Chapter 9

Natural Transformations

We've seen that, when two objects A and B are isomorphic, they generate bijections between sets of arrows, which we can now express as isomorphisms between hom-sets:

$$\mathcal{C}(A, X) \cong \mathcal{C}(B, X)$$

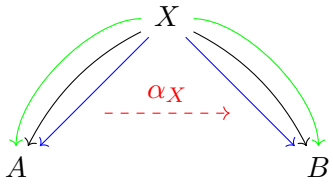
$$\mathcal{C}(X, A) \cong \mathcal{C}(X, B)$$

The converse is not true, though. An isomorphism between hom-sets does not result in an isomorphism between object *unless* additional naturality conditions are satisfied. We'll now re-formulate these naturality conditions in progressively more general settings.

9.1 Natural Transformations Between Hom-Functors

One way an isomorphism between two objects can be established is by directly providing two arrows—one the inverse of the other. But quite often it's easier to do it indirectly, by defining bijections between arrows, either the ones impinging on the two objects, or the ones emanating from the two objects.

For instance, as we've seen before, we may have, for every X, an invertible mapping of arrows α_X.



In other words, for every X, there is a mapping of hom-sets:

$$\alpha_X : \mathcal{C}(X, A) \to \mathcal{C}(X, B)$$

When we vary X, the two hom-sets become two (contravariant) functors, and α can be seen as a mapping between two functors: $\mathcal{C}(-, A)$ and $\mathcal{C}(-, B)$. Such a mapping, or a transformation, is really a family of individual mappings α_X, one per each object X in the category $\mathcal{C}$.

The functor $\mathcal{C}(-, A)$ describes the way the worlds sees A, and the functor $\mathcal{C}(-, B)$ describes the way the world sees B.

The transformation α switches back and forth between these two views. Every component of α, the bijection α_X, shows that the view of A from X is isomorphic to the view of B from X.

The naturality condition we discussed before was the condition:

$$\alpha_Y \circ (- \circ g) = (- \circ g) \circ \alpha_X$$

It relates components of α taken at different objects. In other words, it relates the views from two different observers X and Y, who are connected by the arrow $g\colon Y \to X$.

Both sides of this equation are acting on the hom-set $\mathcal{C}(X, A)$. The result is in the hom-set $\mathcal{C}(Y, B)$

Precomposition with $g\colon Y \to X$ is also a mapping of hom-sets. In fact it is the lifting of g by the contravariant hom-functor. We can write it as $\mathcal{C}(g, A)$ and $\mathcal{C}(g, B)$, respectively.

The naturality condition can therefore be rewritten as:

$$\alpha_Y \circ \mathcal{C}(g, A) = \mathcal{C}(g, B) \circ \alpha_X$$

It can be illustrated by this commuting diagram:

$$\begin{array}{ccc} \mathcal{C}(X, A) & \xrightarrow{\mathcal{C}(g,A)} & \mathcal{C}(Y, A) \\ \downarrow{\scriptstyle \alpha_X} & & \downarrow{\scriptstyle \alpha_Y} \\ \mathcal{C}(X, B) & \xrightarrow{\mathcal{C}(g,B)} & \mathcal{C}(Y, B) \end{array}$$

We can now say that an invertible transformation α between the functors $\mathcal{C}(-, A)$ and $\mathcal{C}(-, B)$ that satisfies the naturality condition is equivalent to an isomorphism between A and B.

We can follow exactly the same reasoning for the outgoing arrows. This time we start with a transformation β whose components are:

$$\beta_X\colon \mathcal{C}(A, X) \to \mathcal{C}(B, X)$$

The two (covariant) functors $\mathcal{C}(A, -)$ and $\mathcal{C}(B, -)$ describe the view of the world from the perspective of A and B, respectively. The invertible transformation β tells us that these two views are equivalent, and the naturality condition

$$(g \circ -) \circ \beta_X = \beta_Y \circ (g \circ -)$$

tells us that they behave nicely when we switch focus.

Here's the commuting diagram that illustrates the naturality condition:

$$\begin{array}{ccc} \mathcal{C}(A, X) & \xrightarrow{\mathcal{C}(A,g)} & \mathcal{C}(A, Y) \\ \downarrow{\scriptstyle \beta_X} & & \downarrow{\scriptstyle \beta_Y} \\ \mathcal{C}(B, X) & \xrightarrow{\mathcal{C}(B,g)} & \mathcal{C}(B, Y) \end{array}$$

Again, such an invertible natural transformation β establishes the isomorphism between A and B.

9.2 Natural Transformation Between Functors

The two hom-functors from the previous section were

$$FX = \mathcal{C}(A, X)$$

$$GX = \mathcal{C}(B, X)$$

They both map the category $\mathcal{C}$ to **Set**, because that's where the hom-sets live. We can say that they create two different *models* of $\mathcal{C}$ inside **Set**.

A natural transformation is a structure-preserving mapping between such models.

This idea naturally extends to functors between any pair of categories. Any two functors

$$F \colon \mathcal{C} \to \mathcal{D}$$

$$G \colon \mathcal{C} \to \mathcal{D}$$

may be seen as two different models of $\mathcal{C}$ inside $\mathcal{D}$.

To transform one model into another we connect the corresponding dots using arrows in $\mathcal{D}$.

For every object X in $\mathcal{C}$ we pick an arrow that goes from FX to GX:

$$\alpha_X \colon FX \to GX$$

A natural transformation thus maps objects to arrows.

The structure of the models, though, has as much to do with objects as it does with arrows, so let's see what happens to arrows. For every arrow $f \colon X \to Y$ in $\mathcal{C}$, there are two corresponding arrows in $\mathcal{D}$:

$$Ff \colon FX \to FY$$

$$Gf \colon GX \to GY$$

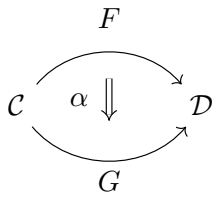
These are the two liftings of f. You can use them to move within the bounds of each of the two models. Then there are the components of α which let you switch between models.

Naturality says that it shouldn't matter whether you first move inside the first model and then jump to the second one, or first jump to the second one and then move within it. This is illustrated by the commuting *naturality square*:

$$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ \downarrow{\scriptstyle \alpha_X} & & \downarrow{\scriptstyle \alpha_Y} \\ GX & \xrightarrow{Gf} & GY \end{array}$$

A family of arrows α_X that satisfies the naturality condition is called a *natural transformation*.

This is a diagram that shows a pair of categories, two functors between them, and a natural transformation α between the functors:



Since for every arrow in $\mathcal{C}$ there is a corresponding naturality square, we can say that a natural transformation maps objects to arrows, and arrows to commuting squares.

If every component α_X of a natural transformation is an isomorphism, α is called a *natural isomorphism*.

We can now restate the main result about isomorphisms: Two objects are isomorphic if and only if there is a natural isomorphism between their hom-functors (either the covariant, or the contravariant ones—either one will do).

Natural transformations provide a very convenient high-level way of expressing commuting conditions in a variety of situations. We'll use them in this capacity to reformulate the definitions of algebraic data types.

9.3 Natural Transformations in Programming

A natural transformation is a family of arrows parameterized by objects. In programming, this corresponds to a family of functions parameterized by types, that is a *polymorphic function*.

The type of the argument to a natural transformation is constructed using one functor, and the return type using another.

In Haskell, we can define a data type that accepts two type constructors representing two functors, and produces a type of natural transformations:

```
data Natural :: (Type -> Type) -> (Type -> Type) -> Type where
  Natural :: (forall a. f a -> g a) -> Natural f g
```

The `forall` quantifier tells the compiler that the function is polymorphic—that is, it's defined for every type `a`. As long as `f` and `g` are functors, this formula defines a natural transformation.

The types defined by `forall` are very special, though. The are polymorphic in the sense of *parametric polymorphism*. It means that a single formula is used for all types. We've seen the example of the identity function, which can be written as:

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

The body of this function is very simple, just the variable `x`. It doesn't matter what type `x` is, the formula remains the same.

This is in contrast to *ad-hoc polymorphism*. An ad-hoc polymorphic function may use different implementations for different types. An example of such a function is `fmap`, the member function of the `Functor` typeclass. There is one implementation of `fmap` for lists, a different one for `Maybe`, and so on, case by case.

It turns out that limiting the type of a natural transformation to adhere to parametric polymorphism has far reaching consequences. Such a function automatically satisfies the naturality condition. It's an example of parametricity producing so called *theorems for free*.

The standard definition of a (parametric) natural transformation in Haskell uses a *type synonym:*

```
type Natural f g = forall a. f a -> g a
```

A `type` declaration introduces an alias, a shorthand, for the right-hand-side.

Here's an example of a useful function that is a natural transformation between the list functor and the `Maybe` functor:

```
safeHead :: Natural [] Maybe
safeHead [] = Nothing
safeHead (a : as) = Just a
```

(The standard library `head` function is "unsafe" in that it faults when given an empty list.)

Another example is the function `reverse`, which reverses a list. It's a natural transformation from the list functor to the list functor:

```
reverse :: Natural [] []
reverse [] = []
reverse (a : as) = reverse as ++ [a]
```

Incidentally, this is a very inefficient implementation. The actual library function uses an optimized algorithm.

A useful intuition for understanding natural transformations builds on the idea that functors acts like containers of data. There are two completely orthogonal things that you can do with a container: You can transform the data it contains, without changing the shape of the container. This is what `fmap` does. Or you can transfer the data, without modifying it, to another container. This is what a natural transformation does: It's a procedure of moving "stuff" between containers without knowing what kind of "stuff" it is.

Naturality condition enforces the orthogonality of these two operations. It doesn't matter if you first modify the data and then move it to another container; or first move it, and then modify.

This is another example of successfully decomposing a complex problem into a sequence of simpler ones. Keep in mind, though, that not every operation with containers of data can be decomposed in that way. Filtering, for instance, requires both examining the data, as well as changing the size or even the shape of the container.

On the other hand, almost every parametrically polymorphic function is a natural transformation. In some cases you may have to consider the identity or the constant functor as either source or the target. For instance, the polymorphic identity function can be though of as a natural transformation between two identity functors.

9.4 The Functor Category

Objects and arrows are drawn differently. Objects are dots and arrows are pointy lines.

In **Cat**, the category of categories, functors are drawn as arrows. But we have natural transformations that go between functors, so it looks like functors could be objects as well.

What is an arrow in one category could be an object in another.

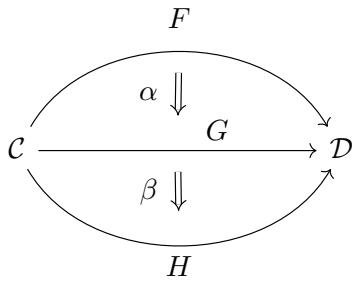
Vertical composition of natural transformations

Natural transformations can only be defined between *parallel* functors, that is functors that share the same source category and the same target category. Such parallel functors form a *functor category*. The standard notation for a functor category between two categories $\mathcal{C}$ and $\mathcal{D}$ is $[\mathcal{C}, \mathcal{D}]$, that is the names of the two categories between square brackets.

The objects in $[\mathcal{C}, \mathcal{D}]$ are functors, the arrows are natural transformations.

To show that this is indeed a category, we have to define the composition of natural transformations. This is easy if we keep in mind that components of natural transformations are regular arrows in the target category. These arrows compose.

Indeed, suppose that we have a natural transformation α between two functors F and G. We want to compose it with another natural transformation β that goes from G to H.



Let's look at the components of these transformations at some object X

$$\alpha_X \colon F\,X \to G\,X$$

$$\beta_X \colon G\,X \to H\,X$$

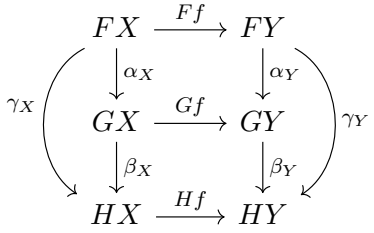
These are just two arrows in $\mathcal{D}$ that are composable. So we can define a composite natural transformation γ as follows:

$$\gamma \colon F \to H$$

$$\gamma_X = \beta_X \circ \alpha_X$$

This is called the *vertical composition* of natural transformations. You'll see it written using a dot $\gamma = \beta \cdot \alpha$ or a simple juxtaposition $\gamma = \beta\alpha$.

Naturality condition for γ can be shown by pasting together (vertically) two naturality squares for α and β:



Since the composition of natural transformations is defined in terms of composition of arrows, it is automatically associative.

There is also an identity natural transformation id_F defined for every functor F. Its component at X is the usual identity arrow at the object FX:

$$(id_F)_X = id_{FX}$$

To summarize, for every pair of categories $\mathcal{C}$ and $\mathcal{D}$ there is a category of functors $[\mathcal{C}, \mathcal{D}]$ with natural transformations as arrows.

The hom-set in that category is the set of natural transformations between two functors F and G. Following the standard notational convention, we write it as:

$$[\mathcal{C}, \mathcal{D}](F, G)$$

with the name of the category followed by the names of the two objects (here, functors) in parentheses.

Exercise 9.4.1. *Prove the naturality condition of the composition of natural transformations:*

$$\gamma_Y \circ Ff = Hf \circ \gamma_X$$

Hint: Use the definition of γ and the two naturality conditions for α and β.

Horizontal composition of natural transformations

The second kind of composition of natural transformations is induced by composition of functors. Suppose that we have a pair of composable functors

$$F \colon \mathcal{C} \to \mathcal{D} \qquad\qquad G \colon \mathcal{D} \to \mathcal{E}$$

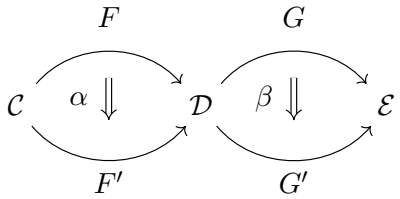
and that this pair is parallel to another pair of composable functors:

$$F' \colon \mathcal{C} \to \mathcal{D} \qquad\qquad G' \colon \mathcal{D} \to \mathcal{E}$$

We also have two natural transformations:

$$\alpha \colon F \to F' \qquad\qquad \beta \colon G \to G'$$

Pictorially:



The *horizontal composition* $\beta \circ \alpha$ maps $G \circ F$ to $G' \circ F'$.

Let's pick an object X in $\mathcal{C}$. We use α to map it to an arrow

$$\alpha_X \colon FX \to F'X$$

We can lift this arrow using G

$$G(\alpha_X) \colon G(FX) \to G(F'X)$$

What we need to define $\beta \circ \alpha$ is an arrow from $G(FX)$ to $G'(F'X)$. To get there, we can use the appropriate component of β

$$\beta_{F'X} \colon G(F'X) \to G'(F'X)$$

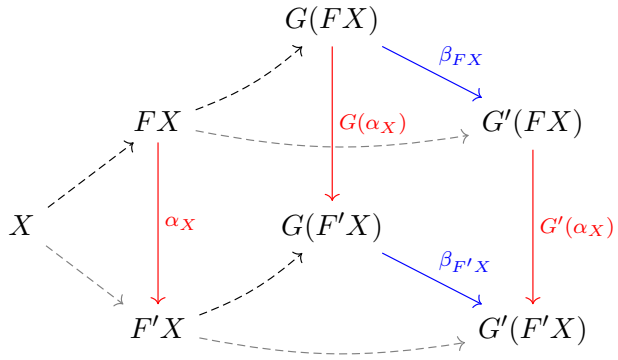
Altogether, we have

$$(\beta \circ \alpha)_X = \beta_{F'X} \circ G(\alpha_X)$$

But there is another equally plausible candidate:

$$(\beta \circ \alpha)_X = G'(\alpha_X) \circ \beta_{FX}$$

Fortunately, the are equal due to naturality of β.

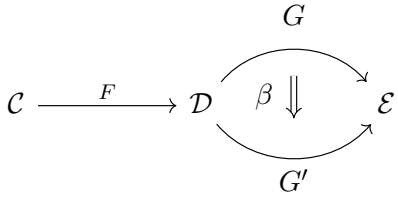


The proof of naturality of $\beta \circ \alpha$ is left as an exercise to a dedicated reader.

Whiskering

Quite often, horizontal composition is used with one of the natural transformations being the identity. There is a shorthand notation for such composition. For instance, $\beta \circ id_F$ is written as $\beta \circ F$.

Because of the characteristic shape of the diagram, such composition is called "whiskering".



In components, we have:

$$(\beta \circ F)_X = \beta_{FX}$$

Let's consider how we would translate this to Haskell. A natural transformation is a polymorphic function. Because of parametricity, it's defined by the same formula for all types. So whiskering on the right doesn't change the formula, it changes function signature.

For instance, if this is the declaration of `beta`:

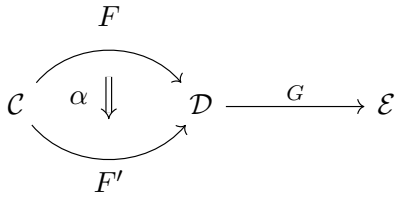
```
beta :: G x -> G' x
```

then its whiskered version would be:

```
beta_f :: G (F x) -> G' (F x)
```

Because of Haskell's type inference, this shift is often implicit.

Similarly, $id_G \circ \alpha$ is written as $G \circ \alpha$.



In components:

$$(G \circ \alpha)_X = G(\alpha_X)$$

In Haskell, the lifting of α_X by G is done using `fmap`, so given:

```
alpha :: F x -> F' x
```

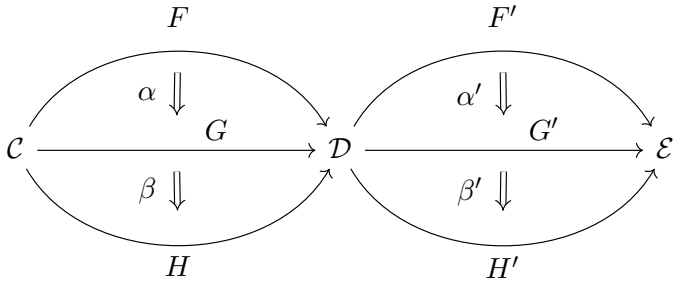
the whiskered version would be:

```
g_alpha :: G (F x) -> G (F' x)
g_alpha = fmap alpha
```

Again, Haskell's type inference engine figures out which version of `fmap` to use (here, it's the one from the `Functor` instance of `G`).

Interchange law

We can combine vertical composition with horizontal composition, as seen in the following diagram:



The interchange law states that the order of composition doesn't matter: we can first do vertical compositions and then the horizontal one, or first do the horizontal compositions and then the vertical one.

9.5 Universal Constructions Revisited

We've seen definitions of sums, products, exponentials, natural numbers, and lists.

The old-school approach to defining such data types is to explore their internals. This is the set-theory way: we look at how the elements of new sets are constructed from the elements of old sets. An element of a sum is either an element of the first set, or the second set. An element of a product is a pair of elements. And so on. We are looking at objects from the engineering point of view.

In category theory we take the opposite approach. We are not interested in what's inside the object or how it's implemented. We are interested in the purpose of the object, how it can be used, and how it interacts with other objects. We are looking at objects from the user's point of view.

Both approaches have their advantages. The categorical approach came later, because you need to study a lot of examples before clear patterns emerge. But once you see the patterns, you discover unexpected connections between things, like the duality between sums and products.

Defining particular objects through their connections requires looking at possibly infinite numbers of objects with which they interact.

"Tell me your place in the Universe, and I'll tell you who you are."

Defining an object by its mappings-out or mappings-in with respect to all objects in the category is called a *universal construction*.

Why are natural transformations so important? It's because most categorical constructions involve commuting diagrams. If we can re-cast these diagrams as naturality squares, we move one level up the abstraction ladder and gain new valuable insights.

Being able to compress a lot of facts into small elegant formulas helps us see new patterns. We'll see, for instance, that natural isomorphisms between hom-sets pop up all over category theory and eventually lead to the idea of an adjunction.

But first we'll study several examples in greater detail to get some understanding of the terse language of category theory. We'll try, for instance, to decode the statement that the sum, or the coproduct of two objects, is defined by the following natural isomorphism:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X] \cong \mathcal{C}(A + B, X)$$

Picking objects

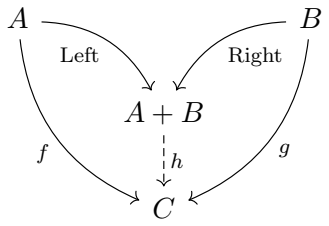
Even such a simple task as pointing at objects has a special interpretation in category theory. We have already seen that pointing at an element of a set is equivalent to selecting a function from the singleton set to it. Similarly, picking an object in a category can be replaced by selecting a functor from the single-object category. Or it can be done using a constant functor from any category.

Quite often we want to pick a pair of objects. That, too, can be accomplished by selecting a functor from a two-object stick-figure category. Similarly, picking an arrow is equivalent to selecting a functor from the "walking arrow" category, and so on.

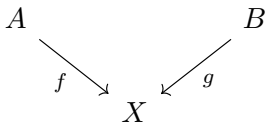
By judiciously selecting our functors and natural transformations between them, we can reformulate all the universal constructions we've seen so far.

Cospans as natural transformations

The definition of a sum requires the selection of two objects to be summed; and a third one to serve as the target of the mapping out.



This diagram can be further decomposed into two simpler shapes called *cospans*:



To construct a cospan we first have to pick a pair of objects. To do that we'll start with a two-object category **2**. We'll call its objects 1 and 2. We'll use a functor

$$D \colon \mathbf{2} \to \mathcal{C}$$

to select the objects A and B:

$$D\,1 = A$$

$$D\,2 = B$$

(D stands for "diagram", since the two objects form a very simple diagram consisting of two dots in $\mathcal{C}$.)

We'll use the constant functor

$$\Delta_X \colon \mathbf{2} \to \mathcal{C}$$

to select the object X. This functor maps both 1 and 2 to X (and the two identity arrows to id_X).

Since both functors go from **2** to $\mathcal{C}$, we can define a natural transformation α between them. In this case, it's just a pair of arrows:

$$\alpha_1 \colon D\,1 \to \Delta_X 1$$

$$\alpha_2 \colon D\,2 \to \Delta_X 2$$

These are exactly the two arrows f and g in the cospan.

Naturality condition for α is trivial, since there are no arrows (other than identities) in **2**.

There may be many cospans sharing the same three objects—meaning: there may be many natural transformations between the two functors D and Δ_X. These natural transformations form a hom-set in the functor category $[\mathbf{2}, \mathcal{C}]$, namely:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X)$$

Functoriality of cospans

Let's consider what happens when we start varying the object X in a cospan. We get a mapping from X to the set of cospans F:

$$FX = [\mathbf{2}, \mathcal{C}](D, \Delta_X)$$

This mapping turns out to be functorial in X.

To see that, consider an arrow $m\colon X \to Y$. The lifting of this arrow is a mapping between two sets of natural transformations:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X) \to [\mathbf{2}, \mathcal{C}](D, \Delta_Y)$$

This might look very abstract until you remember that natural transformations have components, and these components are just regular arrows. An element of the left-hand side is a natural transformation:

$$\mu\colon D \to \Delta_X$$

It has two components corresponding to the two objects in $\mathbf{2}$. For instance, we have

$$\mu_1\colon D\,1 \to \Delta_X 1$$

or, using the definitions of D and Δ:

$$\mu_1\colon A \to X$$

This is just the arrow f in our diagram.

Similarly, the element of the right-hand side is:

$$\nu\colon D \to \Delta_Y$$

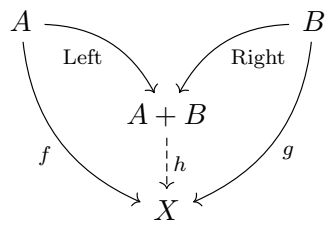
Its component at 1 is an arrow

$$\nu_1\colon A \to Y$$

We can get from μ_1 to ν_1 simply by post-composing it with $m\colon X \to Y$. So the lifting of h is a component-by-component post-compositon $(m \circ -)$.

Sum as a universal cospan

Of all the cospans that you can build on the pair A and B, the one with the arrows we called $Left$ and $Right$ converging on $A + B$ is very special. There is a unique mapping out of it to any other cospan—a mapping that makes two triangles commute.



We are now in a position to translate this condition into a statement about natural transformations and hom-sets. The arrow h is an element of the hom-set

$$\mathcal{C}(A + B, X)$$

A cospan centered at X is a natural transformation, that is an element of the hom-set in the functor category:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X)$$

Both are hom-sets in their respective categories. But they are just sets, that is objects in the category **Set**. This category forms a bridge between the functor category $[\mathbf{2}, \mathcal{C}]$ and a "regular" category $\mathcal{C}$, even though, conceptually, they seem to be at very different levels of abstraction.

As Lao Tzu would say, "Sometimes a set is just a set."

Our universal construction is the bijection or the isomorphism of sets:

$$[\mathbf{2}, \mathcal{C}](D, \Delta_X) \cong \mathcal{C}(A + B, X)$$

Moreover, if we vary the object X, the two sides behave like functors from $\mathcal{C}$ to **Set**. Therefore it makes sense to ask if this mapping of functors is a natural isomorphism.

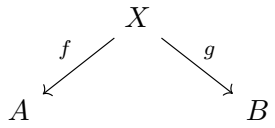
Indeed, it can be shown that the naturality condition for this isomorphism translates into commuting conditions for the triangles in the definition of the sum. So the definition of the sum can be replaced by a single equation.

Product as a universal span

An analogous argument can be made about the universal construction of the product. Again, we start with the stick-figure category **2** and the functor D. But this time we use a natural transformation going in the opposite direction

$$\alpha \colon \Delta_X \to D$$

Such a natural transformation is a pair of arrows that form a *span*:



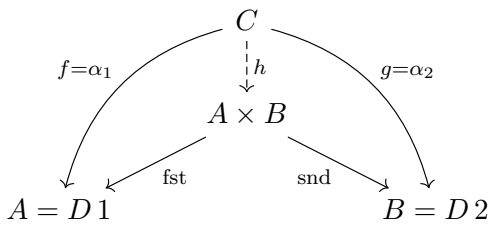
Collectively, these natural transformations form a hom-set in the functor category :

$$[\mathbf{2}, \mathcal{C}](\Delta_X, D)$$

Every element of this hom-set is in one-to-one correspondence with a unique mapping h into the product $A \times B$. Such a mapping is a member of the hom-set $\mathcal{C}(X, A \times B)$. This correspondence is expressed as the isomorphism:

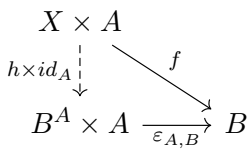
$$[\mathbf{2}, \mathcal{C}](\Delta_X, D) \cong \mathcal{C}(X, A \times B)$$

It can be shown that the naturality of this isomorphism guarantees that the triangles in this diagram commute:



Exponentials

The exponentials, or function objects, are defined by this commuting diagram:



Here, f is an element of the hom-set $\mathcal{C}(X \times A, B)$ and h is an element of $\mathcal{C}(X, B^A)$.

The isomorphism between these sets, natural in X, defines the exponential object.

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

The f in the diagram above is an element of the left-hand side, and h is the corresponding element of the right-hand side. The transformation α maps f to h. In Haskell, we call it `curry`. Its inverse, α^{-1} is known as `uncurry`.

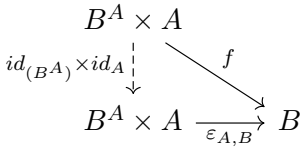
Unlike in the previous examples, here both hom-sets are in the same category, and it's easy to analyze the isomorphism in more detail. In particular, we'd like to see how the commuting condition:

$$f = \varepsilon_{A,B} \circ (h \times id_A)$$

arises from naturality.

The standard Yoneda trick is to make a substitution for X that would reduce one of the hom-sets to an endo-hom-set, that is a hom-set whose source is the same the target. This will allow us to pick a special element of that hom-set, namely the identity arrow.

In our case, substituting B^A for X will allow us to pick $h = id_{(B^A)}$.



The commuting condition in this case tells us that $f = \varepsilon_{A,B}$. In other words, we get the formula for $\varepsilon_{A,B}$ in terms of α:

$$\varepsilon_{A,B} = \alpha^{-1}(id_{(B^A)})$$

Since we recognize α^{-1} as `uncurry`, and ε as function application, we can write it in Haskell as:

```
apply :: (a -> b, a) -> b
apply = uncurry id
```

This my be surprising at first, until you realize that the currying of `(a->b,a)->b` leads to `(a->b)->(a->b)`.

We can also encode the two sides of the main isomorphism as Haskell functors:

```
data LeftFunctor  a b x = LF ((x, a) -> b)
```

```
data RightFunctor a b x = RF (x -> (a -> b))
```

They are both contravariant functors in the type variable `x`.

```
instance Contravariant (LeftFunctor a b) where
  contramap g (LF f) = LF (f . bimap g id)
```

This says that the lifting of $g\colon X \to Y$ acting on $\mathcal{C}(Y \times A, B)$ is given by

$$(- \circ (g \times id_A))\colon \mathcal{C}(Y \times A, B) \to \mathcal{C}(X \times A, B)$$

Similarly

```
instance Contravariant (RightFunctor a b) where
  contramap g (RF h) = RF (h . g)
```

translates to

$$(- \circ g)\colon \mathcal{C}(Y, B^A) \to \mathcal{C}(X, B^A)$$

The natural transformation α is just a thin encapsulation of `curry`; and its inverse is `uncurry`:

```
alpha :: forall a b x. LeftFunctor a b x -> RightFunctor a b x
alpha (LF f) = RF (curry f)
```

```
alpha_1 :: forall a b x. RightFunctor a b x -> LeftFunctor a b x
alpha_1 (RF h) = LF (uncurry h)
```

Using the two formulas for the lifting of $g\colon X \to Y$, here's the naturality square:

$$\begin{array}{ccc} \mathcal{C}(Y \times A, B) & \xrightarrow{(-\circ(g\times id_A))} & \mathcal{C}(X \times A, B) \\ \downarrow{\scriptstyle \alpha_Y} & & \downarrow{\scriptstyle \alpha_X} \\ \mathcal{C}(Y, B^A) & \xrightarrow{(-\circ g)} & \mathcal{C}(X, B^A) \end{array}$$

Let's now apply the Yoneda trick to it and replace Y with B^A. This also allows us to substitute g, which now goes for X to B^A, with h.

$$\begin{array}{ccc} \mathcal{C}(B^A \times A, B) & \xrightarrow{(-\circ(h\times id_A))} & \mathcal{C}(X \times A, B) \\ \downarrow{\scriptstyle \alpha_{(B^A)}} & & \downarrow{\scriptstyle \alpha_X} \\ \mathcal{C}(B^A, B^A) & \xrightarrow{(-\circ h)} & \mathcal{C}(X, B^A) \end{array}$$

We know that the hom-set $\mathcal{C}(B^A, B^A)$ contains at least the identity arrow, so we can pick the element $id_{(B^A)}$ in the lower left corner.

α^{-1} acting on it produces $\varepsilon_{A,B}$ in the upper left corner (that's the `uncurry id` trick).

Pre-composition with h acting on identity produces h in the lower right corner.

α^{-1} acting on h produces f in the upper right corner.

$$\begin{array}{ccc} \varepsilon_{A,B} & \xmapsto{(-\circ(h\times id_A))} & f \\ {\scriptstyle\alpha^{-1}}\uparrow & & \uparrow{\scriptstyle\alpha^{-1}} \\ id_{(B^A)} & \xmapsto{(-\circ h)} & h \end{array}$$

(The $\mapsto$ arrows denote the action of functions on elements of sets.)

So the selection of $id_{(B^A)}$ in the lower left corner fixes the other three corners. In particular, we can see that the upper arrow applied to $\varepsilon_{A,B}$ produces f, which is exactly the commuting condition:

$$\varepsilon_{A,B} \circ (h \times id_A) = f$$

the one that we set out to derive.

9.6 Limits and Colimits

In the previous section we defined the sum and the product using natural transformations. These were transformations between diagrams defined as functors from a very simple stick-figure category **2**, one of them being the constant functor.

Nothing prevents us from replacing the category **2** with something more complex. For instance, we could try categories that have non-trivial arrows between objects, or categories with infinitely many objects.

There is a whole vocabulary built around such constructions.

We used objects in the category **2** for indexing objects in the category $\mathcal{C}$. We can replace **2** with an arbitrary indexing category **I**. A diagram in $\mathcal{C}$ is still defined as a functor $D\colon \mathbf{I} \to \mathcal{C}$. It picks objects in $\mathcal{C}$ as well as some of the arrows between them.

As the second functor we'll still use the constant functor $\Delta_X\colon \mathbf{I} \to \mathcal{C}$.

A natural transformation that's an element of the hom-set

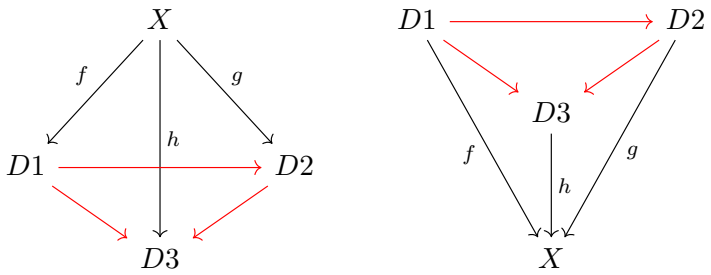
$$[\mathbf{I}, \mathcal{C}](\Delta_X, D)$$

is now called a *cone*. Its dual, an element of

$$[\mathbf{I}, \mathcal{C}](D, \Delta_X)$$

is called a *cocone*. They generalize the span and the cospan, respectively.

Diagramatically, cones and cocones look like this:



Since the indexing category may now contain arrows, the naturality conditions for these diagrams are no longer trivial. The constant functor Δ_X shrinks all vertices to one, so naturality squares turn into triangles. Naturality means that all triangles with X in their apex must now commute.

The universal cone, if it exists, is called the *limit* of the diagram D, and is written as Lim_D. Universality means that it satisfies the following isomorphism, natural in X:

$$[\mathbf{I}, \mathcal{C}](\Delta_X, D) \cong \mathcal{C}(X, \mathrm{Lim}_D)$$

Dually, the universal cocone is called a *colimit*, and is described by the following natural isomorphism:

$$[\mathbf{I}, \mathcal{C}](D, \Delta_X) \cong \mathcal{C}(\mathrm{Colim}_D, X)$$

We can now say that a product is a limit, and a sum is a colimit, of a diagram from the indexing category **2**.

Limits and colimits distill the essence of a pattern.

A limit, like a product, it is defined by its mapping-in property.

A colimit, like a sum, it is defined by its mapping out property.

There are many interesting limits and colimits, and we'll see some when we discuss algebras and coalgebras.

Exercise 9.6.1. *Show that the limit of a "walking arrow" category, that is a two-object category with an arrow connecting the two objects, has the same elements as the first object in the diagram ("elements" are the arrows from the terminal object).*

9.7 The Yoneda Lemma

A functor from some category $\mathcal{C}$ to the category of sets can be thought of as a model of this category in **Set**. Modeling, in general, is a lossy process: it discards some information. A constant functor is an extreme example: it maps the whole category to a single set and its identity function.

A hom-functor produces a model of the category as viewed from a certain vantage point. The functor $\mathcal{C}(A, -)$, for instance, offers the panorama of $\mathcal{C}$ from the vantage point of A. It organizes all the arrows emanating from A into neat packages that are connected by images of arrows that go between them, all in accordance with the original structure of the source category.

Some vantage points are better than others. For instance, the view from the initial object is quite sparse. Every object X is mapped to a singleton set corresponding to the unique mapping $0 \to X$.

The view from the terminal object is more interesting: it maps all objects to their sets of (global) elements.

The Yoneda lemma may be considered one of the most profound statements, or one of the most trivial statements in category theory. Let's start with the profound version.

Consider two models of $\mathcal{C}$ in **Set**: one given by the hom-functor $\mathcal{C}(A, -)$, that is the panoramic view of $\mathcal{C}$ from the vantage point of A; and another given by some functor $F \colon \mathcal{C} \to \mathbf{Set}$. A natural transformation between them embeds one model in the other. It turns out that the set of such natural transformations is fully determined by the value of F at A.

The set of natural transformation is the hom-set in the functor category $[\mathcal{C}, \mathbf{Set}]$, so this is the formal statement of the Yoneda lemma:

$$[\mathcal{C}, \mathbf{Set}](\mathcal{C}(A, -), F) \cong FA$$

The reason this works is because all the mappings involved in this theorem are bound by the requirements of preserving the structure of the category $\mathcal{C}$ and the structure of its models. In particular, naturality conditions impose a huge set of constraints on the way the mapping propagates from one point to another.

The proof of the Yoneda lemma starts with a single identity arrow and lets naturality propagate it across the whole category.

Here's the sketch of the proof. It consists of two parts: First, given a natural transformation we construct an element of FA. Second, given an element of FA we construct the corresponding natural transformation.

First, let's pick an arbitrary element on the left-hand side: a natural transformation α. Its component at X is a function

$$\alpha_X : \mathcal{C}(A, X) \to FX$$

We can now apply the Yoneda trick: substitute A for X and pick the identity id_A as the element of $\mathcal{C}(A, A)$. This gives us an element $\alpha_A(id_A)$ in the set FA.

Now the other way around. Take an element y of the set FA. We want to implement a natural transformation that takes an arrow h from $\mathcal{C}(A, X)$ and produces an element of FX. This is simply done by lifting the arrow h using F. We get a function

$$Fh : FA \to FX$$

We can apply this function to y to get an element of FX. We take this element as the action of α_X on h.

Exercise 9.7.1. *Show that the mapping*

$$\mathcal{C}(A, X) \to FX$$

defined above is a natural transformation. Hint: Vary X using some $f : X \to Y$.

The isomorphism in the Yoneda lemma is natural not only in A but also in F. In other words, you can "move" from the functor F to another functor G by applying an arrow in the functor category, that is a natural transformation. This is quite a leap in the levels of abstraction, but all the definitions of functoriality and naturality work equally well in the functor category, where objects are functors, and arrows are natural transformations.

Yoneda lemma in programming

Now for the trivial part: The proof of the Yoneda lemma translates directly to Haskell code. We start with the type of natural transformation between the hom-functor `a->x` and some functor `f`, and show that it's equivalent to the type of `f` acting on `a`.

```
forall x. (a -> x) -> f x.    -- is isomorphic to (f a)
```

We produce a value of the type `f a` using the standard Yoneda trick

```
yoneda :: Functor f => (forall x. (a -> x) -> f x) -> f a
yoneda g = g id
```

Here's the inverse mapping:

```
yoneda_1 :: Functor f => f a -> (forall x. (a -> x) -> f x)
yoneda_1 y = \h -> fmap h y
```

Note that we are cheating a little by mixing types and sets. The Yoneda lemma in the present formulation works with **Set**-valued functors. Again, the correct incantation is to say that we use the enriched version of the Yoneda lemma in a self-enriched category.

The Yoneda lemma has some interesting applications in programming. For instance, let's consider what happens when we apply the Yoneda lemma to the identity functor. We get the isomorphism between the type `a` (the identity functor acting on `a`) and

```
forall x. (a -> x) -> x
```

We interpret this as saying that any data type `a` can be replaced by a higher order polymorphic function. This function takes another function—called a handler, a callback, or a *continuation*—as an argument.

This is the standard continuation passing transformation that's used a lot in distributed programming, when the value of type `a` has to be retrieved from a remote server. It's also useful as a program transformation that turns recursive algorithms into tail-recursive functions.

Continuation-passing style is difficult to work with because the composition of continuations is highly nontrivial, resulting in what programmers often call a "callback hell." Fortunately continuations form a monad, which means their composition can be automated.

The contravariant Yoneda lemma

By reversing a few arrow, the Yoneda lemma can be applied to contravariant functors as well. It works on natural transformations between the contravariant hom-functor $\mathcal{C}(-, A)$ and a contravariant functor F:

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, A), F) \cong FA$$

This is the Haskell implementation of the mapping:

```
coyoneda :: Contravariant f => (forall x. (x -> a) -> f x) -> f a
coyoneda g = g id
```

And this is the inverse transformation:

```
coyoneda_1 :: Contravariant f => f a -> (forall x. (x -> a) -> f x)
coyoneda_1 y = \h -> contramap h y
```

9.8 Yoneda Embedding

In a closed category, we have exponential objects that serve as stand-ins for hom-sets. This is obviously a thing in the category of sets, where hom-sets, being sets, are automatically objects. But in the category of categories **Cat**, hom-sets are sets of functors, and it's not immediately obvious that they can be promoted to objects—that is categories. But, as we've seen, they can! Functors between any two categories form a functor category, with natural transformations as arrows.

Because of that, it's possible to curry functors just like we curried functions. A functor from a product category can be viewed as a functor returning a functor. In other words, **Cat** is a closed (symmetric) monoidal category.

In particular, we can apply currying to the hom-functor $\mathcal{C}(A, B)$. It is a profunctor, or a functor from the product category:

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

But it's also a contravariant functor in A. For every A in $\mathcal{C}^{op}$ it produces a functor $\mathcal{C}(A, -)$, that is an object in the functor category $[\mathcal{C}, \mathbf{Set}]$. We can write this mapping as:

$$\mathcal{C}^{op} \to [\mathcal{C}, \mathbf{Set}]$$

Alternatively, we can focus on B and get a contravariant functor $\mathcal{C}(-, B)$. This mapping can be written as

$$\mathcal{C} \to [\mathcal{C}^{op}, \mathbf{Set}]$$

Both mappings are functorial, which means that, for instance, an arrow in $\mathcal{C}$ is mapped to a natural transformation in $[\mathcal{C}^{op}, \mathbf{Set}]$.

These **Set**-valued functor categories are common enough that they have special names. The functors in $[\mathcal{C}^{op}, \mathbf{Set}]$ are called *presheaves*, and the ones in $[\mathcal{C}, \mathbf{Set}]$ are called *co-presheaves*. (The names come from algebraic topology.)

Let's focus our attention on the following reading of the hom-functor:

$$\mathcal{Y} \colon \mathcal{C} \to [\mathcal{C}^{op}, \mathbf{Set}]$$

It takes an object X and maps it to a presheaf $\mathcal{C}(-, X)$, which can be visualized as the totality of views of X from all possible directions.

Let's also review its action on arrows. The functor $\mathcal{Y}$ lifts an arrow $f \colon X \to Y$ to a mapping of presheaves:

$$\alpha \colon \mathcal{C}(-, X) \to \mathcal{C}(-, Y)$$

The component of this natural transformation at some Z is a function between hom-sets:

$$\alpha_Z \colon \mathcal{C}(Z, X) \to \mathcal{C}(Z, Y)$$

which is simply implemented as the post-composition $(f \circ -)$.

Such a functor $\mathcal{Y}$ can be thought of as creating a model of $\mathcal{C}$ in the presheaf category. But this is no run-of-the-mill model—it's an *embedding* of one category inside another. This particular one is called the *Yoneda embedding*.

First of all, every object of $\mathcal{C}$ is mapped to a different object (presheaf) in $[\mathcal{C}^{op}, \mathbf{Set}]$. We say that it's "injective on objects." But that's not all: every arrow in $\mathcal{C}$ is mapped to a different arrow. We say that the embedding functor is *faithful*. If that weren't enough,

the mapping of hom-sets is also surjective, meaning that every arrow between objects in $[\mathcal{C}^{op}, \mathbf{Set}]$ comes from some arrow in $\mathcal{C}$. We say that the functor is *full*. Altoghether, the embedding is *fully faithful*.

The latter fact is the direct consequence of the Yoneda lemma. We know that, for any functor $F\colon \mathcal{C}^{op} \to \mathbf{Set}$, we have a natural isomorphism:

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, X), F) \cong FX$$

In particular, we can substitute another hom-functor $\mathcal{C}(-, Y)$ for F:

$$[\mathcal{C}^{op}, \mathbf{Set}](\mathcal{C}(-, X), \mathcal{C}(-, Y)) \cong \mathcal{C}(X, Y)$$

The left-hand side is the hom-set in the presheaf category and the right-hand side is the hom-set in $\mathcal{C}$. They are isomorphic, which proves that the embedding is fully faithful.

Let's have a closer look at this isomorphism. Let's pick an element of the set $\mathcal{C}(X, Y)$—an arrow f. The isomorphism maps it to a natural transformation whose component at Z is a function:

$$\mathcal{C}(Z, X) \to \mathcal{C}(Z, Y)$$

This mapping is implemented as post-composition $(f \circ -)$.

In Haskell, we would write it as:

```
toNat :: (x -> y) -> (forall z. (z -> x) -> (z -> y))
toNat f = \h -> f . h
```

In fact, this syntax works too:

```
toNat f = (f . )
```

The inverse mapping is:

```
fromNat :: (forall z. (z -> x) -> (z -> y)) -> (x -> y)
fromNat alpha = alpha id
```

(Notice the use of the Yoneda trick again.)

This isomorphism maps identity to identity and composition to composition. That's because it's implemented as post-composition, and post-composition preserves both identity and composition. We've seen this in the chapter on isomorphisms:

$$((f \circ g) \circ -) = (f \circ -) \circ (g \circ -)$$

Because it preserves composition and identity, this isomorphism also preserves *isomorphisms*. So if X is isomorphic to Y then the presheaves $\mathcal{C}(-, X)$ and $\mathcal{C}(-, Y)$ are isomorphic, and vice versa.

This is exactly the result that we've been using all along to prove numerous isomorphisms in previous chapters.

9.9 Representable Functors

Objects in a co-presheaf category are functors that assign sets to objects in $\mathcal{C}$. Some of these functors work by picking a reference object A and assigning, to all objects X, their hom-sets $\mathcal{C}(A, X)$. Such functors, and all the functors isomorphic to those, are called *representable*. The whole functor is "represented" by a single object A.

In a closed category, the functor which assigns the set of elements of X^A to every object X is represented by A, because the set of elements of X^A is isomorphic to $\mathcal{C}(A, X)$:

$$\mathcal{C}(1, X^A) \cong \mathcal{C}(1 \times A, X) \cong \mathcal{C}(A, X)$$

Seen this way, the representing object A is like a logarithm of a functor.

The analogy goes deeper: just like a logarithm of a product is a sum of logarithms, a representing object for a product data type is a sum. For instance, the functor that squares its argument using a product, $Fx = x \times x$, is represented by 2, which is the sum $1 + 1$.

Representable functors play a very special role in the category of **Set**-valued functors. Notice that the Yoneda embedding maps objects of $\mathcal{C}$ to representable presheaves. It maps an object X to a presheaf represented by X:

$$\mathcal{Y}\colon X \mapsto \mathcal{C}(-, X)$$

We can find the entire category $\mathcal{C}$, objects and morphisms, embedded inside the presheaf category as representable functors. The question is, what else is there in the presheaf category "in between" representable functors?

Just like rational numbers are dense among real numbers, so representables are "dense" among (co-) presheaves. Every real number may be approximated by rational numbers. Every presheaf is a colimit of representables (and every co-presheaf, a limit). We'll come back to this topic when we talk about (co-) ends.

The guessing game

The idea that objects can be described by the way they interact with other objects is sometimes illustrated by playing imaginary guessing games. One category theorist picks a secret object in a category, and the other has to guess which object it is (up to isomorphism, of course).

The guesser is allowed to point at objects, and use them as "probes" into the secret object. The opponent is supposed to respond, each time, with a set: the set of arrows from the probing object A to the secret object X. This, of course, is the hom-set $\mathcal{C}(A, X)$.

The totality of these answers, as long as the opponent is not cheating, will define a presheaf $F\colon \mathcal{C} \to \mathbf{Set}$, and the object they are hiding is its representing object.

But how do we know they are not cheating? To test that, we have to be able to ask questions about arrows. For every arrow we select, they should give us a function between two sets—the sets they gave us for its endpoints. We can then check if all identity arrows are mapped to identity functions, and whether compositions of arrows map to compositions of functions. In other words, we'll be able to verify that F is a functor.

However, a clever enough opponent may still fool us. The presheaf they are revealing to us may describe a fantastical object—a figment of their imagination—and we won't be able to tell. It turns out that such imaginary objects are often as interesting as the real ones.

Representable functors in programming

In Haskell, we define a class of representable functors using two functions that witness the isomorphism: `tabulate` turns a function into a lookup table, and `index` uses the representing type `Key` to index into it.

```
class Representable f where
  type Key f :: Type
  tabulate :: (Key f -> a) -> f a
  index    :: f a -> (Key f -> a)
```

Algebraic data types that use sums are not representable—there is no formula for taking a logarithm of a sum. List type is defined as a sum, so it's not representable.

However, an infinite stream is. Conceptually, such a stream is like an infinite tuple, which is technically a product. A stream is represented by the type of natural numbers. In other words, an infinite stream is equivalent to a mapping out of natural numbers.

```
data Stream a = Stm a (Stream a)
```

Here's the instance definition:

```
instance Representable Stream where
  type Key Stream = Nat
  tabulate g = tab Z
    where
      tab n = Stm (g n) (tab (S n))
  index stm = \n -> ind n stm
    where
      ind Z (Stm a _) = a
      ind (S n) (Stm _ as) = ind n as
```

Representable types are useful in implementing memoization of functions.

Exercise 9.9.1. *Implement the `Representable` instance for `Pair`:*

```
data Pair x = Pair x x
```

Exercise 9.9.2. *Is the constant functor that maps everything to the terminal object representable? Hint: what's the logarithm of 1?*

In Haskell, such a functor could be implemented as:

```
data Unit a = U
```

Implement the instance of `Representable` for it.

Exercise 9.9.3. *The list functor is not representable. But can it be considered a sum or representables?*

9.10 2-category Cat

In the category of categories, **Cat**, the hom-sets are not just sets. Each of them can be promoted to a functor category, with natural transformations playing the role of arrows. This kind of structure is called a 2-category.

In the language of 2-categories, objects are called 0-cells, arrows between them are called 1-cells, and arrows between arrows are called 2-cells.

The obvious generalization of that picture would be to have 3-cells that go between 2-cells and so on. An n-category has cells going up to the n-th level.

But why not have arrows all the way down? Enter infinity categories. Far from being a curiosity, ∞-categories have practical applications. For instance they are used in algebraic topology to describe points, paths between points, surfaces swiped by paths, volumes swiped by surfaces, and so on, ad infinitum.