $A + B$. The function $[f, g]$ is called the "case analysis" function. Given an element $z \in A + B$, it checks to see which case it falls into (the "A" case or the "B" case) and does the operation f and g accordingly.

As a simple example, the set of boolean values can be defined as $\mathbb{B} = \mathbf{1} + \mathbf{1}$. The element $\iota_1(\star)$ may be thought of as the boolean value "true" and the element $\iota_2(\star)$ as "false." The case analysis function is the "conditional" $\mathbb{B} \to X$ which produces $f(\star)$ if the boolean argument is true and $g(\star)$ if it is false.

In the category **Poset**, the coproduct $\mathbf{A} + \mathbf{B}$ is similar. It constitutes the "pasting together" of $\mathbf{A}$ and $\mathbf{B}$ along with their partial orders. In other words, all the injected elements of $\mathbf{A}$ in $\mathbf{A} + \mathbf{B}$ retain their partial order from $\mathbf{A}$ and similarly for $\mathbf{B}$. There is no additional partial

order relationship between the elements coming from **A** and from **B**. More formally, $\mathbf{A} + \mathbf{B} = (A + B, \sqsubseteq_{A+B})$ where

$$(i, z) \sqsubseteq (j, z') \iff i = j \wedge z \sqsubseteq z'$$

In programming languages, coproducts are called sum types and they are used with "constructor" symbols for denoting the injections (instead of ι_1 and ι_2) . For example, in Haskell, we find the standard type constructor Maybe defined by:

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

This means Maybe a is the coproduct $a + \mathbf{1}$ whose injections are the constructors:

```
Just : a -> Maybe a
Nothing : 1 -> Maybe a
```

The case analysis function $[f, g]$ would be written in this case as

```
\z -> case z of Maybe x => f x | Nothing => g ()
```

Initial objects

The dual concept of terminal objects is that of *initial objects*. The definition is:

> *The initial object in a category* $\mathcal{C}$ *is an object* $\mathbf{0}$ *such that there is a unique morphism* $\square_X : \mathbf{0} \to X$ *to any object* X*.*

The initial object in **Set** is the empty set. There is obviously a unique function from the empty set to another set, viz., the empty function! The initial object in **Poset** is the empty poset.

Exercise 18 (basic) Show that the concept of coproduct in a category $\mathcal{C}$ corresponds to the concept of product in the category $\mathcal{C}^{\mathrm{op}}$. That is, define the coproduct of A and B as the product object in $\mathcal{C}^{\mathrm{op}}$, and argue that it leads to the same definition as given above.

Exercise 19 (basic) Verify that the definition of coproducts for posets satisfies the categorical definition.

Exercise 20 (basic) Check if the category **Rel** has coproducts and an initial object. What are they? How about the category **Pfn**?

Exercise 21 (basic) Check if the categories $\mathbf{Poset}_\perp$ and **PPoset** have coproducts and an initial object.

Exercise 22 (insightful) Define a functor action on morphisms for the coproduct constructor, i.e., an operator $f + g$ that sends morphisms $f : A \to A'$ and $g : B \to B'$ to a morphism $A+B \to A'+B'$ in a canonical way. Check if it preserves the identity morphisms and composition of morphisms.

2.3 Cartesian closed categories

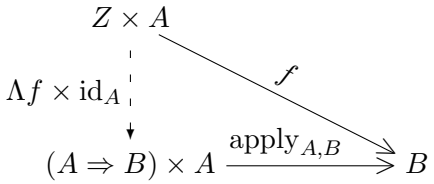
In the category **Set**, the collection of functions from A to B forms a set $[A \to B]$, i.e., another object in the category. In the category **Poset**, the collection of monotone functions from $\mathbf{A}$ to $\mathbf{B}$, also written as $[\mathbf{A} \to \mathbf{B}]$ can be given a partial order:

$$f \sqsubseteq_{A \to B} g \iff \forall x \in A.\, f(x) \sqsubseteq_B g(x)$$

Thus, the poset $([\mathbf{A} \to \mathbf{B}], \sqsubseteq_{A \to B})$ is another object in the category **Poset**. This kind of "function space" objects in categories are called *exponentials*. Here is the formal definition.

For objects A and B in a category $\mathcal{C}$, their *exponential* is an object $A \Rightarrow B$ equipped with a morphism $\text{apply}_{A,B} : (A \Rightarrow B) \times A \to B$ such that this condition holds: for any morphism $f : Z \times A \to B$, there is a unique morphism $h : Z \to (A \Rightarrow B)$ such that $\text{apply}_{A,B} \circ (h \times \text{id}_A) = f$ in the hom set $Z \times A \to B$. This unique morphism h is denoted Λf or (**curry** f).

Diagrammatically:



We can also express the requirements of exponentials as β and η laws:

$$\begin{array}{ll} (\beta) & \text{apply}_{A,B} \circ (\Lambda f \times \text{id}_A) = f \\ (\eta) & \Lambda(\text{apply}_{A,B} \circ (h \times \text{id}_A)) = h \end{array}$$

You may feel that this definition is not as "clean" as ithat of products and coproducts because of the additional baggage of $\times A$ lurking in the definition. A more symmetric version of the definition will be seen in Chapter 5.

A category that has all finite products and all exponentials is called a *cartesian closed category*. Having "all finite products" means that, for all finite collections of objects $\{A_i\}_{i \in I}$, there is a product object $\Pi_{i \in I} A_i$. As established in Exercise 9, this is equivalent to saying that the category has *binary* products and terminal objects. Having all exponentials means that for all pairs of objects A and B, there is an exponential $A \Rightarrow B$.

Notation and terminology Mathematicians use the notation B^A for the exponential instead of $A \Rightarrow B$. This makes sense because a "function" of type $A \Rightarrow B$ picks out an element of B for each element of A. So, the type of the function can be thought of as a (possibly infinite) product $B \times \cdots \times B$ with A-many copies of B, which is like taking the exponent of B to the power of A. Mathematicians also call $\text{apply}_{A,B}$ the "evaluation" map and denote it by $\varepsilon_{A,B}$.

Example 8 (Exponentials in Poset) Let us verify that the function space of posets $[\mathbf{A} \to \mathbf{B}]$ satisfies the categorical definition of exponentials.

- Define the function $\text{apply}_{A,B} : [\mathbf{A} \to \mathbf{B}] \times \mathbf{A} \to \mathbf{B}$ by $\text{apply}(h, x) = h(x)$. We need to verify that it is a monotone function. If $(h, x) \sqsubseteq (h', x')$, we have $h \sqsubseteq h'$ and $x \sqsubseteq x'$. Since h is a monotone function, this implies $h(x) \sqsubseteq h(x')$. By the definition of the partial order $h \sqsubseteq_{A \to B} h'$, we have $h(x') \sqsubseteq_B h'(x')$.
- Define $\Lambda f : \mathbf{Z} \to [\mathbf{A} \to \mathbf{B}]$ by $(\Lambda f)(z)(x) = f(z, x)$. We need to verify that Λf is a monotone function. If $z \sqsubseteq_Z z'$ then, since f is a monotone function, we know that $f(z, x) \sqsubseteq_B f(z', x)$ for all $x \in A$. Therefore $f(z, x) \sqsubseteq f(z', x)$ for all $x \in A$, i.e., $(\Lambda f)(z) \sqsubseteq_{A \to B} (\Lambda f)(z')$.

- Next, we need to verify the β and η laws of exponentials, which we leave to the reader.

Example 9 (Exponentials in Poset$_\perp$) This will actually be a *counterexample*. We want to check if the category **Poset**$_\perp$ of *pointed* posets and strict functions has exponentials. Let us try the monotone function space $[\mathbf{A} \to \mathbf{B}]$ of pointed posets $\mathbf{A}$ and $\mathbf{B}$ as a candidate. It needs to be a pointed poset, i.e., needs to have a least element. That is easy to establish: $\perp_{A\to B}$ is the function $x \mapsto \perp_B$.

- The function $\mathrm{apply}_{A,B} : [\mathbf{A} \to \mathbf{B}] \times \mathbf{A} \to \mathbf{B}$ is forced to be the same as in the previous example. But now we need to verify that it is strict as well. That means $\mathrm{apply}(\perp_{A\to B}, \perp_A) = \perp_B$. Since $\perp_{A\to B}$ maps all arguments to $\perp_B$, this checks out.
- The function $\Lambda f : \mathbf{Z} \to [\mathbf{A} \to \mathbf{B}]$ is forced to be the same as in the previous example. But we now need to verify that it is strict, i.e., $(\Lambda f)(\perp_Z) = \perp_{A\to B}$ for all strict functions $f : \mathbf{Z} \times \mathbf{A} \to \mathbf{B}$. Since $(\Lambda f)(\perp_Z)(x) = f(\perp_Z, x)$, this means $f(\perp_Z, x) = \perp_B$ for all $x \in A$. But this has no reason to be true. The strictness of f implies $f(\perp_Z, \perp_X) = \perp_B$. It does not imply $f(\perp_Z, x) = \perp_B$ for all $x \in A$.

We might think that we have been too liberal in allowing *all* monotone functions to be in the exponential. How about the set of *strict* functions, which we will denote by $[\mathbf{A} \multimap \mathbf{B}]$? Let us try it.

- The function $\mathrm{apply}_{A,B} : [\mathbf{A} \multimap \mathbf{B}] \times \mathbf{A} \to \mathbf{B}$ continues to be strict.
- The function $\Lambda f : \mathbf{Z} \to [\mathbf{A} \multimap \mathbf{B}]$ needs to be strict for strict functions $f : \mathbf{Z} \times \mathbf{A} \to \mathbf{B}$. But this has the same problem as above. The strictness of f does not imply $f(\perp_Z, x) = \perp_B$ for all $x \in A$.

It is possible to construct an argument to the effect that we have exhausted all possibilities. Hence **Poset**$_\perp$ does not have exponentials.

This example illustrates that we cannot take exponentials for granted. In fact, cartesian closed categories are quite rare in mathematics. However, they are central to computer science and many cartesian closed categories have been invented by computer scientists. An example appears is Exercise 12. We will see a large class of examples in terms of functor categories in Chapter 4.

Exercise 23 (basic) Show that the category **PPoset** has exponentials.

Exercise 24 (medium) Show that, if $\mathcal{C}_1$ and $\mathcal{C}_2$ are cartesian closed closed categories, then the product category $\mathcal{C}_1 \times \mathcal{C}_2$ is cartesian closed.

Exercise 25 (insightful) Define half of a functor action on morphisms for the $\Rightarrow$ type constructor. It should be an operator that you may write as $A \Rightarrow g$ which, given a morphism $g : B \to B'$, delivers a morphism $A \Rightarrow g : (A \Rightarrow B) \to (A \Rightarrow B')$. Define it in such a way that the operator preserves identities and composition at the position g, i.e., $A \Rightarrow \mathrm{id}_B = \mathrm{id}_{A\Rightarrow B}$ and $A \Rightarrow (g_2 \circ g_1) = (A \Rightarrow g_2) \circ (A \Rightarrow g_1)$.

Exercise 26 (insightful) Define the other half of the functor action on morphisms for the $\Rightarrow$ type constructor. It should be an operator that you may write as $f \Rightarrow B$, which given a morphism $f : A \to A'$, delivers a morphism $f \Rightarrow B : (A' \Rightarrow B) \to (A \Rightarrow B)$. Note that A' now appears to the *left* of "$\to$" and A to the right. Define it in such a way that the operator preserves identities and composition at the position f, i.e., $\mathrm{id}_A \Rightarrow B = \mathrm{id}_{A\Rightarrow B}$ and $(f_2 \circ f_1) \Rightarrow B = (f_1 \Rightarrow B) \circ (f_2 \Rightarrow B)$.

Typed lambda calculus

Cartesian closed categories are precisely the models of the typed lambda calculus. In this section, we examine this aspect.

A typed lambda calculus has a collection of types that is closed under function spaces, i.e., for all types τ and τ', $\tau \to \tau'$ is a type. For instance, we might start with a fixed collection of "base types" and inductively define:

- every base type β is a type, and
- if τ and τ' are types then $\tau \to \tau'$ is a type.

Terms of typed lambda calculus are given typings of the form:

$$x_1 : \tau_1, \ldots, x_n : \tau_n \rhd M : \sigma$$

where $x_1, \ldots, x_n$ are variables that can occur free in the term M, their types are assumed to be $\tau_1, \ldots, \tau_n$ respectively, and, under this assumption, M is a term of type σ. Here are the terms of the typed lambda calculus given in terms of their type rules.

$$\text{(Variable)} \quad \frac{}{x_1 : \tau_1, \ldots, x_n : \tau_n \rhd x_i : \tau_i}$$

$$\text{(Constant)} \quad \frac{}{\Gamma \rhd k : \sigma} \quad \text{(if } k \text{ is a constant of type } \sigma\text{)}$$

$$\text{(Abstraction)} \quad \frac{\Gamma, x : \sigma \rhd M : \sigma'}{\Gamma \rhd (\lambda x\colon \sigma.\, M) : \sigma \to \sigma'}$$

$$\text{(Application)} \quad \frac{\Gamma \rhd M : \sigma \to \sigma' \quad \Gamma \rhd N : \sigma}{\Gamma \rhd MN : \sigma'}$$

These terms are subject two equational laws:

$$(\beta) \quad (\lambda x\colon \sigma.\, M)\, N \;\equiv\; M[x := N]$$
$$(\eta) \quad \lambda x\colon \sigma.\, Mx \;\equiv\; M$$

We show that a typed lambda calculus can be given semantics in a cartesian closed category. That means that we pick

- a cartesian closed category $\mathcal{C}$,
- an interpretaion of every base type β as an object $[\![\beta]\!]$ of $\mathcal{C}$, and
- an interpretation of every constant $k : \sigma$ as a morphism $[\![k]\!] : \mathbf{1} \to [\![\sigma]\!]$.

Then we can extend the interpretation to all types and terms of the typed lambda calculus as follows:

- for all types τ in the calculus, we can assign an object $[\![\tau]\!]$ of $\mathcal{C}$ as its meaning, and
- for all terms $x_1 : \tau_1, \ldots, x_n : \tau_n \rhd M : \sigma$, we can assign a morphism

  $$[\![x_1 : \tau_1, \ldots, x_n : \tau_n \rhd M : \sigma]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \to [\![\sigma]\!]$$

  as its meaning,

such that the the β and η laws hold.

It is important to notice that meanings are given for terms in typing contexts. The same term in different typing contexts may have different meanings. For instance, the meaning of $x : \sigma \rhd x : \sigma$ will be the identity morphism $\mathrm{id}_{[\![\sigma]\!]} : [\![\sigma]\!] \to [\![\sigma]\!]$. However, the meaning of $x : \sigma, y : \sigma' \rhd x : \sigma$, which is the same term but with an extra variable y in the typing context, will be the projection morphism $\pi_1 : [\![\sigma]\!] \times [\![\sigma']\!] \to [\![\sigma]\!]$. Nevertheless, once we understand what is going on, it is reasonable to abbreviate the heavy notation $[\![x_1 : \tau_1, \ldots, x_n : \tau_n \rhd M : \sigma]\!]$ to just $[\![M]\!]$, which means the "meaning of M".

To a large extent, this exercise will seem quite straightforward because the formulation of typed lambda calculus and cartesian closed categories are very similar. Its main purpose is to show the correspondence between a *syntactic* system (the typed lambda calculus) and a *semantic* system (the theory of cartesian closed categories).

- The meaning of a variable term $x_1 : \tau_1, \ldots, x_n : \tau_n \rhd x_i : \tau_i$ is the projection morphism $\pi_i : [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \to [\![\tau_i]\!]$.

- The meaning of a constant term $\Gamma \rhd k : \sigma$ is the composite $[\![\Gamma]\!] \xrightarrow{!_{[\![\Gamma]\!]}} \mathbf{1} \xrightarrow{[\![k]\!]} [\![\sigma]\!]$.

- If an abstraction term derived using a step

$$\frac{\Gamma, x : \sigma \rhd M : \sigma'}{\Gamma \rhd \lambda x\colon \sigma.\, M : \sigma \to \sigma'}$$

  then, for the subterm, we have meaning of the form $[\![\Gamma, x : \sigma \rhd M : \sigma']\!] : [\![\Gamma]\!] \times [\![\sigma]\!] \to [\![\sigma']\!]$. We define:

$$[\![\Gamma \rhd \lambda x\colon \sigma.\, M : \sigma \to \sigma']\!] \;=\; \Lambda([\![\Gamma, x : \sigma \rhd M : \sigma']\!]) : [\![\Gamma]\!] \to ([\![\sigma]\!] \Rightarrow [\![\sigma']\!])$$

  or in abbreviated notation:

$$[\![\lambda x\colon \sigma.\, M]\!] \;=\; \Lambda([\![M]\!]) : [\![\Gamma]\!] \to ([\![\sigma]\!] \Rightarrow [\![\sigma']\!])$$

- If an application term is derived using the step:

$$\frac{\Gamma \rhd M : \sigma \to \sigma' \quad \Gamma \rhd N : \sigma}{\Gamma \rhd MN : \sigma'}$$

  then we have two morphisms $[\![M]\!] : [\![\Gamma]\!] \rhd ([\![\sigma]\!] \Rightarrow [\![\sigma']\!])$ and $[\![N]\!] : [\![\Gamma]\!] \rhd [\![\sigma]\!]$ for the two subterms. Then, we define

$$[\![MN]\!] = \mathrm{apply}_{[\![\sigma]\!],[\![\sigma']\!]} \circ \langle [\![M]\!], [\![N]\!] \rangle$$

The β and η laws of the typed lambda calculus now follow from the corresponding laws of cartesian closed categories.

Chapter 3

Functors

Functors are type constructors, ergo maps between categories.

Since we have said that a category represents a kind of type, maps between categories should correspond to type constructors or, equivalently, type expressions with type variables. In the simple case, there may be only one kind of type involved, i.e., we would be talking about type constructors where the argument types and the result type are of the same kind (such as **Set**, **Poset** or **DCPO**). In the more interesting cases of functors, there may be different kinds of types involved.

Recalling that categories are graphs with certain closure properties, we would expect that maps between categories would be first of all maps between graphs. So, if $F : \mathcal{C} \to \mathcal{D}$ is a functor, we expect it to map the objects (nodes) of $\mathcal{C}$ to objects of $\mathcal{D}$ and the arrows of $\mathcal{C}$ to arrows of $\mathcal{D}$ such that the source and target of the arrows are respected. More precisely, if F maps objects A and B of $\mathcal{C}$ to objects FA and FB of $\mathcal{D}$, then it should map an arrow $f : A \to B$ in $\mathcal{C}$ to an arrow $Ff : FA \to FB$ in $\mathcal{D}$. Secondly, since categories have the operations of composition and identities, we would expect them to be preserved, i.e., $F(g \circ f) = Fg \circ Ff$ and $F(\mathrm{id}_A) = \mathrm{id}_{FA}$.

Notational issues. Notice that we omit brackets around the arguments of functors, i.e., we write $F(A)$ as simply FA and $F(f)$ as simply Ff. Secondly, note that a functor really consists of two maps, one on objects and one on morphisms. If we wanted to be picky, we would say that a functor $F : \mathcal{C} \to \mathcal{D}$ is a pair $F = (F_{\mathrm{ob}}, F_{\mathrm{mor}})$ where F_{ob} maps the objects of $\mathcal{C}$ to objects of $\mathcal{D}$ and F_{mor} maps morphisms $f : A \to B$ of $\mathcal{C}$ to morphisms of type $F_{\mathrm{ob}}A \to F_{\mathrm{ob}}B$ in $\mathcal{D}$. We obtain a notational economy by "overloading" the same symbol F for both F_{ob} and F_{mor}. But there are really two maps here.

Example 10 (Lists) A simple example of a functor is $\mathbf{List} : \mathbf{Set} \to \mathbf{Set}$ that corresponds to the list type constructor in programming languages. The object part of the functor maps a set A to the set of lists over A, i.e., sequences of the form $[x_1, \ldots, x_n]$ where each x_i is an element of A. The morphism part of the functor maps a function $f : A \to B$ to the function normally written as map f in programming languages which sends a list $[x_1, \ldots, x_n]$ to $[f(x_1), \ldots, f(x_n)]$. In the categorical notation, the function map f will be written as $\mathbf{List}\ f : \mathbf{List}\ A \to \mathbf{List}\ B$.

Exercise 27 (basic) Verify that this definition of **List** constitutes a functor, i.e.,

$$\mathbf{List}\ (g \circ f) = (\mathbf{List}\ g) \circ (\mathbf{List}\ f) \quad \text{and} \quad \mathbf{List}\ \mathrm{id}_A = \mathrm{id}_{\mathbf{List}\ A}$$

Exercise 28 (medium) Consider an alternative definition of a list functor $\mathbf{ListR} : \mathbf{Set} \to \mathbf{Set}$ with

- object part the same as that of the **List** functor, i.e., maps objects A to **List** A, and
- the morphism part **ListR** $f : [x_1, x_2, \ldots, x_n] \mapsto [f(x_n), \ldots, f(x_2), f(x_1)]$.

Show that **ListR** is a functor.

Example 11 (Diagonal functor) The functor denoted $\Delta : \mathbf{Set} \to \mathbf{Set}$ has the object part that maps a set A to $A \times A$. The morphism part maps $f : A \to B$ to the function that may be written as $f \times f$, defined by $(f \times f)(x_1, x_2) = (f(x_1),\ f(x_2))$.

Example 12 (Powerset functor) The powerset functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ has the object part mapping a set A to its powerset $\mathcal{P}(A)$, which is in turn a set. The morphism part maps a function $f : A \to B$ to the "direct image" function of f, i.e., $\mathcal{P}f(u) = f[u] = \{\, f(x) \mid x \in u \,\}$.

Example 13 (Product and coproduct) The product type constructor on sets is a "binary" type operator, i.e., in writing $A \times B$, we are applying the type constructor $\times$ to *two* sets A and B. However, the pair of sets (A, B) can be regarded as the object of the product category $\mathbf{Set} \times \mathbf{Set}$. In this fashion, we can reduce "binary functors" to ordinary functors that operate on a product category. The product functor on sets is defined by:

$$\begin{array}{rcl} \times & : & \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set} \\ \times(A, B) & = & A \times B \\ \times(f, g) & = & \lambda(x, y) : A \times B.\ (f(x), g(y)) \end{array}$$

In practice, we write $\times(A, B)$ as simply $A \times B$ and $\times(f, g)$ as $f \times g$. The coproduct of sets has a parallel definition as a functor:

$$\begin{array}{rcl} + & : & \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set} \\ +(A, B) & = & A + B \\ +(f, g) & = & \lambda z : A + B.\ \mathbf{case}\ z\ \mathbf{of} \\ & & \qquad \mathrm{inl}(x) \Rightarrow \mathrm{inl}(f(x)) \\ & & \quad \mid\ \mathrm{inr}(y) \Rightarrow \mathrm{inr}(g(y)) \end{array}$$

Once again, we write $+(A, B)$ as simply $A + B$ and $+(f, g)$ as $f + g$.]

Example 14 (Covariant function space functor) Given two sets A and B, there is a set $[A \to B]$ consisting of all the functions from A to B. We will first note that the function space $[K \to B]$ for a fixed set K is a functor in the type variable B.

$$\begin{array}{rcl} [K \to -] & : & \mathbf{Set} \to \mathit{Set} \\ [K \to -](B) & = & K \to B \\ [K \to -](f) & = & \lambda h : [K \to B].\ h; f \end{array}$$

If $f : B \to B'$ is a function, then we are expecting $[K \to f]$ to be a function from the function space $[K \to B]$ to $[K \to B']$. $\lambda h.\, h; f$ gives such a function, which basically pre-composes f. We might also write it with the short hand notation $-; f$.

Contravariant functors We can also consider functors that reverse the direction of the arrows, called *contravariant functors*. A contravariant functor F from $\mathcal{C}$ to $\mathcal{D}$ has an object part similar to an ordinary functor, mapping the objects of $\mathcal{C}$ to objects of $\mathcal{D}$, but its morphism part reverses the direction of arrows, i.e., a morphism $f : A \to B$ in $\mathcal{C}$ is mapped to an arrow $Ff : FB \to FA$ in $\mathcal{D}$. The preservation of composition means that the composite $A \xrightarrow{f} B \xrightarrow{g} C$ should get mapped to $FC \xrightarrow{Fg} FB \xrightarrow{Ff} FA$. In words, the requirements are:

$$\begin{array}{rcl} F(g \circ f) & = & Ff \circ Fg \\ F(\mathrm{id}_A) & = & \mathrm{id}_{FA} \end{array}$$

The case of contravariant functors can be reduced to that of ordinary functors using the notion of the dual category. A "contravariant functor" from $\mathcal{C}$ to $\mathcal{D}$ can be regarded as an ordinary functor from $\mathcal{C}^{\mathrm{op}}$ to $\mathcal{D}$. If $f : A \to B$ is an arrow of $\mathcal{C}$, there is a corresponding arrow $f^\sharp : B \to A$ in $\mathcal{C}^{\mathrm{op}}$. So, an ordinary functor $F' : \mathcal{C}^{\mathrm{op}} \to \mathcal{D}$ maps $f^\sharp : B \to A$ in $\mathcal{C}^{\mathrm{op}}$ to $F'f^\sharp : FB \to FA$ and composition is preserved in the normal way. In practice, we dispense with the additional notation of $\sharp$ and simply write $F : \mathcal{C}^{\mathrm{op}} \to \mathcal{D}$ to mean that F is a contravariant functor.

Example 15 (Contravariant powrset functor) The contravariant powerset functor $\overline{\mathcal{P}} : \mathbf{Set}^{\mathrm{op}} \to \mathbf{Set}$ is defined as follows.

- The object part is the same as that of the normal powerset functor: $\overline{\mathcal{P}}A = \mathcal{P}A$.
- The morphism part maps a function $f : A \to B$ to the "inverse image" function of f, $\overline{\mathcal{P}}f : \mathcal{P}B \to \mathcal{P}A$ given by

$$\overline{\mathcal{P}}f(v) = f^{-1}[v] = \{\, x \in A \mid f(x) \in v \,\}$$

Exercise 29 (medium) Verify that $\mathcal{P}$ is a contravariant functor.

Example 16 (Contravariant function space functor) This example is the twin of Example 14. Given a fixed set K, consider the function space $[A \to K]$ regarded as a type expression in the type variable A.

$$\begin{array}{rcl} [- \to K] & : & \mathbf{Set}^{\mathrm{op}} \to \mathbf{Set} \\ [- \to K](A) & = & [A \to K] \\ [- \to K](f) & = & \lambda h : [A \to K].\ f; h \end{array}$$

If $f : A' \to A$ is a function, then we are expecting $[f \to K]$ to be a function from $[A \to K]$ to $[A' \to K]$. $\lambda h.\ f; h$ is such a function, which basically post-composes f. We might also write it with the short hand notation $f; -$.

Exercise 30 (medium) "The category **Rel** is its own dual." Verify this fact by exhibiting an isomorphism $\mathbf{Rel}^{\mathrm{op}} \cong \mathbf{Rel}$. What does "isomorphism" mean? For a starter, you can argue that

- the objects of $\mathbf{Rel}^{\mathrm{op}}$ and **Rel** are in one-to-one-correspondence, and
- the morphisms $A \to B$ in $\mathbf{Rel}^{\mathrm{op}}$ are in one-to-one correspondence with morphisms $A \to B$ in **Rel**.

The more formal definition requires us to provide two functors $F : \mathbf{Rel}^{\mathrm{op}} \to \mathbf{Rel}$ and $G : \mathbf{Rel} \to \mathbf{Rel}^{\mathrm{op}}$ such that their composites are identity functors: $G \circ F = \mathrm{Id}_{\mathbf{Rel}^{\mathrm{op}}}$ and $F \circ G = \mathrm{Id}_{\mathbf{Rel}}$.

Mixed variance functors. The above example shows that the function space type constructor is contravariant in the first argument and the example 14 showed that it is covariant in the second argument. The use of the dual category allows us to combine the two results:

$$\begin{array}{rcl} [- \to -] & : & \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set} \\ [- \to -](A, B) & = & [A \to B] \\ [- \to -](f, g) & = & \lambda h : [A \to B].\ f; h; g \end{array}$$

So, given functions $f : A' \to A$ and $g : B \to B'$, we are expecting $[f \to g]$ to be a function from $[A \to B]$ to $[A' \to B']$. $\lambda h.\ f; h; g$ is such a function.

Notation. Whenever we have a functor of multiple arguments, such as $[- \to -]$ above, we have the option of forming functors by keeping one or more arguments fixed and varying the others. The special cases $[A \to -]$ and $[- \to B]$ are obtained in this way. In the place holder "$-$", we can put an object to obtain an object and a morphism to obtain a morphism. Putting an object in the place holder seems quite straightforward, e.g., $[A \to X]$ is obtained by putting X in the place holder of $[A \to -]$. However, putting a morphism in the place holder gives rise to a funny looking expression, e.g., $[A \to g]$. What can it mean? The normal interpretation is that the letter A stands for not only the object A, but also its identity arrow id_A. So, $[A \to g]$ is interpreted as $[\mathrm{id}_A \to g]$, which is nothing but $\lambda h.\ \mathrm{id}_A; h; g = \lambda h.\ h; g$.

3.1 Heterogeneous functors

All the examples of functors seen so far are somewhat special: their source category and target category are built from the same basic category such as **Set**. We have seen examples where the source category is not simply **Set** but a kind expression built from **Set**, such as $\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set}$ used for the function space functor. Nevertheless, the base category involved is still **Set** everywhere. We will call such functors "homogeneous functors" because there is a single underlying category involved. (This is not a technical notion, but it is useful pedagogical idea.) Most functors that arise in type theory are homogeneous in this sense.

When different categories are involved in the source and target of a functor, we will call it a "heterogeneous functor." We show some examples of such functors.

For a simple example, consider a functor $G : \mathbf{Poset} \to \mathbf{Set}$, that maps each poset $(A, \sqsubseteq_A)$ to its underlying set A. A monotone function $f : (A, \sqsubseteq_A) \to (B, \sqsubseteq_B)$ can be regarded as just an ordinary function $f : A \to B$ by forgetting the fact that it preserves the order. Clearly, G preserves the composition of monotone functions and the identities. Hence it is a functor. Such functors are called "forgetful functors" because the object part forgets some structure of the objects and the morphism part forgets the fact that the morphisms preserve that part of the structure.

Consider a functor $F : \mathbf{Set} \to \mathbf{Poset}$ that goes in the reverse direction. Every set A can be regarded as a discrete poset $(A, =_A)$ whose partial order is nothing but the equality relation of A. Then every function $f : A \to B$ may be regarded as a monotone function of type $FA \to FB$. Again, F preserves the composition of functions and identities.

Since we regard each category as representing a particular "type theory," heterogeneous functors of this kind give us ways to form bridges between different type theories. Such bridges abound in algebra. We look at some examples.

- A set A along with an associative binary operation "$\cdot$" is called a *semigroup*. We write a semigroup $(A, .)$ as just A, leaving the binary operation implicit. A semigroup morphism $f : A \to B$ is a function between the underlying sets that preserves the binary operation:

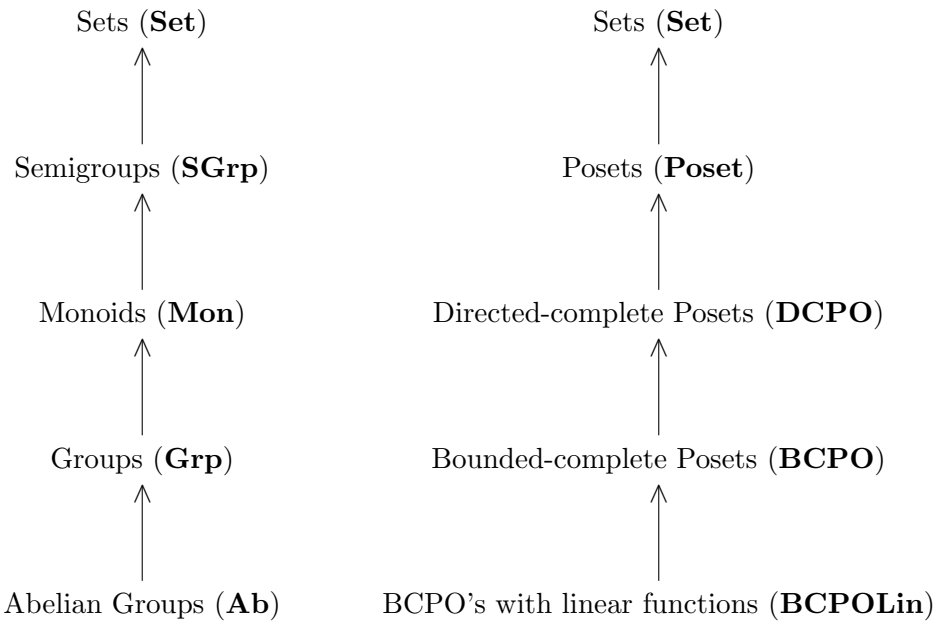


Figure 3.1: Inheritance of structure

$f(x \cdot y) = f(x) \cdot f(y)$. The usual notion of compositions and identities give a category **SGrp** of semigroups.

- If a semigroup A has unit element 1_A (that is, an element such that $x \cdot 1_A = x$ and $1_A \cdot x = x$ for all $x \in A$) then it is called a *monoid*. It is a theorem of semigroup theory that if a semigroup has a unit element then it is unique. Monoid homomorphisms are semigroup homomorphisms that also preserve the unit elements: $f(1_A) = 1_B$. Monoids and monoid homomorphism now form another category, denoted **Mon**.

- If a monoid A has inverses (that is, an element x^{-1}, for each $x \in A$, such that $x \cdot x^{-1} = 1_A$ and $x^{-1} \cdot x = 1_A$) then it is called a *group*. It is a theorem of group theory that, if an element x has an inverse, it is unique. Group homomorphisms are monoid homomorphisms that also preserve the inverses: $f(x^{-1}) = f(x)^{-1}$. Groups and group homomorphisms form a category denoted **Grp**.

- A group A is called a *commutative group* if its binary operation is commutative: $x \cdot y = y \cdot x$ for all $x, y \in A$. Commutative groups are commonly referred to as "abelian groups" in honor of the mathematician Niels Henrik Abel. It is conventional to write their binary operations as "+" rather than "$\cdot$". Abelian group homomorphisms are just group homomorphisms. This data gives rise to a category **Ab**.

In each step of the above narrative, we have added some "structure," i.e., some operations or axioms, giving rise to a natural inheritance hierarchy as shown in Figure 3.1. Category theory formalizes such inheritance hierarchies by exhibiting forgetful functors. There is an obvious forgetful functor **SGrp** $\to$ **Set** which forgets the associative operation and another one **Mon** $\to$ **SGrp** which forgets the units of the monoids and so on. Moreover, these functors compose. For example, the composition of the forgetful functors **Mon** $\to$ **SGrp** $\to$ **Set** forgets the associative operation as well as its units.

Similar inheritance hierarchies also arise in the structures used for programming language semantics.

- As mentioned before, **Poset** is the category of partial ordered sets with monotone functions as morphisms. There is a forgetful functor **Poset** $\to$ **Set**.

- A *directed set* in a poset A is a subset of A that "tends towards a limit." Formally, it is a subset $u \in A$ such that every pair of elements $x_1, x_2 \in u$ has an upper bound in u (that is, an element $z \in u$ such that $x_1 \sqsubseteq_A z$ and $x_2 \sqsubseteq_A z$). A *directed-complete poset* (dcpo) is one in which every directed set u has a least upper bound $\bigsqcup_A u$. (We often abbreviate "least upper bound" to "lub.")

  A morphism between dcpo's $f : A \to B$ must be a monotone function that also preserves the least upper bounds of directed sets: $f(\bigsqcup_A u) = \bigsqcup_B f[u]$. Such functions are said to be *continuous*. dcpo's and continuous functions form a category **DCPO**. The forgetful functor **DCPO** $\to$ **Poset** forgets the fact that the dcpo's have directed lub's and that their morphisms preserve the directed lub's.

- *Bounded-complete* posets are structures studied by Dana Scott for giving semantics to typed and untyped lambda calculi. They are dcpo's in which every finite bounded subset has a lub. (It then follows that every bounded subset — finite or infinite — has a lub.) The empty set has a lub as well, which is nothing but the least element of the poset, denoted $\bot_A$. Morphisms between bounded-complete posets are normally taken to be just continuous functions, i.e., they preserve the least upper bounds of directed subsets, but not necessarily the least upper bounds of bounded subsets. This gives us a category **BCPO**. The forgetful functor **BCPO** $\to$ **DCPO** forgets the fact that the bcpo's have bounded lub's.

- Morphisms that preserve the least upper bounds of *all* bounded subsets (not only directed subsets) are called *linear functions* (also called *additive functions*). Using these morphisms we obtain a more specialized category **BCPOLin**. The forgetful functor **BCPOLin** $\to$ **BCPO** forgets the fact the morphism are linear and regards them as just continuous functions.

Composition of functors It is easy to see that functors compose. If $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ are functors, then their composite $GF : \mathcal{C} \to \mathcal{E}$ is obtained by composing the object parts and the morphism parts respectively of F and G. There are also obvious identity functors $\mathrm{Id}_\mathcal{C} : \mathcal{C} \to \mathcal{C}$ for each category $\mathcal{C}$, and these identity functors form the units for composition $F \circ \mathrm{Id}_\mathcal{C} = F$ and $\mathrm{id}_\mathcal{D} \circ F = F$.

You would guess that this gives rise to a "category of categories." The objects of the big category are all categories, and the morphisms are functors. However, you might wonder if the category of all categories itself is an object of this category...

Multiple views of categories We consider situations where the same category can be viewed in two different ways and examine how functors help us to relate the multiple views. The category **Rel**, of sets and relations, is unlike most other categories we have looked at. Its morphisms are not functions. (Cf. Exercise 5). However, it is easy to view a relation $r : X \to Y$ as a function $r' : X \to \mathcal{P}Y$, given by $r'(x) = \{\, y \in Y \mid (x, y) \in r \,\}$. Such functions are typically thought of as "nondeterministic functions." We will define a category $\mathbf{Set}_\mathcal{P}$ whose morphisms $X \to Y$ are functions of type $X \to \mathcal{P}Y$ and whose composition mimics the composition in **Rel**. If $f : X \to \mathcal{P}Y$ and $g : Y \to \mathcal{P}Z$ are morphisms, their composition $g \circ f : X \to \mathcal{P}Z$ is defined as follows:

$$(g \circ f)(x) = \bigcup g[f(x)]$$

where $g[v]$ means the direct image of $v \subseteq Y$ under g. The identity morphism $\mathrm{id}_X : X \to \mathcal{P}X$ maps $x \mapsto \{x\}$. This gives us the category $\mathbf{Set}_{\mathcal{P}}$.

Exercise 31 (medium) Verify that $\mathbf{Set}_{\mathcal{P}}$ is a category, i.e., that the composition is associative and that the identity morphisms satisfy the identity axioms.

(This category is an example of general pattern in category theory, called the *Kliesli category of a monad*.)

Now, we define functors $F : \mathbf{Rel} \to \mathbf{Set}_P$ and $G : \mathbf{Set}_P \to \mathbf{Rel}$ that show the correspondence between the two categories:

$$\begin{array}{lrcl} F & : & X & \mapsto & X \\ F & : & (r : X \to Y) & \mapsto & (Fr : X \to \mathcal{P}Y) \text{ where } Fr(x) = \{\, y \in Y \mid (x,y) \in r \,\} \\ G & : & X & \mapsto & X \\ G & : & (f : X \to \mathcal{P}Y) & \mapsto & (Gf : X \to Y) \text{ where } Gf = \{\, (x,y) \mid y \in f(x) \,\} \end{array}$$

Exercise 32 (medium) Verify that F and G are functors, i.e., they preserve composition and the identity arrows.

The two functors F and G are mutually inverse, i.e., $G \circ F = \mathrm{Id}_{\mathbf{Rel}}$ and $F \circ G = \mathrm{Id}_{\mathbf{Set}_{\mathcal{P}}}$. In other words, **Rel** and $\mathbf{Set}_{\mathcal{P}}$ are *isomorphic*. they are the "same category" up to isomorphism.

Finally, we consider an example of a heterogeneous functor that maps between what appear to be completely different type theories. Recall that the powerset $\mathcal{P}X$ of a set X forms a *complete boolean algebra*, i.e., every subset $u \in \mathcal{P}X$ has a least upper bound $\bigcup u$ and a greatest lower bound $\bigcap u$, and every element $a \in \mathcal{P}X$ has a complement $\overline{a} \in \mathcal{P}X$. More generally, in an arbitrary complete boolean algebra, the least upper bound is denoted by $\bigsqcup u$, the greatest lower bound by $\bigsqcap u$ and the complement by a'. (I will not bother to state the axioms of the complete boolean algebras because we are just interested in the example of functors. Please feel free to look up the axioms in nCatLab.) A morphism of complete boolean algebras $f : A \to B$ preserves all of this structure: $f(\bigsqcup u) = \bigsqcup f[u]$, $f(\bigsqcap u) = \bigsqcap f[u]$ and $f(a') = f(a)'$. This data forms a category denoted **cBA**.

Our example functor $F : \mathbf{Rel} \to \mathbf{cBA}$ gives a relationship between **Rel** and the category of complete boolean algebras.

$$\begin{array}{rcl} X & \mapsto & \mathcal{P}X \\ (r : X \to Y) & \mapsto & Fr : \mathcal{P}X \to \mathcal{P}Y \text{ where } Fr(a) = \{\, y \mid x \in a \wedge (x,y) \in r \,\} \end{array}$$

We verify that F preserves composition. If $X \xrightarrow{r} Y \xrightarrow{s} Z$ is a composition in **Rel**, its image under F is $\mathcal{P}X \xrightarrow{Fr} \mathcal{P}Y \xrightarrow{Fs} \mathcal{P}Z$ is a composition in **cBA**. We calculate it as follows:

$$\begin{array}{rcl} Fs(Fr(a)) & = & Fs(\{\, y \mid x \in a \wedge (x,y) \in r \,\}) \\ & = & \{\, z \mid y \in \{\, y \mid x \in a \wedge (x,y) \in r \,\} \wedge (y,z) \in s \,\} \\ & = & \{\, z \mid \exists y.\, x \in a \wedge (x,y) \in r \wedge (y,z) \in s \,\} \\ & = & \{\, z \mid x \in a \wedge (x,z) \in (s \circ r) \,\} \\ & = & F(s \circ r)(a) \end{array}$$

We cannot find a functor $\mathbf{cBA} \to \mathbf{Rel}$ going in the reverse direction because an arbitrary complete boolean algebra has no reason to be similar to a powerset boolean algebra. However, we can consider the special case of complete boolean algebras that are "atomic." An element α of a poset with a least element $\bot$ is called an *atom* if it is immediately above $\bot$, i.e., $\bot \sqsubseteq x \sqsubseteq$

$\alpha \implies x = \bot \vee x = \alpha$. A complete boolean algebra is called *atomic* if every element $x \in X$ is the lub of a set of atoms. In a powerset boolean algebra $\mathcal{P}X$, the atoms are the singleton sets $\{x\}$ and all the sets can be expressed as the unions (lub's) of such singleton sets. In fact, every complete atomic boolean algebra is, up to isomorphism, the same as the powerset boolean algebra of all its atoms. (Because every element is the lub of a set of atoms, we can identify the element with the set of atoms that it is the lub of.) Such complete atomic boolean algebras form a subcategory **cABA** of **cBA** where the morphisms are still morphisms of complete boolean algebras, i.e., they don't treat the atoms in any special way.

We can now exhibit a functor $G : \mathbf{cABA} \to \mathbf{Rel}$ as follows:

$$\begin{array}{rcl} A & \mapsto & \text{Atoms}(A) \\ (f : A \to B) & \mapsto & (Gf : GA \to GB) \text{ where } (\alpha, \beta) \in Gf \iff \beta \sqsubseteq_B f(\alpha) \end{array}$$

The previous functor F as being of type $\mathbf{Rel} \to \mathbf{cBA}$ has its image within **cABA**. So, we can regard it as a functor of type $\mathbf{Rel} \to \mathbf{cABA}$. We can examine what relationship there is between F and G. Are they mutually inverse? Well, they are, almost. Consider the composite $\mathbf{Rel} \xrightarrow{F} \mathbf{cABA} \xrightarrow{G} \mathbf{Rel}$. On objects, it maps $X \mapsto \mathcal{P}X \mapsto \text{Atoms}(\mathcal{P}X)$. Since the atoms of $\mathcal{P}X$ are the singleton sets, which are one-to-one with X, we have an isomorphism $GF(X) \cong X$, but GF is not the identity. Going in the reverse direction, the composite $\mathbf{cABA} \xrightarrow{G} \mathbf{Rel} \xrightarrow{F} \mathbf{cABA}$ maps $A \mapsto \text{Atoms}(A) \mapsto \mathcal{P}(\text{Atoms}(A)$. Once again, since A is atomic, $A \cong \mathcal{P}(\text{Atoms}(A))$, but FG is not the identity.

This example shows that the notion of isomorphism is not all that useful in the context of categories. There is a more general notion called "equivalence," which serves the purpose better. Two functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ constitute an *equivalence* of categories $\mathcal{C}$ and $\mathcal{D}$ if there are natural isomorphisms $G \circ F \cong \text{Id}_{\mathcal{C}}$ and $F \circ G \cong \text{Id}_{\mathcal{D}}$. (The term "natural" here is a technical notion, which we examine in detail in the next section.) The fact that isomorphism is not a useful notion for categories and a more general notion is needed, is a deep feature of category theory, which you are invited to contemplate on.

3.2 The category of categories

We have said that the sources and targets in a function type notation are to be regarded as "types" and such types form categories. Now that we have started putting categories as sources and targets of functors, in a notation such as $F : \mathcal{C} \to \mathcal{D}$, we may think of categories themselves as "types," which form candidates to form a category.

The category of all categories **Cat** has objects that are categories and morphisms are functors. The composition of functors and the identity functors complete the picture.

If you are alert, you might wonder, "Wait a minute. How can we have a category of all categories? Wouldn't that lead to the same kind of paradoxes as the set of all sets?" Indeed, it would. To get out of trouble, we think of some kind of a cardinality restriction. We talk of "small" categories which have limited cardinality, and "large" categories which have no limit on cardinality. Then, we can think of a large category **Cat** whose objects are all small categories.

There are many ways of making such cardinality distinctions. The simplest approach, used most often by category theorists, is to regard "small" as meaning set-theoretic. The collection of objects in the category is a set. So, a small category is one whose objects form a set. A large category is one whose objects form a proper class.

Chapter 4

Natural transformations

Natural transformations are polymorphic functions, ergo maps between functors.

According to Eilenberg and Mac Lane, who invented category theory, "category" was defined to define "functor," and "functor" was defined to define "natural transformation."

What does this mean? If we look around in mathematics, we find structures such as "groups" which are naturally occurring (integers under addition, positive rationals under multiplication, permutation groups, symmetry groups and so on). We also find structures such as "topological spaces" which are not so natural. Rather, it is the maps between topological spaces, viz., continuous functions, that are naturally occurring. The definition of topological spaces was reverse engineered from the notion of continuous functions, so as to form the vehicle for the definition of continuity. The observation of Eilenberg and Mac Lane seems to mean that "natural transformations" are naturally occurring in mathematics and the notions of categories and functors were reverse engineered from them in two levels of abstraction, in order to make the definition of natural transformation possible. So, natural transformations form the *raison d'etre* for category theory.

What is so important about natural transformations that they warranted the entire development of the theory of categories? The answer is that they represent polymorphic functions. Recall that objects are like types and functors are like type constructors (or parameterized types). Polymorphic functions are maps between type constructors. To a Computer Scientist, they are an everyday phenomenon.

Consider a few examples of polymorphic functions:

$$\begin{array}{rcl} \mathit{reverse}[A] & : & \mathbf{List}\ A \to \mathbf{List}\ A \\ \mathit{append}[A] & : & \mathbf{List}\ A \times \mathbf{List}\ A \to \mathbf{List}\ A \\ \mathit{length}[A] & : & \mathbf{List}\ A \to \mathit{Int} \\ \mathit{map}[A,B] & : & [A \to B] \to [\mathbf{List}\ A \to \mathbf{List}\ B] \\ \mathit{foldr}[A,B] & : & [A \times B \to B] \times B \to [\mathbf{List}\ A \to B] \\ \mathit{traverse}[A] & : & \mathbf{Tree}\ A \to \mathbf{List}\ A \end{array}$$

In each of these cases, we have a sense that the functions listed are "generic" in the type parameters such as A and B, i.e., they will act the same way no matter what those types are. For example, *reverse* does the same thing no matter whether it is acting on a list of integers, a list strings, a list of list of strings, or a list of functions of some type. Similar intuitions hold for all the other examples.

In mathematics, we regard such polymorphic functions as being *families* of functions, parameterized by the type parameters. For example, *reverse* is a family of functions consisting of $\mathit{reverse}[\mathit{Int}]$, $\mathit{reverse}[\mathit{String}]$, $\mathit{reverse}[\mathit{Int} \Rightarrow \mathit{Int}]$ and so on. More succinctly, we write it as

$\{reverse[A]\}_A$ which is meant to mean that we are talking about a family of *reverse* functions, one for each type A in our universe of types. A particular instance of *reverse* is called a "component" of the polymorphic family.

A "universe" of types is a category, as we suggested at the outset. Let it be a category $\mathcal{C}$. "**List**" is a type constructor, or a functor, of type $\mathcal{C} \to \mathcal{C}$. The family *reverse* is then regarded as an abstract form of map **List** $\dot{\to}$ **List**. (Note that it is not a "function" in the set-theoretic sense. It consists of an *infinite* family of set-theoretic functions!) Such maps can be composed, e.g., *reverse* $\circ$ *reverse* and *length* $\circ$ *reverse* make sense. Composition works "component-wise" by composing the component functions of the family at each type A. There is an identity polymorphic function $\mathrm{id}_F : F \dot{\to} F$ with identity functions $\mathrm{id}_{FA} : FA \to FA$ as its components, and it forms the unit for the composition.

All that has been said so far is true for any families of functions. It does not depend on the idea that all the components of a polymorphic function act the "same way." To capture the idea of acting the same way, category theory proposes the condition of naturality. A polymorphic family such as *reverse* is said to be *natural* if, for *every* morphism $f : A \to A'$ in the category $\mathcal{C}$, the diagram on the right commutes:

$$\begin{array}{ccc} A & & \mathbf{List}\ A \xrightarrow{reverse[A]} \mathbf{List}\ A \\ \downarrow f & & \mathbf{List}\ f \downarrow \qquad\qquad \downarrow \mathbf{List}\ f \\ A' & & \mathbf{List}\ A' \xrightarrow{reverse[A']} \mathbf{List}\ A' \end{array}$$

This condition drastically cuts down the possible families that *reverse* can be. To see this, instantiate $A = Int$ and $A' = String$. Consider a particular list in **List** *Int* such as $[1, 2, \ldots, n]$. The component $reverse[Int]$ maps it to the reversed list $[n, \ldots, 2, 1]$. Now, consider an arbitrary list of strings $[s_1, s_2, \ldots, s_n]$. The diagram above must commute for all possible functions $f : Int \to String$. In particular, it must be commute for the function f that maps $1 \mapsto s_1$, $\ldots$, $n \mapsto s_n$. (We are not concerned with what it maps the other integers to.) We can see that **List** f maps $[1, 2, \ldots, n] \mapsto [s_1, s_2, \ldots, s_n]$ and it also maps $[n, \ldots, 2, 1] \mapsto [s_n, \ldots, s_2, s_1]$. So, the above commutative diagram says that $reverse[String]([s_1, \ldots, s_n])$ must *necessarily* be the list $[s_n, \ldots, s_1]$. So, $reverse[String]$ must do the same kind of reversal action that $reverse[Int]$ does. If we know what *reverse* does for a particular type such as *Int*, then we know what it does for all types. This is the essence of the naturality condition.

Definition 17 If $F, G : \mathcal{C} \to \mathcal{D}$ are two functors then a *natural transformation* $\eta : F \dot{\to} G$ is a family of morphisms $\eta_X : FX \to GX$ in $\mathcal{D}$, indexed by objects X of $\mathcal{C}$, such that for every morphism $f : X \to X'$ in $\mathcal{C}$, the diagram on the right commutes:

$$\begin{array}{ccc} X & & FX \xrightarrow{\eta_X} GX \\ \downarrow f & & Ff \downarrow \qquad\qquad \downarrow Gf \\ X' & & FX' \xrightarrow{\eta_{X'}} GX' \end{array}$$

(We use subscripting "η_X" as well as square bracket indexing "$\eta[X]$" for denoting the components of natural transformations. Note that the naturality of the family is indicated by putting a small dot above the arrow "$\dot{\to}$". Sometimes we also use "$\Rightarrow$" to denote the same.)

It is easy to see that the functors of type $\mathcal{C} \to \mathcal{D}$ form a category with natural transformations as morphisms. This is called a *functor category* and denoted $\mathcal{D}^{\mathcal{C}}$ or $[\mathcal{C}, \mathcal{D}]$.

Exercise 33 (medium) Consider natural transformations $\eta : \mathrm{Id} \dot{\to} \mathrm{Id}$ for functors of type $\mathbf{Set} \to \mathbf{Set}$. (That is, the components of η are of type $\eta_X : X \to X$ indexed by sets X). What can η be? (Hint: consider the naturality condition for morphisms of type $1 \to X$.) Can you answer the same question for other categories mentioned in Chapters 1 and 3?

Exercise 34 (medium) Consider natural transformations η with components $\eta[X, Y] : X \times Y \to X$ for functors of type $\mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$. What can η be? Consider the same question for the other categories mentioned in Sections 1 and 2.

Limitations of naturality

The conditions of naturality can be used with mixed variance functors. For example, the function *map* has the source type $[A \to B]$ and the target type $[\mathbf{List}\ A \to \mathbf{List}\ B]$, which are both functors of type $\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set}$ (contravariant in A and covariant in B). The naturality property for *map* is a commutative diagram as below:

$$\begin{array}{cc} A & B \\ \uparrow f & \downarrow g \\ A' & B' \end{array} \qquad \begin{array}{ccc} [A \to B] & \xrightarrow{map[A,B]} & [\mathbf{List}\ A \to \mathbf{List}\ B] \\ {\scriptstyle [f \to g]}\downarrow & & \downarrow{\scriptstyle [\mathbf{List}\ f \to \mathbf{List}\ g]} \\ [A' \to B'] & \xrightarrow{map[A',B']} & [\mathbf{List}\ A' \to \mathbf{List}\ B'] \end{array}$$

However, for *foldr*, the source type is $[A \times B \to B] \times B$, where the type parameter B occurs in *both covariant and contravariant positions*. Simply put, the source type $[A \times B \to B] \times B$ is *not a functor*. Natural transformations are only defined for functors. However, we still have the same intuitive idea that *foldr* acts the "same way" for all types A and B. But naturality is unable to capture this intuition. Two possible generalizations of naturality exist for such cases. One, called *diagonal naturality* or *dinaturality* treats the covariant and contravariant positions of B as different type parameters, e.g., $[A \times B^- \to B^+] \times B^+$ and then considers the special case where they are instantiated to the same type. This leads to a sophisticated theory but runs into the problem that dinatural transformations do not compose. The second solution is to use a separate class of "edges" (such as relations) to talk about the correspondences between types in the vertical dimension and disregard any notion of composition for such edges. This leads to the theory of *relational parametricity*.

4.1 Independence of type parameters

Consider a natural transformation with multiple type parameters:

$$\eta_{X,Y,Z} : F(X, Y, Z) \to G(X, Y, Z) : \mathcal{C}_1 \times \mathcal{C}_2 \times \mathcal{C}_3 \to \mathcal{D}$$

It is clearly a map from objects of $\mathcal{C}_1 \times \mathcal{C}_2 \times \mathcal{C}_3$, which are triples (X, Y, Z) of objects from $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ respectively, to appropriately typed morphisms of $\mathcal{D}$. The naturality condition for η says that, for all morphisms in the product category $\mathcal{C}_1 \times \mathcal{C}_2 \times \mathcal{C}_3$, the diagram on the right commutes:

$$\begin{array}{c} (X, Y, Z) \\ \downarrow{\scriptstyle (f,g,h)} \\ (X', Y', Z') \end{array} \qquad \begin{array}{ccc} F(X, Y, Z) & \xrightarrow{\eta_{X,Y,Z}} & G(X, Y, Z) \\ {\scriptstyle F(f,g,h)}\downarrow & & \downarrow{\scriptstyle G(f,g,h)} \\ F(X', Y', Z') & \xrightarrow{\eta_{X',Y',Z'}} & G(X', Y', Z') \end{array}$$

The independence principle is that, for η to be natural, it is enough for it to be natural in each type parameter (X, Y and Z) *independently* while the other parameters are kept "fixed." [1]

What does it mean to keep a type parameter fixed? Recall that an arrow $(X, Y, Z) \xrightarrow{(f,g,h)} (X', Y', Z')$ in the product category is really a collection of three arrows $X \xrightarrow{f} X'$, $Y \xrightarrow{g} Y'$ and $Z \xrightarrow{h} Z'$. Keeping a type parameter "fixed" means picking the identity arrow for that parameter. For example, $Y \xrightarrow{\mathrm{id}_Y} Y$ keeps Y fixed. So, the following diagram on the right states the property of being "natural in X" (while the other parameters are fixed):

$$\begin{array}{ccc} X & & F(X,Y,Z) \xrightarrow{\eta_{X,Y,Z}} G(X,Y,Z) \\ \downarrow f & F(f,\mathrm{id}_Y,\mathrm{id}_Z)\downarrow & \downarrow G(f,\mathrm{id}_Y,\mathrm{id}_Z) \\ X' & & F(X',Y,Z) \xrightarrow{\eta_{X',Y,Z}} G(X',Y,Z) \end{array}$$

More succinctly this diagram says that $\eta_1 : F(-, Y, Z) \dot{\to} G(-, Y, Z)$, obtained by setting $(\eta_1)_X = \eta_{X,Y,Z}$ is natural for every Y and Z.

Theorem 18 A transformation $\eta : F \to G$ for functors $F, G : \mathcal{C}_1 \times \mathcal{C}_2 \to \mathcal{D}$ is natural if and only if the corresponding restrictions $\eta_1 : F(-, Y) \to F(-, Y)$ and $\eta_2 : F(X, -) \to F(X, -)$ are natural for every X and Y.

Proof: It is obvious that the naturality of η_1 and η_2 follows from that of η. For the converse, decompose an arrow $(f, g) : (X, Y) \to (X', Y')$ in the product category as $(f, \mathrm{id}_Y); (\mathrm{id}_{X'}, g)$ and witness the diagram:

$$\begin{array}{ccc} (X,Y) & & F(X,Y) \xrightarrow{\eta_{X,Y}} G(X,Y) \\ \downarrow (f,\mathrm{id}_Y) & F(f,\mathrm{id}_Y)\downarrow & \downarrow G(f,\mathrm{id}_Y) \\ (X',Y) & & F(X',Y) \xrightarrow{\eta_{X',Y}} G(X',Y) \\ \downarrow (\mathrm{id}_{X'},g) & F(\mathrm{id}_{X'},g)\downarrow & \downarrow G(\mathrm{id}_{X'},g) \\ (X',Y') & & F(X',Y') \xrightarrow{\eta_{X',Y'}} G(X',Y') \end{array}$$

The upper rectangle commutes by the naturality of η_1 for a particular Y, and the lower rectangle commutes by the naturality of η_2 for a particular X'. So, the outer rectangle commutes. ■

[1] You might recall similar independence principle in structures used in denotational semantics. For example, a multiple argument function between posets is monotone if and only if it is monotone in each argument independently. However, such an independence principle fails in most other areas of mathematics. For example, a multiple argument linear transformation between vector spaces may be linear in each argument (in which case it is "multilinear") or it may be linear in all the arguments together. The two are different concepts. In category theory, on the other hand, "multinatural" is the same as "natural," "multifunctor" is the same as "functor" and so on. This is the luxury of cartesian closed categories, of which category theory is itself an instance.

Notation. Because identity morphisms leave type variables in type expressions unchanged, it is common to write id_X as simply X. So, the above diagram can be rewritten as:

$$\begin{array}{ccc}
(X,Y) & & F(X,Y) \xrightarrow{\eta_{X,Y}} G(X,Y) \\
\Big\downarrow{\scriptstyle (f,Y)} & & \Big\downarrow{\scriptstyle F(f,Y)} \qquad\qquad \Big\downarrow{\scriptstyle G(f,Y)} \\
(X',Y) & & F(X',Y) \xrightarrow{\eta_{X',Y}} G(X',Y) \\
\Big\downarrow{\scriptstyle (X',g)} & & \Big\downarrow{\scriptstyle F(X',g)} \qquad\qquad \Big\downarrow{\scriptstyle G(X',g)} \\
(X',Y') & & F(X',Y') \xrightarrow{\eta_{X',Y'}} G(X',Y')
\end{array}$$

To understand such diagrams, think of categories as *graphs* and diagrams as displaying pieces of the graph. An arrow either takes you from one object to another or leaves you in the same place. In the latter case, you just retain the type letter.

4.2 Compositions for natural transformations

If $\eta : F \dot{\to} G$ and $\tau : G \dot{\to} H$ are natural transformations, there is an evident composition $\tau \circ \eta : F \dot{\to} H$, which works component-wise. This is referred to as the "vertical composition" of natural transformations, with the intuition presented by this diagram:

$$\mathcal{C} \;\underset{\xrightarrow[H]{}}{\overset{\xrightarrow{F}}{\Downarrow \eta \atop \Downarrow \tau}}\; \mathcal{D}$$

Natural transformations also have another "horizontal composition," which is a bit more intricate. To make the idea intuitive, we first consider a couple of special cases.

- Consider the situation below:

$$\mathcal{C} \;\underset{\xrightarrow[G]{}}{\overset{\xrightarrow{F}}{\Downarrow \eta}}\; \mathcal{D} \xrightarrow{\qquad H \qquad} \mathcal{E}$$

  Since every component $\eta_X : FX \to GX$ is an arrow in $\mathcal{D}$, the morphism part of H sends it to an arrow $H\eta_X : HFX \to HGX$ in $\mathcal{E}$. These morphisms $H\eta_X$ may be regarded as the components of a natural transformation denoted $H\eta : HF \dot{\to} HG$. Why is it natural? Since the square on the left, below, commutes by virtue of η being a natural transformation, we can apply the functor H to every object and morphism in the diagram to obtain the commuting square on the right. (Note that this depends crucially on the fact that the functor H preserves composition!)

$$\begin{array}{ccc}
X & & FX \xrightarrow{\eta_X} GX \\
\Big\downarrow{\scriptstyle f} & & \Big\downarrow{\scriptstyle Ff} \qquad\qquad \Big\downarrow{\scriptstyle Gf} \\
X' & & FX' \xrightarrow{\eta_{X'}} GX'
\end{array}
\qquad\qquad
\begin{array}{ccc}
HFX & \xrightarrow{H\eta_X} & HGX \\
\Big\downarrow{\scriptstyle HFf} & & \Big\downarrow{\scriptstyle HGf} \\
HFX' & \xrightarrow{H\eta_{X'}} & HGX'
\end{array}$$

- Second, consider the symmetric situation below:

$$\mathcal{C} \xrightarrow{\quad F \quad} \mathcal{D} \underset{K}{\overset{H}{\Downarrow \tau}} \mathcal{E}$$

  Since τ has components $\tau_Y : HY \to KY$ in $\mathcal{E}$ for every object Y of $\mathcal{D}$, by setting $Y = FX$, we get morphisms $\tau_{FX} : HFX \to KFX$ for every object X of $\mathcal{C}$. These morphisms form the components of a natural transformation denoted $\tau F : HF \dot{\to} KF$. Why is it natural? Since τ has commuting square for every morphism $g : Y \to Y'$ in $\mathcal{D}$, it in particular has commuting squares for morphism of the form $Ff : FX \to FX'$.

Now the general case is the situation:

$$\mathcal{C} \underset{G}{\overset{F}{\Downarrow \eta}} \mathcal{D} \underset{K}{\overset{H}{\Downarrow \tau}} \mathcal{E}$$

where we would like to obtain a composite natural transformation $\tau\eta : HF \dot{\to} KG$. It can be obtained in two ways:

- $HF \xrightarrow{\tau F} KF \xrightarrow{K\eta} KG$.
- $HF \xrightarrow{H\eta} HG \xrightarrow{\tau G} KG$.

Naturality of η and τ implies that these two ways of composing the transformations are equal. That equal composite is denoted $\tau\eta$ or $\tau \cdot \eta$. The identity natural transformation $\mathrm{id} : \mathrm{Id}_{\mathcal{C}} \dot{\to} \mathrm{Id}_{\mathcal{C}}$ serves as the unit for this composition.

Exercise 35 (basic) Prove that the two composite natural transformations above are equal.

Exercise 36 (medium) Prove that horizontal composition is associative: $(\xi \cdot \tau) \cdot \eta = \xi \cdot (\tau \cdot \eta)$.

The two forms of composition for natural transformations satisfy an interesting "interchange law":

$$(\tau' \cdot \eta') \circ (\tau \cdot \eta) = (\tau' \circ \tau) \cdot (\eta' \circ \eta)$$

The category of all (small) categories **Cat** is now seen to have extra structure. It not only has objects (categories) and morphisms (functors), but also another layer of second level morphisms between those morphisms. More precisely, the morphisms (functors) between any pair of objects $[\mathcal{C}, \mathcal{D}]$ form categories themselves! The morphisms in these functors categories are natural transformations that satisfy the interchange law. Structures of this form are called *2-dimensional categories* or *2-categories*. So, **Cat** is a 2-category. The terminology used is:

0-cells = objects like categories
1-cells = morphisms between 0-cells
2-cells = morphisms between 1-cells

4.3 Hom functors and the Yoneda lemma

In the category **Set**, we have the function space functor $[- \to -] : \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set}$ which allows us to treat the collection of morphisms between two types as another object in the category. In arbitrary categories, the collection of morphisms may not necessarily be an object, but it is at least a set. We write $\hom_{\mathcal{C}}(A, B)$, $\hom(A, B)$ or $\mathcal{C}(A, B)$, for the *set* of morphisms between A and B in a category $\mathcal{C}$ (depending on the disambiguation needed in a context). There is an evident functor

$$\hom_{\mathcal{C}} : \mathcal{C}^{\mathrm{op}} \times \mathcal{C} \to \mathbf{Set}$$

with action on morphisms $f : A' \to A$ and $g : B \to B'$ being the function $\hom_{\mathcal{C}}(f, g) : \hom_{\mathcal{C}}(A, B) \to \hom_{\mathcal{C}}(A', B')$ that maps $h \mapsto f; h; g$. Hom-functors give us interesting (and important!) examples of the naturality condition at play.

Consider natural transformations of type

$$\phi : \hom_{\mathcal{C}}(-, A) \dot{\to} \hom_{\mathcal{C}}(-, B) : \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$$

What kind of natural transformations are possible? Given a morphism $h \in \hom_{\mathcal{C}}(X, A)$, ϕ_X must map it to a morphism in $\hom_{\mathcal{C}}(X, B)$. But naturality means that it must do so *generically* in X. So, what can it do generically? If there is a morphism $g : B \to B'$, it can post-compose g to obtain $h; g : X \to B$. This is certainly generic in X. So, this is one possibility. What else? Nothing, apparently.

Lemma 19 (Contravariant Yoneda lemma, special case) Natural transformations of type $\hom_{\mathcal{C}}(-, A) \dot{\to} \hom_{\mathcal{C}}(-, B)$ are bijective with morphisms $\hom_{\mathcal{C}}(A, B)$:

$$Nat(\hom_{\mathcal{C}}(-, A), \hom_{\mathcal{C}}(-, B)) \quad \cong \quad \hom_{\mathcal{C}}(A, B) \tag{4.1}$$

(We use the notation $Nat(F, G)$ for the set of natural transformations between two functors $F, G : \mathcal{C} \to \mathcal{D}$. It means the same as $\hom_{[\mathcal{C},\mathcal{D}]}(F, G)$. Note that in the current application, $\mathcal{D} = \mathbf{Set}$.)

To get better intuition for the lemma, you might consider the following type isomorphism in a polymorphic programming language:

$$\forall X.\, [[X \to A] \to [X \to B]] \quad \cong \quad [A \to B]$$

and use some operational intuition. If I have to produce a function of type $X \to B$ using a function of type $X \to A$ (but generically in X), what can I do? All I can do is to compose it with a function of type $A \to B$ that I might know about. Not much else.

Proof: To show the bijection (4.1), we must first show that for each $g \in \hom_{\mathcal{C}}(A, B)$ there is a natural transformation $\hom_{\mathcal{C}}(-, A) \dot{\to} \hom_{\mathcal{C}}(-, B)$. This is straightforward verification. Secondly, we must show that these are the *only* natural transformations there are. That is the interesting part.

Given a morphism $g \in \hom(A, B)$, we have the post-composition transformation $\hom(X, g) : h \mapsto h; g$. It is easily seen to be natural:[2]

$$\begin{array}{ccc} X & \hom(X, A) & \xrightarrow{\hom(X,g)} \hom(X, B) \\ \uparrow f & \hom(f,A)\downarrow & \downarrow \hom(f,B) \\ X' & \hom(X', A) & \xrightarrow{\hom(X',g)} \hom(X', B) \end{array} \qquad \begin{array}{ccc} h & \longmapsto & h; g \\ \downarrow & & \downarrow \\ f; h & \longmapsto & f; h; g \end{array}$$

[2]Note that the notation $\hom(f, A)$ means the same as $\hom(f, \mathrm{id}_A)$ and $\hom(X, g)$ means the same as $\hom(\mathrm{id}_X, g)$.

Conversely, given any natural transformation $\phi : \hom(-, A) \dot{\to} \hom(-, B)$ we can consider its action on $\mathrm{id}_A \in \hom(A, A)$. Denote this by $g_0 = \phi_A(\mathrm{id}_A)$. We can show that $\phi_X(h) = h; g_0$ always. How? Witness the commuting square:

$$\begin{array}{ccc} A & \hom(A,A) & \xrightarrow{\phi_A} \hom(A,B) \\ \uparrow h & \hom(h,A)\downarrow & \downarrow \hom(h,B) \\ X & \hom(X,A) & \xrightarrow{\phi_X} \hom(X,B) \end{array}$$

Chasing $\mathrm{id}_A \in \hom(A, A)$ to the right and down, we obtain $h; g_0$. Chasing it down and right, we obtain $\phi_X(\mathrm{id}_A; h) = \phi_X(h)$. These two must be equal. Hence, we see that every natural transformation of type $\hom(-, A) \dot{\to} \hom(-, B)$ is of the form $h \mapsto h; g_0$ for some morphism $g_0 \in \hom(A, B)$. ■

This is a deep result! The Yoneda lemma, in its multiple forms, is one of the most powerful tools in category theory. So you should take time to work through the proof and contemplate its meaning.

It turns out that the second functor in the isomorphism (4.1) does not need to be a hom-functor, even though it is intuitive to consider that case. Generalizing it to an arbitrary contravariant functor, we obtain the general Yoneda lemma.

Lemma 20 (Contravariant Yoneda lemma, general case) If $K : \mathcal{C}^{\mathrm{op}} \to \mathbf{Set}$ is any functor, then natural transformations of type $\hom(-, A) \dot{\to} K$ are bijective with the set $K(A)$.

$$Nat(\hom(-, A),\, K) \quad \cong \quad K(A) \tag{4.2}$$

There is an obvious dual of the lemma for the covariant case:

Lemma 21 (Covariant Yoneda lemma, general case) If $F : \mathcal{C} \to \mathbf{Set}$ is any functor, then natural transformations of type $\hom(A, -) \dot{\to} F$ are bijective with the set $F(A)$.

$$Nat(\hom(A, -),\, F) \quad \cong \quad F(A) \tag{4.3}$$

The proof is dual to the previous lemma, but you might write it down to get a feel for the naturality argument involved. As a corollary, we obtain the formula:

$$Nat(\hom(A, -),\, \hom(B, -)) \quad \cong \quad \hom(B, A) \tag{4.4}$$

Exercise 37 (insightful) Prove at least one of the Lemmas 20 and 21, preferably both.

4.4 Functor categories

As remarked earlier, the functors $\mathcal{C} \to \mathcal{D}$ or $\mathcal{C}^{\mathrm{op}} \to \mathcal{D}$ form categories with natural transformations as the morphisms. An important special case is obtained by setting $\mathcal{D} = \mathbf{Set}$, the category of sets and functions. Functor categories of the form $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ (or, equivalently, $[\mathcal{C}, \mathbf{Set}]$) are called *presheaves*.

A presheaf category has very strong properties. It is always *cartesian closed* (no matter what $\mathcal{C}$ we start with). So, it forms model of the simply typed lambda calculus. A presheaf category is also a *topos*. So, it forms a model of the intuitionistic logic!

Application: Meta-programming

To provide some intuition for the a category $[\mathcal{C}, \mathbf{Set}]$, we consider a simple example. Consider the category $\mathcal{C}$ that consists of just two objects and one arrow as follows:

$$\begin{array}{c} 0 \\ e\downarrow \\ 1 \end{array}$$

A functor $F : \mathcal{C} \to \mathbf{Set}$ gives us an image $\mathcal{C}$ in **Set**:

$$\begin{array}{c} F0 \\ Fe\downarrow \\ F1 \end{array}$$

Not that this diagram is in **Set**. So, $F0$ and $F1$ are just sets and Fe is just an ordinary function. So, every functor $[\mathcal{C}, \mathbf{Set}]$ is just a pair of sets with a chosen function between them.

We consider a programming language whose types can be interpreted as diagrams of this kind. This kind of a language is suitable for meta-programming. In such a language, we expect to be able to write programs that act on other programs to produce yet other programs. For example, in the Lisp programming language, one can put a quotation mark around an S-expression to treat it as data, apply other functions to construct new S-expressions, and finally use an `eval` operator to evaluate the resulting S-expressions. Types in such a programming language can be interpreted as objects $[\mathcal{C}, \mathbf{Set}]$. The set $F0$ denotes the set of expressions of some type; $F1$ denotes the set of values of the type; the function Fe is the `eval` operator that can evaluate expressions to values.

Every term in the programming language $x_1 \colon F_1, \ldots, x_n \colon F_n \vdash M : G$ has two levels of meaning. At the expression level, it maps n-tuples of expressions of type $F_1 0 \times \cdots \times F_n 0$ to expressions $G0$. At the value level, it maps n-tuples of values of type $F_1 1 \times \cdots \times F_n 1$ to values of type $G1$. The two levels of meaning must correspond in the sense that the following diagram must commute:

$$\begin{array}{ccc} F_1 0 \times \cdots \times F_n 0 & \xrightarrow{[\![M]\!]_0} & G0 \\ {\scriptstyle F_1 e \times \cdots \times F_n e}\downarrow & & \downarrow{\scriptstyle Ge} \\ F_1 1 \times \cdots \times F_n 1 & \xrightarrow{[\![M]\!]_1} & G1 \end{array}$$

Note that this is just the naturality square that says that $[\![M]\!]$ is a natural transformation. It just says that if we first evaluate some tuple of argument expressions and then compute the value-level meaning of M, we get the same results as applying the expression-level meaning of M and evaluating the resulting expression.

Application: Local variables

Following Kripke, semantic models of intuitionistic logic are thought of as "possible world" models. This is a useful viewpoint to take in understanding the functor category $[\mathcal{C}, \mathbf{Set}]$. The idea is that each object of $\mathcal{C}$ is a "possible world," i.e., a *context* in which the types and propositions can be interpreted. So, a type in this sense is not simply a set or something akin to a set, but rather it is an entire family of sets, one for each context X in $\mathcal{C}$. A morphism $f : X \to Y$ in $\mathcal{C}$ represents the idea that Y is a "future world" of X, i.e., obtained by some form of context switching which allows us to transition into a larger context. The morphism f gives

us information about the *way* in which Y is a future world of X. The action of a type F on the morphism f gives us a function $Ff : FX \to FY$, which tells us how to reinterpret all the values of FX as values in the larger context.

To make these ideas concrete, we given an interpretation of a simple imperative programming language in a functor category. Imperative programming languages are well-suited for such an interpretation because the meanings of all their types depend on the variables (or storage locations) that are available. The language has three basic types

- *Var*, which is the set of mutable variables available in the store,
- *Exp*, which is the type of "expressions" that read the state of the store and produce an integer, and
- *Comm*, which is the type of "commands" that read and write the variables in the store.

We will model stores as sets of storage locations numbered consecutively $\{0, 1, \ldots, K-1\}$. For convenience, the set $\{0, 1, \ldots, K-1\}$ is denoted by K, in the set-theoretic representation of natural numbers. If $K \leq L$, we can give an injection $f : K \to L$, which tells us how to reinterpret the locations in the store K as the locations in L. So, the category of worlds $\mathcal{C}$ has

- natural numbers $K = \{0, 1, \ldots, K-1\}$ as objects, and
- injections $f : K \to L$ as morphisms.

These will be our "possible worlds."

For each world K, we have the set of *states* for a store with K locations. This is denoted

$$\mathcal{S}K = [K \to \mathit{Int}]$$

We extend it to a *contravariant* functor by defining $\mathcal{S}(f : K \to L) : \mathcal{S}L \to \mathcal{S}K$ to be the map $s \mapsto \lambda i.\, s(f(i))$. In words, if s is a state of the larger store L, we obtain a state of the smaller store K by projecting it down to the locations in K.

For each world K, we also have a set of *state transformations*

$$\mathcal{T}K = [\mathcal{S}K \to \mathcal{S}K]$$

We turn $\mathcal{T}$ into *covariant* functor by mapping morphisms $f : K \to L$ into functions $\mathcal{T}f : \mathcal{T}K \to \mathcal{T}L$, which, intuitively, extend a state transformation for the small store K into one for the large store by mimicking the action of the transformation on the small store. To express the action cleanly, partition L as $L_0 + L'$ where $L_0 = L_0$ is the image of K in L and L' is the set of extra locations. This gives an isomorphism $\mathcal{S}L \cong \mathcal{S}L_0 \times \mathcal{S}L'$. If $t \in \mathcal{T}K$ is a state transformation then the corresponding transformation $\mathcal{T}f(t) \in \mathcal{T}L$ is obtained by:

$$\begin{array}{ccc} \mathcal{S}L & \xrightarrow{\cong} & \mathcal{S}L_0 \times \mathcal{S}L' \\ {\scriptstyle \mathcal{T}f(t)}\,\vdots & & \Big\downarrow {\scriptstyle t \times \mathrm{id}} \\ \mathcal{S}L & \xleftarrow{\cong} & \mathcal{S}L_0 \times \mathcal{S}L' \end{array}$$

Given these basic functors, the programming language types are defined as functors of type $\mathcal{C} \to \mathbf{Set}$ as follows:

- $\mathit{Var}(L) = L$, the set of available locations. The functor action on morphisms is $\mathit{Var}(f) : i \mapsto f(i)$.

- $Exp(L) = [\mathcal{S}L \to Int]$, the set of state-dependent expression valuations. The functor action on morphisms is $Exp(f) = [\mathcal{S}f \to \mathrm{id}_{Int}]$.
- $Comm(L) = \mathcal{T}L$, the set of state transformations. The functor action on morphisms is that of $\mathcal{T}$, $Comm(f) = \mathcal{T}f$.

Product types are interpreted "pointwise," if F and G are functors in $[\mathcal{C}, \mathbf{Set}]$ then $F \times G$ is the functor defined by $(F \times G)L = FL \times GL$ and $(F \times G)f = Ff \times Gf$. The terminal object in $[\mathcal{C}, \mathbf{Set}]$ is the constantly-singleton functor $\mathbf{1}$, given by $\mathbf{1}(L) = 1$, $\mathbf{1}(f) = \mathrm{id}_1$. This is the unit for the product constructor.

Every term is interpreted as a natural transformation. For example, the programming language term (along with its typing):

$$x : \mathbf{var} \vdash (x := x + 1) : \mathbf{comm}$$

is interpreted as a natural transformation of type $Var \dot{\to} Comm$.

$$[\![x : \mathbf{var} \vdash (x := x + 1) : \mathbf{comm}]\!]_L(i) = \lambda s.\, s[i \to s(i) + 1]$$

Given a world L, and a variable (storage location) i in that world, it gives a state transformation that maps a state s to a new version of the state where the value of the location i is updated to one plus its old value. This is a natural transformation. Given another world M with an injection $f : L \to M$, and a location $i' = f(i) \in M$, the state transformation obtained $\lambda s'.\, s'[i' \to s'(i') + 1]$ is precisely the expansion of the original command meaning along the morphism $Comm(f) = \mathcal{T}f$.

This presheaf category is not well-pointed. Neither does the programming language satisfy a context lemma. So, both are as they should be. To see the issue, consider what closed terms there are of type **comm**. A closed term cannot have any free identifiers. Without free identifiers, it cannot act on any external variables (i.e., storage locations). So, essentially, it can do *nothing*. All closed terms of type **comm** must be observationally equivalent to **skip**, the do-nothing command. Consider, on the other hand, closed terms of type **comm** $\to$ **comm**, or, equivalently, terms of the form c : **comm** $\vdash T[c]$: **comm**. We can enumerate a countably infinite number of terms for $T[c]$ all of which are observationally distinct: c, $c; c$, $c; c; c$, ... So, if we think of types as sets with elements, we are hopelessly lost. The type **comm** denotes a singleton set, but the type **comm** $\to$ **comm** has an infinite number of elements. How is that possible?

The functor category model neatly explains the issue. A type in the model is not a single set. Rather, it is a functor, mapping each world in $\mathcal{C}$ to a set. The **comm** type at world 0 (which denotes the empty set of locations), is indeed a singleton: $Comm(0) = \{\mathrm{id}\}$. However, $Comm(1)$, $Comm(2)$ etc. have more interesting elements. $Comm(1)$ has all state transformations for a single storage variable, $Comm(2)$ has all state transformations for two storage variables and so on. A "function," i.e., morphism, of type $Comm \dot{\to} Comm$, must give a natural family of maps for *each world*, i.e., a function of type $Comm(0) \to Comm(0)$, one of type $Comm(1) \to Comm(1)$, another of type $Comm(2) \to Comm(2)$ and so on. Just the fact that $Comm(0)$ is a singleton does not settle the issue. Hence, it is perfectly possible for $Comm(0)$ to be a singleton but $Comm \dot{\to} Comm$ to be infinite.

Exercise 38 (insightful) What are the global elements of the functors Var, $Comm$ and Exp?

Exercise 39 (insightful) Give defintions of natural transformations

$$\begin{aligned} &skip : \mathbf{1} \dot{\to} Comm \\ &seq : Comm \times Comm \dot{\to} Comm \\ &assign : Var \times Exp \dot{\to} Comm \end{aligned}$$

which can serve to interpret the evident commands **skip**, $C_1; C_2$ and $V := E$.

Yoneda embedding

If $\mathcal{C}$ is any category, there is an "embedding" of $\mathcal{C}$ in the functor category $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ as follows:

$$\begin{array}{rcl} A & \mapsto & \mathrm{hom}_{\mathcal{C}}(-, A) \\ (f : A \to B) & \mapsto & \mathrm{hom}_{\mathcal{C}}(-, f) : \mathrm{hom}_{\mathcal{C}}(-, A) \dot{\to} \mathrm{hom}_{\mathcal{C}}(-, B) \end{array}$$

We have just described a functor of type $\mathcal{C} \to [\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ called the "Yoneda embedding" of the category $\mathcal{C}$. The contravariant Yoneda lemma tells us that the second line in this mapping is a bijection. That means that the Yoneda embedding functor is full and faithful. The embedding gives us an exact image of $\mathcal{C}$ in $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$.

Recall that a presheaf category $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ is always cartesian closed. So, if $\mathcal{C}$ is not cartesian closed, $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ is a way of extending it to a cartesian closed category. If $\mathcal{C}$ is already cartesian closed or partially cartesian closed (i.e., has some products and exponents), all those products and exponents are preserved. But, whatever products and exponents are missing are created. [3]

One final remark before we close this section: Presheaf categories, or functor categories in general, provide excellent examples to test our categorical intuitions. Presheaf categories are not at all "set-like". Recall our first example of non-set-like categories, *viz.*, product categories $\mathcal{C}_1 \times \mathcal{C}_2$. We said that an object of $\mathcal{C}_1 \times \mathcal{C}_2$ is a pair of objects (A, B). So, it does not make sense to talk about its "elements." An object of $[\mathcal{C}^{\mathrm{op}}, \mathbf{Set}]$ is even more elaborate. It is a potentially infinite family of objects as well as morphisms between them. A morphism between two such "objects" is again a potentially infinite family of maps. So, we must leave set theory far behind to understand functor categories. Despite their lack of familiarity, all presheaf categories form models of typed lambda calculus. So, normal type constructions such as products and function spaces make sense for them.

Exercise 40 (insightful) Prove that the Yoneda embedding functor is full and faithful.

[3] This paragraph is packed with the revelation of some extra-ordinary facts. You will need to contemplate them more deeply at your leisure.

Chapter 5

Adjunctions

Adjunctions characterize structure and inheritance of structure.

Adjunctions, or adjoint functors, form one of the central concepts in category theory. They characterize almost all type constructors that one uses in type theory. They also characterize the inheritance hierarchies of mathematical structures as in Fig. 3.1. They also provide the foundation for monads and comonads.

5.1 Adjunctions in type theory

We first look at adjunctions for "homogeneous" functors, i.e., functors on a single underlying category. We use examples to motivate the definition.

Products

Consider the product construction in a category $\mathcal{C}$. A product $A \times B$ in $\mathcal{C}$ is an object equipped with the following morphisms:

1. Given a pair of morphisms $f : X \to A$ and $g : X \to B$, there must a morphism $\mathbf{pair}_{X,A,B} : X \to A \times B$, whose effect should be to produce a pair of results in $A \times B$. The morphism $\mathbf{pair}_{X,A,B}$ is informally written as $\langle f, g \rangle$. We can describe construction via a deduction rule:
$$\frac{f : X \to A \quad g : X \to B}{\langle f, g \rangle : X \to A \times B}$$
2. Given a morphism $h : X \to A \times B$, there must be morphisms $\mathrm{fst}_{X,A,B}(h) : X \to A$ and $\mathrm{snd}_{X,A,B}(h) : X \to B$, which project out the first and second components of the product. We write them also as deduction rules:
$$\frac{h : X \to A \times B}{\mathrm{fst}_{X,A,B}(h) : X \to A} \qquad \frac{h : X \to A \times B}{\mathrm{snd}_{X,A,B}(h) : X \to B}$$

Further, these constructions should satisfy the (β) and (η) equivalences as follows:

$$\begin{aligned} \mathrm{fst}\ \langle f, g \rangle &= f \\ \mathrm{snd}\ \langle f, g \rangle &= g \\ \langle \mathrm{fst}(h), \mathrm{snd}(h) \rangle &= h \end{aligned}$$

We can simplify the presentation using the notion of product category $\mathcal{C} \times \mathcal{C}$. Note that the pair of morphisms (f, g) is a simply a morphism in $\mathcal{C} \times \mathcal{C}$. We also abbreviate the pair

$(\mathrm{fst}_{X,A,B}(h), \mathrm{snd}_{X,A,B}(h))$ as $(\mathbf{proj}_{X,A,B}(h))$. Using these notations, our deduction rules become:

$$\frac{(f,g) : (X,X) \to (A,B)}{\mathbf{pair}_{X,A,B}(f,g) : X \to A \times B} \qquad \frac{h : X \to A \times B}{\mathbf{proj}_{X,A,B}(h) : (X,X) \to (A,B)}$$

and the equational laws now become:

$$\begin{array}{rcll} \mathbf{proj}_{X,A,B}(\mathbf{pair}_{X,A,B}(f,g)) & = & (f,g) & : \quad (X,X) \to (A,B) \\ \mathbf{pair}_{X,A,B}(\mathbf{proj}_{X,A,B}(h)) & = & h & : \quad X \to A \times B \end{array}$$

Note that the equational laws say precisely that **proj** and **pair** are *mutually inverse*.

The last step of abstraction is to note that (X,X) is obtained from the object X of $\mathcal{C}$ by the action of the diagonal functor $\Delta : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$. (Cf. Example 11). It is defined by $\Delta X = (X,X)$ and $\Delta f = (f,f)$.

So, the entire definition of products can be summarized by writing a single bidirectional deduction rule:

$$\frac{\Delta X \to (A,B) \qquad [\text{in } \mathcal{C} \times \mathcal{C}]}{X \to A \times B \qquad [\text{in } \mathcal{C}]}$$

The double line indicates that there are constructions in both the directions and they are mutually inverse. In addition, we also require that the constructions in the two directions should be natural transformations in all the type variables: X, A and B.

A more formal statement of this rule is to write:

$$\hom_{\mathcal{C}\times\mathcal{C}}(\Delta X,\, (A,B)) \quad \cong_{X,A,B} \quad \hom_{\mathcal{C}}(X,\, A \times B) \tag{5.1}$$

where the type subscripts on $\cong$ indicate that it is a *natural* isomorphism.

Coproducts

The coproduct construction in a category $\mathcal{C}$ involes an object $A + B$ in $\mathcal{C}$ equipped with the following morphisms:

1. Given a pair of morphisms $f : A \to X$ and $g : B \to X$, there must a morphism $\mathbf{case}_{A,B,X} : A + B \to X$, whose effect should be to do a case anlaysis on the argument in $A + B$. The combinator $\mathbf{case}_{X,A,B}(f,g)$ is also informally written as $[f,g]$. We denote the construction via the deduction rule:
$$\frac{f : A \to X \quad g : B \to X}{[f,g] : A + B \to X}$$

2. Given a morphism $h : A + B \to X$, there must be morphisms $\mathrm{inl}_{A,B,X}(h) : A \to X$ and $\mathrm{inr}_{A,B,X}(h) : B \to X$, which inject their argument into the left or right summand respectively. We write them as the deduction rules:
$$\frac{h : A + B \to X}{\mathrm{inl}_{A,B,X}(h) : A \to X} \qquad \frac{h : A + B \to X}{\mathrm{inr}_{A,B,X}(h) : A \to X}$$

Further, these constructions should satisfy the (β) and (η) equivalences as follows:

$$\begin{array}{rcl} \mathrm{inl}\,[f,g] & = & f \\ \mathrm{inr}\,[f,g] & = & g \\ [\mathrm{inl}(h), \mathrm{inr}(h)] & = & h \end{array}$$

In a manner similar to the case of products, we can simplify these criteria into a single bi-directional deduction rule:

$$\frac{(A, B) \to \Delta X \qquad [\text{in } \mathcal{C} \times \mathcal{C}]}{A + B \to X \qquad [\text{in } \mathcal{C}]}$$

This is expressed as a natural isomorphism of hom-sets:

$$\hom_{\mathcal{C}\times\mathcal{C}}((A, B),\, \Delta X) \quad \cong_{A,B,X} \quad \hom_{\mathcal{C}}(A + B,\, X) \tag{5.2}$$

Exponents

The exponent construction $A \Rightarrow B$ (or the "function space" or "internal hom") is equipped with the following morphisms:

1. Given a morphism $f : X \times A \to B$, there must be a morphism $\Lambda(f) : X \to (A \Rightarrow B)$, representing the "lambda abstraction" or "currying" of f:

$$\frac{f : X \times A \to B}{\Lambda(f) : X \to (A \Rightarrow B)}$$

2. This construction should have an inverse:

$$\frac{h : X \to (A \Rightarrow B)}{\mathbf{uncurry}(h) : X \times A \to B}$$

Once again, these two rules can be expressed as a bi-directional rule

$$\frac{X \times A \to B \quad [\text{in } \mathcal{C}]}{X \to (A \Rightarrow B) \quad [\text{in } \mathcal{C}]}$$

and as a natural isomorphism:

$$\hom_{\mathcal{C}}(X \times A,\, B) \quad \cong_{X,A,B} \quad \hom_{\mathcal{C}}(X,\, A \Rightarrow B) \tag{5.3}$$

The general picture

Let us put all the three examples side by side and compare them:

$$\frac{\Delta X \to (A, B) \quad [\text{in } \mathcal{C} \times \mathcal{C}]}{X \to A \times B \quad [\text{in } \mathcal{C}]} \qquad \frac{A + B \to X \quad [\text{in } \mathcal{C}]}{(A, B) \to \Delta X \quad [\text{in } \mathcal{C} \times \mathcal{C}]} \qquad \frac{X \times A \to B \quad [\text{in } \mathcal{C}]}{X \to (A \Rightarrow B) \quad [\text{in } \mathcal{C}]}$$

We have inverted the top and bottom lines in the coproduct case to bring out the symmetry.

It is clear that, in general, there are two categories involved, one above the line and one below the line. We will use the letters $\mathcal{C}$ and $\mathcal{X}$ to stand for them. There are also two functors involved, one above the line which occurs to the left of "$\to$" and the other below the line which occurs to the right of "$\to$". So, in general, we are looking at bidirectional rules like this:

$$\frac{Fx \to c \quad [\text{in } \mathcal{C}]}{x \to Gc \quad [\text{in } \mathcal{X}]} \tag{5.4}$$

(We use lower case letters for objects in order to avoid confusion with the letters in the examples.) The functor F is of type $\mathcal{X} \to \mathcal{C}$ so that Fx is an object of $\mathcal{C}$. The functor G is of

type $\mathcal{C} \to \mathcal{X}$ so that Gc is an object of $\mathcal{X}$. We can also write the meaning of the bi-directional rule explicitly as a natural isomorphism between hom-sets:

$$\phi_{x,c} \quad : \quad \hom_{\mathcal{C}}(Fx,\, c) \quad \cong_{x,c} \quad \hom_{\mathcal{X}}(x,\, Gc) \tag{5.5}$$

We use the symbolic notation $F \dashv G : \mathcal{C} \to \mathcal{X}$ or simply $F \dashv G$ to denote the situation. We say that:

1. F is a *left adjoint* of G,
2. G is a *right adjoint* of F, and
3. (F, G) form an *adjoint pair* of functors.

It can be proved that F and G determine each other uniquely upto natural isomorphism. That is, given a functor $F : \mathcal{X} \to \mathcal{C}$, if F has a right adjoint $\mathcal{C} \to \mathcal{X}$ then it will be unique upto isomorphism. (If $G, G' : \mathcal{C} \to \mathcal{X}$ are both right adjoint to F, then $Gc \cong_c G'c$.) Similar statement holds for G determining F.

The letters F and G are almost always used in the literature to stand for the left adjoints and right adjoints respectively. This helps us memorize them. Note that F is called the "left" adjoint because it occurs in the left argument of a hom-functor and G is the "right" adjoint because it occurs in the right argument of a hom-functor. The order in which the two hom-sets are written is irrelevant. The isomorphism above means the same as:

$$\hom_{\mathcal{X}}(x,\, Gc) \quad \cong_{x,c} \quad \hom_{\mathcal{C}}(Fx,\, c)$$

where we find G occurring to the left of $\cong$ symbol and F to the right. That is immaterial. F is still the "left" adoint and G is still the "right" adjoint.

We show in the table below, what the various parameters are in our examples:

	$\mathcal{C}$	$\mathcal{X}$	F	G
products	$\mathcal{C} \times \mathcal{C}$	$\mathcal{C}$	$\Delta : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$	$\times : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$
coproducts	$\mathcal{C}$	$\mathcal{C} \times \mathcal{C}$	$+ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$	$\Delta : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$
exponents	$\mathcal{C}$	$\mathcal{C}$	$(- \times A) : \mathcal{C} \to \mathcal{C}$	$(A \Rightarrow -) : \mathcal{C} \to \mathcal{C}$

You will see in the literature statements such as "products are right adjoint to the diagonal," "coproducts are left adjoint to the diagonal," and "exponents are right adjoing to the product."

Exercise 41 (basic) Pattern match each of the example adjunctions given in this section against the general form (5.5) and write down what the various variables are instantiated to. These include the categories $\mathcal{X}$ and $\mathcal{C}$, the functors F and G, the object variables x and c, and the natural isomorphism $\phi_{x,c}$.

Warning Most beginning students attempt to generalize some intuition from the examples to form an idea of what adjunctions mean in general. It is tempting to think of F and G as inverses of each other. That is no good, however. If F and G were inverses, that would make the categories $\mathcal{X}$ and $\mathcal{C}$ isomorphic. But our examples show clearly that they are not isomorphic categories. However, F and G are "opposites" in a more abstract sense, as will become clear in the ensuing discussion. The bi-directional deduction rule is perhaps the best means of remembering the general picture and pattern matching against examples:

5.2 Adjunctions in algebra

Recall the "inheritance of structure" examples from Fig. 3.1. Each step of inheritance there is captured by an adjunction. Such algebraic examples provide sharper intuition for the adjunction concept than the type theory examples. Hence, we study them closely.

A semigroup is a pair $A = (A, \cdot)$ where $\cdot$ is an associative binary operation on A (called "multiplication"). We have indicated that there is a forgetful functor $G : \mathbf{SGrp} \to \mathbf{Set}$, which maps $(A, \cdot)$ to its underlying set A and every semigroup homomorphism to its underlying function. The forgetful functor has a left adjoint $F : \mathbf{Set} \to \mathbf{SGrp}$.

If X is a set, there is a semigroup *generated* by X. It is the minimal semigroup that includes X and is closed under a formal multiplication operation. The free semigroup FX must therefore include all the elements of X, and all formal products of the form $x_1 x_2 \cdots x_k$ for elements $x_1, \ldots, x_k \in X$. Such formal products are essentially nonempty sequences of elements of X, and we can treat them as such. A function $f : X \to Y$ in **Set** can be extended to a semigroup homomorphism $Ff : FX \to FY$ by setting $Ff(x_1 \cdots x_k) = f(x_1) \cdots f(x_k)$. Thus F is a functor $\mathbf{Set} \to \mathbf{SGrp}$.

Now, we argue that there is a natural isomorphism:

$$\frac{FX \to A \quad [\text{in } \mathbf{SGrp}]}{X \to GA \quad [\text{in } \mathbf{Set}]}$$

Given a semigroup homomorphism $h : FX \to A$, we can consider its action on formal products in FX that contain a single element of X (or, put another way, the action on singleton sequences). This gives us a function $\phi(h)$ from X to the underlying set of A (which is denoted GA). Moreover, the entire homomorphism h can be uniquely recovered from the function $\phi(h)$ because $h(x_1 \cdots x_k) = h(x_1) \cdots h(x_k)$. Thus ϕ is an isomorphism $\mathbf{SGrp}(FX, A) \cong \mathbf{Set}(X, GA)$. It remains to show that it is natural. Consider naturality in the X parameter.

$$\begin{array}{ccc} X & \mathbf{SGrp}(FX, A) & \xrightarrow{\phi_{X,A}} \mathbf{Set}(X, GA) \\ \uparrow f & \downarrow \mathbf{SGrp}(Ff, A) & \downarrow \mathbf{Set}(f, GA) \\ Y & \mathbf{SGrp}(FY, A) & \xrightarrow{\phi_{Y,A}} \mathbf{Set}(Y, GA) \end{array}$$

Given a homomorphism $h : FX \to A$ in the top left hand corner, the function $\phi_{X,A}(h)$ is just its restriction to singletons. The function $\mathbf{Set}(f, GA)$ sends it to $\phi_{X,A}(h) \circ f$, which is just the mapping $y \mapsto h(f(y))$. On the other hand, the action of $\mathbf{SGrp}(Ff, A)$ on h is to send it to $h \circ Ff$. Since Ff is defined by $Ff(y_1 \cdots y_k) = f(y_1) \cdots f(y_k)$, restricting $h \circ Ff$ to singletons in FY gives presicely the same mapping $y \mapsto h(f(y))$. Naturality in the A parameter is left to the reader.

Consider the next step in the inheritance hierarchy of Fig. 3.1, $G : \mathbf{Mon} \to \mathbf{SGrp}$. A monoid is a semigroup with a unit element. The forgetful functor G forgets the fact that a monoid has a unit element and treats it simply as a semigroup. What is the left adjoint to this functor, if any?

$$\frac{FX \to A \quad [\text{in } \mathbf{Mon}]}{X \to GA \quad [\text{in } \mathbf{SGrp}]}$$

If X is a semigroup, we need to construct the "free monoid generated" by X such that monoid homomorphisms $FX \to A$ can be uniquely constructed from the underlying semigroup morphisms. Consider two cases:

1. The semigroup X already has a unit element 1_X. Note that the semigroup GA has 1_A as its unit element. (GA is just the "underlying" semigroup of monoid A, and the latter has the unit element 1_A, which still remains an element in GA.) In this case, a semigroup homomorphism $f : X \to GA$ is already a monoid homomorphism, and we we should take $FX = X$ and map f to itself in $FX \to A$.

2. The semigroup X does not have a unit element. In this case, we can take $FX = X \uplus \{1\}$ adjoining a new unit element. Any semigroup homomorphism $f : X \to GA$ can be canonically extended to a monoid homomorphism $f^* : FX \to A$ whose function graph is $f^* = f \uplus \{(1, 1_A)\}$.

So, "the free monoid generated" by X is the same as X if X has a unit element, and adjoins a new unit element to it otherwise. In semigroup theory, such a monoid is denoted X^1.

We notice from Fig. 3.1 that we have one adjuction pair $F_1 \dashv G_1 : \mathbf{SGrp} \to \mathbf{Set}$ and another adjunction pair $F_2 \dashv G_2 : \mathbf{Mon} \to \mathbf{SGrp}$. It is a natural question to ask whether such sequences of adjunctions compose. Indeed that is the case.

Theorem 22 If $F_1 \dashv G_1 : \mathcal{C} \to \mathcal{X}$ and $F_2 \dashv G_2 : \mathcal{D} \to \mathcal{C}$ are adjunctions, then their composition $F_2F_1 \dashv G_1G_2 : \mathcal{D} \to \mathcal{X}$ is an adjunction pair.

Exercise 42 (insightful) Produce concrete definitions for the adjunction pair of functors between **Set** and **Mon**. Show that they form an adjunction.

Exercise 43 (challenging) Consider at least two other adjascent structures in the hierarchy given in Fig. 3.1 and find the adjunction pair of functions governing their relationship.

Once we understand the algebraic examples of adjunctions, it is possible to glean some intuition from them. The functor G, the right adjoint, forgets structure. For example, it maps semigroups $(A, \cdot)$ to their underlying sets A. It maps monoids $(A, \cdot, 1_A)$ to their underlying semigroups $(A, \cdot)$, and so on. The functor F, working in the opposite direction, adds the required structure in a minimal way. The left adjoint $\mathbf{Set} \to \mathbf{SGrp}$, maps a set X to the free semigroup generated by X. It is in a certain sense, the least that one needs to do to convert a set into a semigroup. Similarly, the left adjoint $\mathbf{SGrp} \to \mathbf{Mon}$ does the least amount of work needed to convert a semigroup into a monoid. Thus, the right adjoints "subtract" structure and left adjoints "add" structure in such a way that they work in a mutually opposite way.

This intuition is not cast in stone, however. It is hard to see such adding and subtracting going in the type-theoretic examples of adjunctions. Moreover, if the adjunctions in a category $\mathcal{C}$ add and subtract structure in this fashio, what about the dual category $\mathcal{C}^{\mathrm{op}}$. There, the left adjoints would be doing the subtracting and the right adjoints would be doing the adding. In categories like **Rel**, which are their own duals, each functor in the adjunction pair may be doing adding and subtracting. Unfortunately, intuitions can only go so far.

5.3 Units and counits of adjunctions

We reproduce below the natural isomorphism for adjunctions, given previously in (5.5):

$$\phi_{x,c} \quad : \quad \hom_{\mathcal{C}}(Fx,\, c) \quad \cong_{x,c} \quad \hom_{\mathcal{X}}(x,\, Gc) \quad : \quad \phi^{-1}_{x,c}$$

Since ϕ and ϕ^{-1} are natural transformations between hom-functors, we might wonder if the Yoneda lemma-like reasoning has something useful to say about them. Indeed it does.

By setting $c = Fx$ in the isomorphism, we get $\hom_{\mathcal{C}}(Fx, Fx)$ on the left hand side, which contains the identity morphism id_{Fx}. The left-to-right isomorphism maps the identity to a morphism $x \to GFx$ in $\mathcal{X}$. It turns out that this is a natural transformation:

$$\eta : \mathrm{Id} \to GF \qquad \eta_x : x \to GFx \qquad \eta_x = \phi_{x,Fx}(\mathrm{id}_x)$$

This is called the *unit* of the adjunction. Moreover, by the contravariant

Dually, by setting $x = Gc$ in the isomorphism, we get $\hom_{\mathcal{X}}(Gc, Gc)$ on the right hand side, which contains the identity morphism id_{Gc}. The right-to-left isomorphism maps it to another natural transformation, that is called the *counit* of the adjunction:[1]

$$\epsilon : FG \to \mathrm{Id} \qquad \epsilon_c : FGc \to c \qquad \epsilon_c = \phi^{-1}_{Gc,c}(\mathrm{id}_{Gc})$$

We show examples of units and counits below. But it is worthwhile to continue at the abstract level because the units and the counits have an interesting story to tell.

The natural transformation $\phi_{x,c}$ as well as its inverse must work *generically* in x and c, without depending on what these types are. So, what can $\phi_{x,c}$ do? Given a morphism $f : Fx \to c$, it needs to produce a morphism $\phi_{x,c}(f) : x \to Gc$. We have already identified that it has a way to go from x to GFx by the morphism η_x (which is natural in x). It can post-compse Gf to it as follows:

$$x \xrightarrow{\eta_x} GFx \xrightarrow{Gf} Gc \tag{5.6}$$

In fact, we can prove that $\phi_{x,c}(f)$ must be precisely this, using an argument similar to that of Yoneda Lemma.

$$\begin{array}{ccc} Fx & \hom(Fx, Fx) & \xrightarrow{\phi_{x,Fx}} \hom(x, GFx) \\ \Big\downarrow f & \hom(Fx, f)\Big\downarrow & \qquad\qquad \Big\downarrow \hom(x, Gf) \\ c & \hom(Fx, c) & \xrightarrow{\phi_{x,c}} \hom(x, Gc) \end{array}$$

Considering $\mathrm{id}_{Fx} : Fx \to Fx$ in the top left hand corner, chasing it right and down yields $\eta_x; Gf$. Chasing it down and right yields $\phi_{x,c}(f)$. The two must be equal by naturality of ϕ in the second type parameter.

Similarly, naturality in the first type parameter for ϕ^{-1} shows that $\phi^{-1}_{x,c}(g : x \to Gc)$ is equal to the composite:

$$Fx \xrightarrow{Fg} FGc \xrightarrow{\epsilon_c} c \tag{5.7}$$

We have now expressed both $\phi_{x,c}$ and $\phi^{-1}_{x,c}$ in terms of the unit and the counit. We can restate the fact that they are mutual inverses:

$$\begin{array}{lrcl} (\forall f : Fx \to c) & Fx \xrightarrow{F\eta_x} FGFx \xrightarrow{FGf} FGc \xrightarrow{\epsilon_c} c & = & f \\ (\forall g : x \to Gc) & x \xrightarrow{\eta_x} GFx \xrightarrow{GFg} GFGc \xrightarrow{G\epsilon_c} Gc & = & g \end{array}$$

It turns out that we can capture the requisite conditions just by instantiating these for $f = \mathrm{id}_{Fx} : Fx \to Fx$ and $g = \mathrm{id}_{Gc} : Gc \to Gc$.

Theorem 23 A pair of functors form an adjunction $F \dashv G : \mathcal{C} \to \mathcal{X}$ if and only if there are natural transformations $\eta : \mathrm{Id} \dot{\to} GF$ and $\epsilon : FG \dot{\to} \mathrm{Id}$ such that the following composites are identities:

$$G \xrightarrow{\eta G} GFG \xrightarrow{G\epsilon} G \qquad F \xrightarrow{F\eta} FGF \xrightarrow{\epsilon F} F$$

[1]The terminology of "unit" and "counit" for these natural transformations may seem unmotivated. You will need to wait until the next chapter — on Monads — to see the motivation.

From the unit and the counit, the natural bijection $\phi_{x,c}$ can be computed using the formulas (5.6) and (5.7).

This characterisation can be alternatively taken as the *definition* of adjunctions. It is considered preferable to our original definition because it does not depend on hom-sets, which are not always available in various generalisations of the basic category theory (such enriched categories).

Exercise 44 (medium) Show that $\phi^{-1}_{x,c}(g)$ is equal to the composite morphism in (5.7).

Exercise 45 (insightful) We have proved the "only if" part of Theorem 23 in the discussion leading up to it. Prove the converse. That is, given that the composites displayed in the Theorem are identities, show that they determine a natural isomorphism $\phi_{x,c}$.

Example 24 (Products) Consider the adjunction isomorphism for products in a category given by (5.1). We reproduce it below along with the natural transformations witnessing the isomorphism:

$$\mathbf{pair}_{X,(A,B)} \quad : \quad \hom_{\mathcal{C}\times\mathcal{C}}(\Delta X,\, (A,B)) \quad \cong_{X,A,B} \quad \hom_{\mathcal{C}}(X,\, A\times B) \quad : \quad \mathbf{proj}_{X,(A,B)}$$

Recall that the left adjoint F is $\Delta : \mathcal{C} \to \mathcal{C}\times\mathcal{C}$ and the right adjoint G is $\times : \mathcal{C}\times\mathcal{C} \to \mathcal{C}$. The composite GF is then $\times\Delta : \mathcal{C} \to \mathcal{C}$ which maps $X \mapsto X \times X$, and the composite FG is $\Delta\times : \mathcal{C}\times\mathcal{C} \to \mathcal{C}\times\mathcal{C}$ which maps $(A,B) \mapsto (A\times B,\, A\times B)$.

The counit of the adjunction is $\epsilon_{A,B} = \mathbf{proj}_{A\times B,(A,B)}(\mathrm{id}_{A\times B})$. Recalling that $\mathbf{proj}(h)$ means $(\mathrm{fst}(h),\, \mathrm{snd}(h))$, we can say that $\epsilon_{A,B} = (\mathrm{fst}(\mathrm{id}_{A\times B}), \mathrm{snd}(\mathrm{id}_{A\times B}))$. These two are nothing but projection morphisms, normally written as $\pi_1 : A\times B \to A$ and $\pi_2 : A\times B \to B$. For example, in **Set**, $\pi_1 : (a,b) \mapsto a$ and $\pi_2 : (a,b) \mapsto b$.

The unit of the adjunction is $\eta_X : X \to X\times X$ given by $\eta_X = \mathbf{pair}_{X,A\times B}(\mathrm{id}_X, \mathrm{id}_X)$. This is often called the "diagonal" morphism and denoted $\delta_X : X \to X\times X$. For example, in **Set**, this morphism is $\delta : x \mapsto (x,x)$.

Exercise 46 (medium) Find the unit and the counit of the natural isomorphisms for coproducts (5.2) and exponents (5.3).

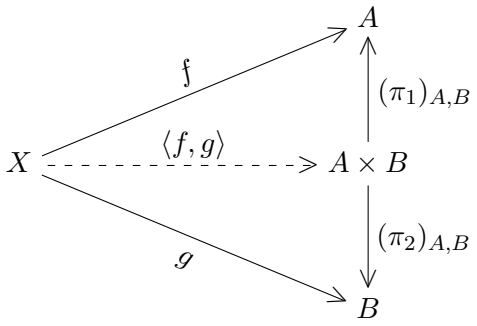
Exercise 47 (medium) Find the unit and the counit of the natural isomorphisms for $F \dashv G : \mathbf{SGrp} \to \mathbf{Set}$ and $F' \dashv G' : \mathbf{Mon} \to \mathbf{SGrp}$. Alternatively, you can do so for any of the adjunctions in the inheritance hierarchy in Fig. 3.1.

5.4 Universals

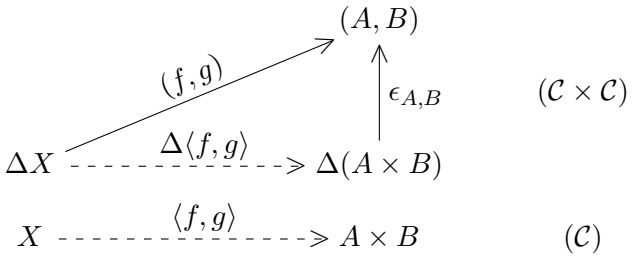
The standard presentation of products in a category looks quite different from what we have seen above in terms of adjunctions. However, the standard presentation and the adjunction presentation are equivalent. Understanding this correspondence leads us to correlate the notion of "universal arrows" and "adjunctions."

A product of two objects A and B in a category $\mathcal{C}$ is usually said to be an object $A\times B$ along with two morphisms $(pi_1)_{A,B} : A\times B \to A$ and $(\pi_2)_{A,B} : A\times B \to B$ such that, for every pair of morphisms $(f : X \to A,\, g : X \to B)$ in $\mathcal{C}$ there is a unique arrow $X \to A\times B$ such that

the following diagram commutes:



Let us simplify this presentation using product categories. The pair of morphisms (f, g) can be expressed as an arrow in the product category from $(X, X) \to (A, B)$. Likewise, the pair of projections (π_1, π_2) can be expressed as an arrow in the product category from $(A \times B,\ A \times B) \to (A, B)$. Denote this by $\epsilon_{A,B}$. We see that duplicated objects like (X, X) and $(A \times B,\ A \times B)$ can be represented by the diagonal functor $\Delta : \mathcal{C} \to \mathcal{C} \times \mathcal{C}$. Then we have the following revised picture:



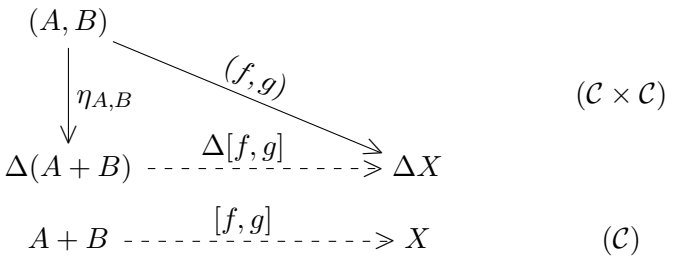
The first part of the picture, the triangle, is in $\mathcal{C} \times \mathcal{C}$. The second part of the picture, consisting of a single arrow, is in $\mathcal{C}$. It is easy to see that the bottom line of the triangle is nothing but the diagonal functor Δ applied to the second picture. So, in words, the diagram is saying:

> The product of A and B in $\mathcal{C}$ is an object $A \times B$ in $\mathcal{C}$, along with an arrow
>
> $$\epsilon_{A,B} : \Delta(A \times B) \to (A, B) \qquad \text{in } \mathcal{C} \times \mathcal{C}$$
>
> such that every arrow $(f, g) : \Delta X \to (A, B)$ uniquely factors through an arrow $\Delta(\langle f, g\rangle) : \Delta X \to \Delta(A \times B)$.

That is quite a complicated statement! But you can at least see that its main purpose is to state that the arrows $\Delta X \to (A, B)$ in $\mathcal{C} \times \mathcal{C}$ and the arrows $X \to A \times B$ are in one-to-one correspondence. And, we know that that is what adjunctions are about.

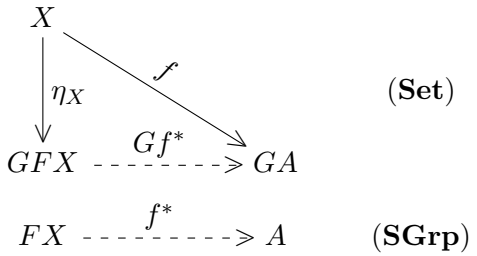
We look at a couple of more examples so that you can see the pattern. A corresponding definition of *coproducts* is as follows:

> The coproduct of A and B in $\mathcal{C}$ is an object $A + B$ in $\mathcal{C}$, along with an arrow
>
> $$\eta_{A,B} : (A, B) \to \Delta(A + B) \qquad \text{in } \mathcal{C} \times \mathcal{C}$$
>
> such that every arrow $(f, g) : (A, B) \to \Delta X$ uniquely factors through an arrow $\Delta([f, g]) : \Delta(A + B) \to \Delta(X)$.



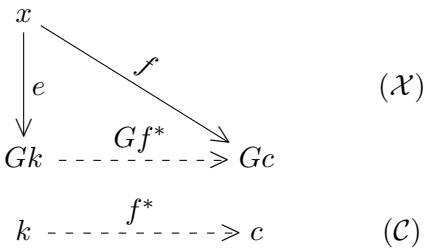
We will recast the adjunction $F \dashv G : \mathbf{SGrp} \to \mathbf{Set}$ in the same mold:

> The free semigroup generated by a set X is an object FX in **SGrp**, along with an arrow
>
> $$\eta_X : X \to GFX \qquad \text{in } \mathbf{Set}$$
>
> such that every arrow $f : X \to GA$ uniquely factors through an arrow $Gf^* : GFX \to GA$.



The general definition of universal arrows is as follows:

Definition 25 (Universal arrow) Given a functor $G : \mathcal{C} \to \mathcal{X}$ and an object x of $\mathcal{X}$, a *universal arrow* from x to G is an arrow $e : x \to Gk$ in $\mathcal{X}$ such that every arrow $f : x \to Gc$ in $\mathcal{X}$ uniquely factors through e, i.e., there is a unique arrow $f^* : k \to c$ in $\mathcal{C}$ such that $f = e; Gf^*$.



(We are using tricky language here. When we ask for "an arrow $e : x \to Gk$," we are asking for a pair $\langle k, e : x \to Gk \rangle$. When we say "every arrow $f : x \to Gc$," we are similarly quantifying over pairs $\langle c, f : x \to Gc \rangle$.)

This concept is harder to state than adjunctions because of the very many ideas involved. But it is easy to see that it is essentially stating that there is a bijection between the arrows $x \to Gc$ in $\mathcal{X}$ and the arrows $k \to c$ in $\mathcal{C}$, i.e.,

$$\phi_c \quad : \quad \hom_{\mathcal{C}}(k, c) \quad \cong \quad \hom_{\mathcal{X}}(x, Gc)$$

The isomorphism statement does not tell us how to construct such a bijection, whereas the statement of the definition tells us that there should be an arrow $e : x \to Gc$ and that every arrow $f : x \to Gc$ should uniquely factor through it.

The concept of an adjunction is now a straightforward generalization of the concept of universal arrow, where instead of an arrow from a particular x in $\mathcal{X}$, we ask for an arrow from *every* x in $\mathcal{X}$. The objects k are then given functorially in x, and ϕ becomes natural in x.

Exercise 48 (medium) Show that the examples of coproducts and free semigroups are instances of the universal arrow concept.

Exercise 49 (insightful) Formulate the dual concept, "couniversal arrow," which goes from a functor $F : \mathcal{X} \to \mathcal{C}$ to an object a of $\mathcal{C}$. Verify that the notion of products is an instance of the idea.

Chapter 6

Monads

Monads create structure.

If adjunctions characterize structure, monads may be said to "create" it. More precisely, monads give rise to algebraic structures so that one gets them for "free." This point is somewhat contentious. Some researchers hold that the algebraic structures created for free may not be entirely suitable for one's purposes. So, one should not merely depend on such free structures. Nevertheless, the idea that one gets structure for free is worth a careful examination.

There are two equivalent formulations of monads, one called "Kleisli triples" and the other called "monads." The latter is more modern, and generalizes to 2-categories easily. However, Kleisli triples are easier to provide a concrete intuition. So, we start with Kleisli triples.

6.1 Kleisli triples and Kleisli categories

Recall the functor $\mathbf{List} : \mathbf{Set} \to \mathbf{Set}$ from Example 10, indeed the first example of a functor we have seen. It has an interesting piece of mathematical structure associated with it, which serves to motivate the idea of monads.

1. There is a natural transformation $\eta : \mathrm{Id} \to \mathbf{List}$ which serves to inject a set X into $\mathbf{List}\, X$. It is given by $\eta_X(x) = [x]$, where $[x]$ is the singleton list containing x. The natural transformation η is called the *unit* of the **List** monad.

2. Given a function $f : X \to \mathbf{List}\, Y$, we can extend it to a function $f^* : \mathbf{List}\, X \to \mathbf{List}\, Y$ by defining $f^*([x_1, x_2, \ldots, x_k]) = f(x_1) \cdot f(x_2) \cdots f(x_n)$, where the binary operation "$\cdot$" is the list append or concatenation operation. The mapping $(\ )^*$ of functions $X \to \mathbf{List}\, Y$ to $\mathbf{List}\, X \to \mathbf{List}\, Y$ is called the *Kleisli extension* operation of the **List** monad.

These two constructions lead us to construct a new category called the *Kleisli category of the List monad*, denoted $\mathbf{Set}_{\mathbf{List}}$. Its objects are still sets but morphisms $f : X \xrightarrow{K} Y$ are functions $f : X \to \mathbf{List}\, Y$ in **Set**. (Note that we are using a decorated arrow symbol $\xrightarrow{K}$ to represent the arrows in the Kleisli category). The identity arrows $\mathrm{id}_X : X \xrightarrow{K} X$ are nothing but the unit natural transformations η_X. To compose two arrows $f : X \xrightarrow{K} Y$ and $g : Y \xrightarrow{K} Z$, we form the following composite in **Set**:

$$X \xrightarrow{f} \mathbf{List}\, Y \xrightarrow{g^*} \mathbf{List}\, Z$$

where we have used the Kleisli extension of g in the second position. For this to be a category, we need the axioms of categories to be satisfied, which say:

$$\begin{array}{ll} f \circ \mathrm{id}_X = f & f^* \circ \eta_X = f \\ \mathrm{id}_Y \circ f = f & (\eta_Y)^* \circ f = f \\ h \circ (g \circ f) = (h \circ g) \circ f & h^* \circ (g^* \circ f) = (h^* \circ g)^* \circ f \end{array}$$

The equations on the left are in the Kleisli category and those on the right are the corresponding formulations in **Set**. In fact, these equations can be simplified somewhat, leading to a more succint definition.

Definition 26 (Kleisli triple) A Kleisli triple $(T, \eta, (-)^*)$ consists of a functor $T : \mathcal{C} \to \mathcal{C}$ from a category to itself, a natural transformation $\eta : \mathrm{Id} \to T$ and a mapping of arrows $f : X \to TY$ to $f^* : TX \to TY$ satisfying three laws:

$$\begin{array}{l} f^* \circ \eta_X = f \\ (\eta_Y)^* = \mathrm{id}_{TY} \\ (h^* \circ g)^* = h^* \circ g^* \end{array}$$

(The notion of Kleisli triple is equivalent to that of monads, which will be seen in the next section. So, it is common to refer to Kleisli triples themselves as monads.)

Theorem 27 (Kleisli category) If $T = (T, \eta, (-)^*)$ is a Kleisli triple on a category $\mathcal{C}$, then there is a category $\mathcal{C}_T$ whose objects are the same as those of $\mathcal{C}$ and arrows $f : X \overset{K}{\to} Y$ are the arrows $f : X \to TY$ of $\mathcal{C}$. The identity arrrows $X \overset{K}{\to} X$ are the unit natural transformations $\eta_X : X \to TX$ and composition of arrows is given by the construction:

$$X \xrightarrow{f} TY \xrightarrow{g^*} TZ$$

Dually, a Kleisli co-triple $(L, \epsilon, (-)^*)$ consists of an endofuctor $L : \mathcal{C} \to \mathcal{C}$, a natural transformation $\epsilon : L \to \mathrm{Id}$, and a mapping of arrows $f : LX \to Y$ to $f^* : LX \to LY$ satisfying three laws that are dual to the above. The Kleisli category of a co-triple $\mathcal{C}_L$ (sometimes called the "co-Kleisli" category) has the same objects as $\mathcal{C}$ but arrows $f : X \overset{K}{\to} Y$ obtained from the arrows $f : LX \to Y$ of $\mathcal{C}$.

Exercise 50 (Powerset) Consider the powerset functor $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$. We can give it a Kleisli triple structure as follows:

- The unit $\eta_X : X \to \mathcal{P}X$ maps x to the singleton set $\{x\}$.
- The Kleisli extension $f^* : \mathcal{P}X \to \mathcal{P}X$ maps a subset $u \subseteq X$ to $\bigcup_{x \in u} f(x)$.

It is easy to verify that the axioms of Kleisli triples are satisfied.

The Kleisli category $\mathbf{Set}_{\mathcal{P}}$ has sets as its objects and functions $X \to \mathcal{P}X$ as its morphisms. The identity morphism is the function $x \mapsto \{x\}$. The composition of $f : X \to \mathcal{P}Y$ and $g : Y \to \mathcal{P}Z$ is the function $x \mapsto \bigcup_{y \in f(x)} g(y)$.

An equivalent formulation of the Kleisli category is the category **Rel** of sets and relations. The identity arrows $\mathrm{id}_X : X \to X$ are just the identity relations. The composition of relations $r : X \to Y$ and $s : Y \to Z$ is the normal relational composition $r \cdot s : X \to Z$. ■

Exercise 51 (basic) Prove Theorem 27 by verifying that the axioms of categories are satisfied.

Exercise 52 (basic) Verify that the unit and the Kleisli extension operation of the **List** monad satisfy the laws of Kleisli triples.

Exercise 53 (medium) Find a Kleisli triple structure for the powerset function $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$.

Exercise 54 (medium) Complete the definitions of Kleisli co-triples and the Kleisli categories of co-triples.

Exercise 55 (medium) Consider the category $\mathbf{CPO}_\perp$, whose objects are "pointed" dcpo's (i.e., dcpo's A with least elements, denoted $\perp_A$) and whose morphisms are *strict* continuous functions (continuous funciton preserving the least elements: $f(\perp_A) = \perp_B$.) The *lifting* functor $(\ )_\perp : \mathbf{CPO}_\perp \to \mathbf{CPO}_\perp$ sends a pointed cpo A to another one, $A_\perp$, which has an additional $\perp$ element adjoined to A and partial order extended by setting $\perp \sqsubseteq x$ for all $x \in A$. On morphisms, the action $f_\perp : A_\perp \to B_\perp$ is the obvious one. Find a Kleisli co-triple structure for the lifting functor and examine its Kleisli category.

The point of monads (and Kleisli triples) is that they automatically define a new category from a base category. Recall, from Section 1.4, that a programming language (or, in fact, any formal language with a term structure that provides for variables substitution) can be viewed as a category. The opposite is true as well. If we have a category, we can devise a language notation for it where the terms denote the arrows of the category. In fact, we can devise two language notations, corresponding to *call-by-value* programming languages and *call-by-name* programming languages.

The call-by-value notation is as follows:

$$\frac{}{x : A \vdash x : A} \qquad \frac{}{\eta_A : A \to TA}$$

$$\frac{x : A \vdash M_x : B \quad y : B \vdash N_y : C}{x : A \vdash N_y[y := M_x] : C} \qquad \frac{f : A \to TB \quad g : B \to TC}{g^* \circ f : A \to TC}$$

Here the monad T is implicit in the programming language. It is not expressed in the type system. So, a term of type A is in reality a "computation" yielding a result of type A.

In the call-by-name notation, the monad is explicit in the type system of the programming language:

$$\frac{}{\eta_A : A \to TA} \qquad \frac{}{x : A \vdash [x] : TA}$$

$$\frac{f : A \to TB \quad g : B \to TC}{g^* \circ f : A \to TC} \qquad \frac{x : A \vdash M_x : TB \quad y : B \vdash N_y : TC}{x : A \vdash [N_y \mid y \leftarrow M_x] : TC}$$

Returning to our example of the **List** monad, we can devise a language where a term $x : A \vdash M_x : B$ means a function of type $A \to \mathbf{List}\, B$.

- A variable term x in this language would mean the singleton list $[x]$.
- If we substitute for a variable y in N_y by a term M_x, it means that we let y range over the elements of the list M_x, take the list N_y for each of those elements, and consider the concatenation of all those lists as a big list.

We may devise a computational mechanism for the language, whereby terms in the language are evaluated to produce the elements of the denoted list *one by one*. For instance, the term $x * x$ can be evaluated by specifying that x should take its values from the list $[2, 3]$. The evaluation then produces 4 the first time and 9 the second time. The notaion of "list comprehensions" available in functional programming languages works this way.

Moggi noticed that call-by-value programming languages have semantic structure that corresponds to Kleisli categories of triples. For example, the category of sets and relations **Rel** can be thought of as the Kleisli category of the powerset monad $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$. As we remarked in Section 3.1, **Rel** is isomorphic to $\mathbf{Set}_{\mathcal{P}}$, which is nothing but the Kleisli category of the powerset functor. A morphism in $\mathbf{Set}_{\mathcal{P}}$ is a function of type $X \to \mathcal{P}Y$ (or a "nondeterministic function"). An evaluators for terms in this language produces some arbitrary element of the denoted set.

For another example, the category of sets and partial functions **Pfn** can be thought of as the Kleisli category of lifting. The *lifting* of a set X is a set $X_\perp = X \uplus \perp$ that has an adjoined element $\perp$ representing an undefined value. A partial function $f : X \rightharpoonup Y$ can be equivalently viewed as a total function $f : X \to Y_\perp$.

Exercise 56 (medium) Characterize the powerset functor and the lifting functor as Kleisli triples, i.e., find the natural transformation η and the Kleisli extension $(-)^*$ and verify that the equational laws hold.

6.2 Monads and algebras

The modern definition of monads uses a structure similar to that of monoids. It only uses the concepts of functor and natural transformation (no functions on hom-sets), so that it can generalize to arbitrary 2-categories.

Definition 28 (monad) A *monad* on a category $\mathcal{C}$ is a triple $\langle T, \eta, \mu \rangle$ where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\eta : \mathrm{id} \to T$ and $\mu : TT \to T$ are natural transformations such that the following diagrams commute:

The natural transformation η is called the *unit* of the monad and μ is called the *multiplication* of the monad.

The concepts of Kleisli triples and monads are equivalent.

Remaining sections yet to be written...

Glossary of symbols

$\times$, product of categories, 6, product object in a category, 12, product functor on sets, 24
$+$, coproduct in a category, 16, coproduct functor on sets, 24
$\dashv$, adjoint pair of functors, 46
$[- \to -]$, function space functor on sets, 24
$\dot{\to}$, natural transformation, 32
$\cdot$, horizontal composition of natural transformations, 36
$\Rightarrow$, exponential object (function space) in a category, 18
$[\mathcal{C}, \mathcal{D}]$, the category of functors from $\mathcal{C}$ to $\mathcal{D}$, 32

0, initial object in a category, 17
1, terminal object in a category, 14

Ab, the category of abelian (commutative) groups, 27

BCPO, the category of bounded-complete partial orders (bcpo's), 28
BCPOLin, the category of bcpo's and linear functions, 28

$\mathcal{C}(A, B)$, the set of morphisms between two objects in $\mathcal{C}$, 37
cABA, the category of complete atomic boolean algebras, 30
cBA, the category of complete boolean algebras, 29
$\mathcal{C}^{\mathrm{op}}$, the dual category of $\mathcal{C}$, 8

$\mathcal{D}^{\mathcal{C}}$, the category of functors from $\mathcal{C}$ to $\mathcal{D}$, 32
DCPO, the category of directed complete partial orders (dcpo's), 28
Δ, diagonal functor, 24

ϵ, a natural transformation, typically the counit of an adjunction pair, 49
η, a natural transformation, typically the unit of an adjunction pair, 49

F, a functor, typically the left adjoint in an adjunction pair, 46
$f^\sharp$, a morphism in the dual category, 8

G, a functor, typically the right adjoint in an adjunction pair, 46
Grp, the category of groups, 27

hom, the set of morphisms between two objects in a category, 37
$\mathrm{hom}_{\mathcal{C}}$, the set of morphisms between two objects in $\mathcal{C}$, 37

Mon, the category of monoids, 27

$Nat(-, -)$, the set of natural transformations between two functors, 37

$\mathcal{P}$, powerset functor, 24