10

Natural Transformations

We talked about functors as mappings between categories that preserve their structure.

A functor "embeds" one category in another. It may collapse multiple things into one, but it never breaks connections. One way of thinking about it is that with a functor we are modeling one category inside another. The source category serves as a model, a blueprint, for some structure that's part of the target category.

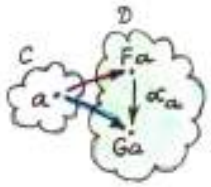
10

Natural Transformations

ファンクターとは、カテゴリ間の構造を保持するマッピングのことである。ファンクタは、あるカテゴリーを別のカテゴリーに「埋め込む」。複数のものを1つにまとめることはあっても、つながりが壊れることはない。一つの考え方として，ファンクタは，あるカテゴリを別のカテゴリの中にモデル化しているのである．元のカテゴリは、対象となるカテゴリの一部である構造のモデル、青写真として機能する。

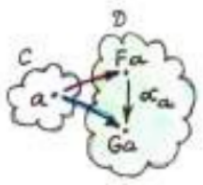
There may be many ways of embedding one category in another. Sometimes they are equivalent, sometimes very different. One may collapse the whole source category into one object, another may map every object to a different object and every morphism to a different morphism. The same blueprint may be realized in many different ways. Natural transformations help us compare these realizations. They are mappings of functors — special mappings that preserve their functorial nature.

Consider two functors *F* and *G* between categories **C** and **D**. If you focus on just one object *a* in **C**, it is mapped to two objects: *Fa* and *Ga*. A mapping of functors should therefore map *Fa* to *Ga*.



Notice that *Fa* and *Ga* are objects in the same category **D**. Mappings between objects in the same category should not go against the grain of

あるカテゴリーを別のカテゴリーに埋め込むには、いろいろな方法があるだろう。それらは等価であることもあれば、非常に異なることもあります。あるものは元のカテゴリー全体を一つのオブジェクトに畳み込むかもしれないし、別のものはすべてのオブジェクトを別のオブジェクトに、すべてのモルヒズムを別のモルヒズムに写すかもしれない。同じ設計図が多くの異なる方法で実現されるかもしれない。自然変換は、これらの実現方法を比較するのに役立つ。自然変換はファンクタの写像であり、そのファンクタとしての性質を保つ特別な写像である。カテゴリー□と　の間の2つのファンクター□とを考えてみよう。　の中の1つのオブジェクト　に注目すると、それは□と□という2つのオブジェクトに写像されるのである。したがって、ファンクタの写像は ŭ を　Lu_1D44E に写像する必要がある。



□と□は同じカテゴリ□にあるオブジェクトであることに注目しましょう。同じカテゴリに属するオブジェクト間のマッピングは、以下に反するものであってはならない。

the category. We don't want to make artificial connections between objects. So it's *natural* to use existing connections, namely morphisms. A natural transformation is a selection of morphisms: for every object a, it picks one morphism from Fa to Ga. If we call the natural transformation α, this morphism is called the *component* of α at a, or α_a.

$$\alpha_a :: Fa \to Ga$$

Keep in mind that a is an object in **C** while α_a is a morphism in **D**.

If, for some a, there is no morphism between Fa and Ga in **D**, there can be no natural transformation between F and G.

Of course that's only half of the story, because functors not only map objects, they map morphisms as well. So what does a natural transformation do with those mappings? It turns out that the mapping of morphisms is fixed — under any natural transformation between F and G, $F\ f$ must be transformed into $G\ f$. What's more, the mapping of morphisms by the two functors drastically restricts the choices we have in defining a natural transformation that's compatible with it. Consider a morphism f between two objects a and b in **C**. It's mapped to two morphisms, $F\ f$ and Gf in **D**:

$$Ff :: Fa \to Fb$$
$$Gf :: Ga \to Gb$$

The natural transformation α provides two additional morphisms that complete the diagram in D:

$$\alpha_a :: Fa \to Ga$$
$$\alpha_b :: Fb \to Gb$$

に反するものであってはならない。我々は、オブジェクト間の人工的な接続を作りたくないのです。だから、既存の接続、すなわちモルヒズムを使うのが自然である。自然変換は形態素の選択である：すべてのオブジェクト□に対して、　から□への形態素を一つ選ぶのである。自然変換を　□　と呼ぶと、この形態素は　□　の　　Ll_lD44E での成分、あるいは　□　と呼ばれる。

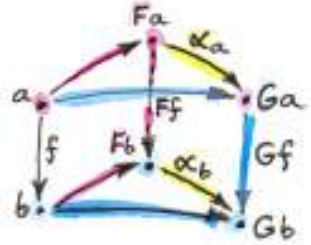
$$\alpha_a :: Fa \to Ga$$

Ll_lD44E は　□　のオブジェクトであり、□　　Ll_lD44E は　□　のモーフィズムであることに留意してください。ある　　L144　　に対して、□　と　□　の間の形態素がない場合、□　と　　Lu_lD43A の間の自然変換は存在し得ない。もちろん、ファンクタは対象を写すだけでなく、形態素も写すので、これは話の半分に過ぎない。では、自然変換はこれらの写像をどのように扱うのでしょうか？□と　Lu_lD43A の間の自然変換の下では、　は　Lu_lD43A に変換されなければならないのです。さらに、2つのファンクターによるモルヒズムの写像は、それと互換性のある自然変換を定義する際の選択肢を大幅に制限するものである。□と　の間のモルヒズム□を考えてみましょう。これは□と□の2つのモルヒズムに写像される。

$$Ff :: Fa \to Fb$$
$$Gf :: Ga \to Gb$$

自然変換 ũ はDのダイアグラムを完成させる2つの追加的な形態素を提供する。

$$\alpha_a :: Fa \to Ga$$
$$\alpha_b :: Fb \to Gb$$

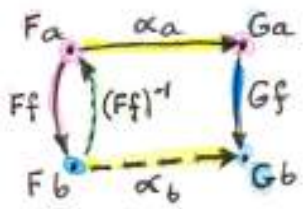


Now we have two ways of getting from Fa to Gb. To make sure that they are equal, we must impose the *naturality condition* that holds for any f:

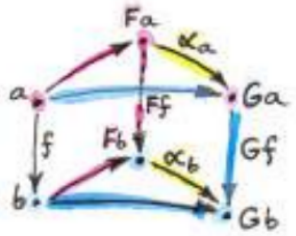
$$Gf \circ \alpha_a = \alpha_b \circ Ff$$

The naturality condition is a pretty stringent requirement. For instance, if the morphism Ff is invertible, naturality determines α_b in terms of α_a. It *transports* α_a along f:

$$\alpha_b = (Gf) \circ \alpha_a \circ (Ff)^{-1}$$



If there is more than one invertible morphism between two objects, all these transports have to agree. In general, though, morphisms are not

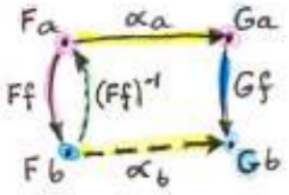


さて、ŭから☒へ行く方法は2つある。それらが等しいことを確認するために、任意の　　に対して成立する自然性条件を課す必要があります。

$$Gf \circ \alpha_a = \alpha_b \circ Ff$$

自然性条件はかなり厳しい条件である。例えば、モルヒズム ☒ が反転可能であれば、自然性は ☒ の観点から　　を決定する。これは ☒ を ☒ に沿って輸送します。

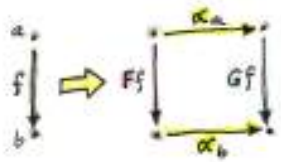
$$\alpha_b = (Gf) \circ \alpha_a \circ (Ff)^{-1}$$



2つのオブジェクトの間に複数の反転可能なモルヒズムがある場合、これらの輸送はすべて一致しなければならない。しかし、一般に、モルヒズムは

invertible; but you can see that the existence of natural transformations between two functors is far from guaranteed. So the scarcity or the abundance of functors that are related by natural transformations may tell you a lot about the structure of categories between which they operate. We'll see some examples of that when we talk about limits and the Yoneda lemma.

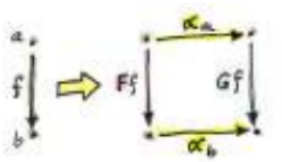
Looking at a natural transformation component-wise, one may say that it maps objects to morphisms. Because of the naturality condition, one may also say that it maps morphisms to commuting squares — there is one commuting naturality square in **D** for every morphism in **C**.



This property of natural transformations comes in very handy in a lot of categorical constructions, which often include commuting diagrams. With a judicious choice of functors, a lot of these commutativity conditions may be transformed into naturality conditions. We'll see examples of that when we get to limits, colimits, and adjunctions.

Finally, natural transformations may be used to define isomorphisms of functors. Saying that two functors are naturally isomorphic is almost like saying they are the same. *Natural isomorphism* is defined as a natural transformation whose components are all isomorphisms (invertible morphisms).

しかし，2つのファンクタの間に自然な変換が存在することは保証されていないことがおわかりいただけると思います．ですから、自然変換によって関係付けられるファンクタの少なさ、多さは、それが作用するカテゴリの構造について多くのことを教えてくれるかもしれません。極限と米田レンマの話をするときに、その例をいくつか見てみよう。自然変換を構成要素ごとに見ると、オブジェクトをモルヒズムに写すと言うことができる。自然性の条件から、モルヒズムを交番正方形に写すと言うこともできる － □のすべてのモルヒズムに対して□の交番自然性正方形が1つ存在するのだ。



自然変換のこの性質は、多くのカテゴリ構造で非常に便利であり、それはしばしば交叉図を含む。ファンクタの賢明な選択によって、これらの可換性の条件の多くは自然性の条件に変換されるかもしれません。極限、コリミット、アジャンクションのところで、その例を見ることにしよう。最後に，自然変換はファンクタの同型性を定義するのに使われることがある．2つのファンクターが自然同型であると言うことは、それらが同じであると言うこととほとんど同じです。自然同型は，構成要素がすべて同型（反転可能な形態素）である自然変換として定義される．

10.1 Polymorphic Functions

We talked about the role of functors (or, more specifically, endofunctors) in programming. They correspond to type constructors that map types to types. They also map functions to functions, and this mapping is implemented by a higher order function `fmap` (or `transform`, `then`, and the like in C++).

To construct a natural transformation we start with an object, here a type, `a`. One functor, `F`, maps it to the type *Fa*. Another functor, `G`, maps it to *Ga*. The component of a natural transformation `alpha` at `a` is a function from *Fa* to *Ga*. In pseudo-Haskell:

```
alpha_a :: F a -> G a
```

A natural transformation is a polymorphic function that is defined for all types `a`:

```
alpha :: forall a . F a -> G a
```

The `forall a` is optional in Haskell (and in fact requires turning on the language extension `ExplicitForAll`). Normally, you would write it like this:

```
alpha :: F a -> G a
```

Keep in mind that it's really a family of functions parameterized by `a`. This is another example of the terseness of the Haskell syntax. A similar construct in C++ would be slightly more verbose:

```
template<class A> G<A> alpha(F<A>);
```


10.1 Polymorphic Functions

プログラミングにおけるファンクタ（より具体的にはエンドファンクタ）の役割に ついてお話しました。ファンクタは型と型を対応させるタイプ・コンストラクタに相当します。また，関数と関数の対応付けも行い，この対応付けは高次の関数fmap（C++ではtransform，then，など）によって実装されています．自然な変換を行うには、まず、オブジェクト、ここでは型、aから始めます。 1つのファンクタFは、これを型□ に写すものです。もう1つのファンクタ G はこれを に写す。aにおける自然変換alphaの成分は□から Lu_lD44E への関数である。擬似Haskellでは

```
alpha_a :: F a -> G a
```

自然変換は、すべての型 a に対して定義される多相関数です。

```
alpha :: forall a . F a -> G a
```

Haskellではforall aはオプションである（実際には言語拡張ExplicitForAllをオンにする必要がある）。通常はこのように書きます。

```
alpha :: F a -> G a
```

これは、実際にはaでパラメータ化された関数群であることに注意してください。 これは、Haskellの構文の簡潔さを示すもう1つの例です。C++で同様の構文を使用する場合は、もう少し冗長になります。

```
template<class A> G<A> alpha(F<A>);
```

There is a more profound difference between Haskell's polymorphic functions and C++ generic functions, and it's reflected in the way these functions are implemented and type-checked. In Haskell, a polymorphic function must be defined uniformly for all types. One formula must work across all types. This is called *parametric polymorphism*.

C++, on the other hand, supports by default *ad hoc polymorphism*, which means that a template doesn't have to be well-defined for all types. Whether a template will work for a given type is decided at instantiation time, where a concrete type is substituted for the type parameter. Type checking is deferred, which unfortunately often leads to incomprehensible error messages.

In C++, there is also a mechanism for function overloading and template specialization, which allows different definitions of the same function for different types. In Haskell this functionality is provided by type classes and type families.

Haskell's parametric polymorphism has an unexpected consequence: any polymorphic function of the type:

```
alpha :: F a -> G a
```

where `F` and `G` are functors, automatically satisfies the naturality condition. Here it is in categorical notation (f is a function $f :: a \to b$):

$$Gf \circ \alpha_a = \alpha_b \circ Ff$$

In Haskell, the action of a functor `G` on a morphism `f` is implemented using `fmap`. I'll first write it in pseudo-Haskell, with explicit type annotations:

Haskellの多相関数とC++の汎用関数にはもっと大きな違いがあり、それは関数の実装方法と型チェックの方法に反映されています。Haskellでは、多相関数はすべての型に対して一様に定義されなければなりません。一つの式がすべての型にまたがって機能しなければなりません。これはパラメトリックポリモーフィズムと呼ばれる。一方，C++では，デフォルトでアドホック多相性をサポートしています．つまり，テンプレートはすべての型に対してうまく定義されている必要はないのです．テンプレートがある型に対して有効かどうかは，インスタンス生成時に決定され，具体的な型が型パラメータに代入されます．型チェックは後回しにされるため、残念ながら理解不能なエラーメッセージが出ることがよくあります。C++では、関数のオーバーロードやテンプレートの特殊化の仕組みもあり、同じ関数を異なる型に対して異なる定義ができるようになっている。Haskellでは、この機能は、型クラスと型ファミリによって提供される。Haskellのパラメトリック多相性は，型の多相性関数があれば，予想外の結果をもたらします．

```
alpha :: F a -> G a
```

ここで，FとGはファンクタであり，任意の多相関数は自動的に自然性条件を満 たす．これはカテゴリカル記法（□は関数 □ → □）で表されています。

$$Gf \circ \alpha_a = \alpha_b \circ Ff$$

Haskellでは、モルヒズムfに対するファンクタGの作用は、fmapを使って実装されています。まず、それを明示的な型注釈を付けて擬似的にHaskellで書いてみます。

```
fmap_G f . alpha_a = alpha_b . fmap_F f
```

Because of type inference, these annotations are not necessary, and the following equation holds:

```
fmap f . alpha = alpha . fmap f
```

This is still not real Haskell — function equality is not expressible in code — but it's an identity that can be used by the programmer in equational reasoning; or by the compiler, to implement optimizations.

The reason why the naturality condition is automatic in Haskell has to do with "theorems for free." Parametric polymorphism, which is used to define natural transformations in Haskell, imposes very strong limitations on the implementation — one formula for all types. These limitations translate into equational theorems about such functions. In the case of functions that transform functors, free theorems are the naturality conditions.[1]

One way of thinking about functors in Haskell that I mentioned earlier is to consider them generalized containers. We can continue this analogy and consider natural transformations to be recipes for repackaging the contents of one container into another container. We are not touching the items themselves: we don't modify them, and we don't create new ones. We are just copying (some of) them, sometimes multiple times, into a new container.

The naturality condition becomes the statement that it doesn't matter whether we modify the items first, through the application of `fmap`, and repackage later; or repackage first, and then modify the items in

[1] You may read more about free theorems in my blog "Parametricity: Money for Nothing and Theorems for Free."


```
fmap G f . alpha a = alpha b . fmap F f
```

型推論があるので、これらのアノテーションは不要で、次の式が成り立つ。

```
fmap f . alpha = alpha . fmap f
```

関数の等式はコードで表現できないが、プログラマが等式推論に使ったり、コンパイラが最適化を実装するのに使える恒等式である。Haskellで自然性条件が自動化されている理由は、"定理を無料で提供する "ことと関係がある。Haskellで自然変換を定義するために使われるパラメトリック多相性は、実装に非常に強い制限を課しており、すべての型に対して1つの式を定義している。この制限は、このような関数に関する等式定理に反映される。ファンクタを変換する関数の場合、自由定理は自然性条件となる。1 先に述べたHaskellにおけるファンクタの考え方の1つに、一般化されたコンテナという考え方がある。このアナロジーを続けて、自然変換は、ある容器の中身を別の容器に詰め替えるためのレシピと考えることができる。私たちは、アイテムそのものには触れない。アイテムを変更したり、新しいアイテムを作ったりしない。私たちはただ、（その一部を）新しいコンテナに、時には複数回、コピーしているだけなのです。自然条件は、fmapを適用してアイテムを最初に変更し、後でリパッケージする か、最初にリパッケージし、その後でアイテムを変更するかは問題ではない、と いうステートメントになる。

自由な定理について詳しくは、私のブログ「パラメトリック」を参照してください。無償の定理については、私のブログ "Parametricity: Money for Nothing and Theorems for Free" を参照してください。

the new container, with its own implementation of fmap. These two actions, repackaging and fmapping, are orthogonal. “One moves the eggs, the other boils them.”

Let’s see a few examples of natural transformations in Haskell. The first is between the list functor, and the Maybe functor. It returns the head of the list, but only if the list is non-empty:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

It’s a function polymorphic in a. It works for any type a, with no limitations, so it is an example of parametric polymorphism. Therefore it is a natural transformation between the two functors. But just to convince ourselves, let’s verify the naturality condition.

```
fmap f . safeHead = safeHead . fmap f
```

We have two cases to consider; an empty list:

```
fmap f (safeHead []) = fmap f Nothing = Nothing
```

```
safeHead (fmap f []) = safeHead [] = Nothing
```

and a non-empty list:

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


新しいコンテナには、fmapの実装があります。この2つの動作、再梱包とfmap pingは直交しています。”一方は卵を動かし、他方は卵をゆでる。” Haskellにおける自然な変換の例をいくつか見てみよう。最初の例はリストファンクターと Maybe ファンクターの間のものだ。これはリストの先頭を返しますが、リストが空でないときだけです。

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x
```

これは，aで多相性な関数です． これは，任意の型aに対して制限なく動作するので，パラメトリック多相性の例と言えます．したがって、これは2つのファンクタの間の自然な変換です。しかし，自分自身を納得させるために，自然であることの条件を検証してみましょ う．

```
fmap f . safeHead = safeHead . fmap f
```

2つのケースを考えてみよう。空リストだ。

```
fmap f (safeHead [] ) = fmap f Nothing = Nothing です。
```

```
safeHead (fmap f [] ) = safeHead [] = Nothing (何もない)
```

and a non-empty list:

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

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

I used the implementation of `fmap` for lists:

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

and for `Maybe`:

```
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

An interesting case is when one of the functors is the trivial `Const` functor. A natural transformation from or to a `Const` functor looks just like a function that's either polymorphic in its return type or in its argument type.

For instance, `length` can be thought of as a natural transformation from the list functor to the `Const Int` functor:

```
length :: [a] -> Const Int a
length [] = Const 0
length (x:xs) = Const (1 + unConst (length xs))
```

Here, `unConst` is used to peel off the `Const` constructor:

```
unConst :: Const c a -> c
unConst (Const x) = x
```

Of course, in practice `length` is defined as:

safeHead (fmap f (x : xs)) = safeHead (f x : fmap f xs) = Just (f x) ---------- fmap f (x : xs) = safeHead (x : xs) = just (f x)

リスト用のfmapの実装を使ってみました。

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

and for `Maybe`:

```
fmap f Nothing = Nothing
fmap f (Just x) = Just (f x)
```

興味深いケースは，ファンクタの1つがつまらないConstファンクタである場合 です．Constファンクタから，あるいはConstファンクタへの自然な変換は，戻り値の型 や引数の型が多相性である関数と同じように見えます．例えば，lengthはリストファンクタからConst Intファンクタへの自然な変換と考えることができます．

```
length :: [a] -> Const Int a
length [] = Const 0
length (x:xs) = Const (1 + unConst (length xs))
```

ここで、unConst は Const コンストラクタを取り除くのに使われます。

```
unConst :: Const c a -> c
unConst (Const x) = x
```

もちろん、実際には、長さは次のように定義されます。

```
length :: [a] -> Int
```

which effectively hides the fact that it's a natural transformation.

Finding a parametrically polymorphic function *from* a `Const` functor is a little harder, since it would require the creation of a value from nothing. The best we can do is:

```
scam :: Const Int a -> Maybe a
scam (Const x) = Nothing
```

Another common functor that we've seen already, and which will play an important role in the Yoneda lemma, is the `Reader` functor. I will rewrite its definition as a `newtype`:

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

It is parameterized by two types, but is (covariantly) functorial only in the second one:

```
instance Functor (Reader e) where
    fmap f (Reader g) = Reader (\x -> f (g x))
```

For every type `e`, you can define a family of natural transformations from `Reader e` to any other functor `f`. We'll see later that the members of this family are always in one to one correspondence with the elements of `f e` (the Yoneda lemma).

For instance, consider the somewhat trivial unit type `()` with one element `()`. The functor `Reader ()` takes any type `a` and maps it into a function type `() -> a`. These are just all the functions that pick a single element from the set `a`. There are as many of these as there are elements in `a`. Now let's consider natural transformations from this functor to the `Maybe` functor:


```
length :: [a] -> Int
```

というように定義され、それが自然な変換であることを効果的に隠しています。Constファンクタからパラメトリックに多相な関数を見つけるのは少し難しいです。なぜなら、何もないところから値を作る必要があるからです。私たちができる最善の方法は

```
scam :: Const Int a -> Maybe a
scam (Const x) = Nothing
```

米田レンマで重要な役割を果たす、もう一つの一般的なファンクタは Reader ファンクタです。その定義を新しい型として書き換えてみることにする.

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

これは2つの型によってパラメータ化されるが，2番目の型においてのみ （共変的に）ファンクション化される。

```
instance Functor ( Reader e) ここで
    fmap f (Reader g) = Reader (\x -> f (g x))
```

すべての型eに対して、Reader eから他の任意のファンクタfへの自然変換の族を定義することができます。この族のメンバは、f e の要素と常に1対1に対応することを後で確認する(米田レンマ)。例えば、1つの要素()を持つややつまらない単位型()を考えてみよう。ファンクタ Reader () は任意の型 a を受け取り，それを関数型 () -> a に写す． これらは，集合 a から1つの要素を選ぶすべての関数である． これらは a の要素の数と同じだけある． では，このファンクタから Maybe ファンクタへの自然変換を考えてみよう.

```
alpha :: Reader () a -> Maybe a
```

There are only two of these, `dumb` and `obvious`:

```
dumb (Reader _) = Nothing
```

and

```
obvious (Reader g) = Just (g ())
```

(The only thing you can do with `g` is to apply it to the unit value `()`.)

And, indeed, as predicted by the Yoneda lemma, these correspond to the two elements of the `Maybe ()` type, which are `Nothing` and `Just ()`. We'll come back to the Yoneda lemma later — this was just a little teaser.

10.2 Beyond Naturality

A parametrically polymorphic function between two functors (including the edge case of the `Const` functor) is always a natural transformation. Since all standard algebraic data types are functors, any polymorphic function between such types is a natural transformation.

We also have function types at our disposal, and those are functorial in their return type. We can use them to build functors (like the `Reader` functor) and define natural transformations that are higher-order functions.

However, function types are not covariant in the argument type. They are *contravariant*. Of course contravariant functors are equivalent


```
alpha :: Reader () a -> Maybe a
```

これらは2つだけで、`dumb` と `obvious` です.

```
dumb (Reader _) = Nothing
```

and

```
obvious (Reader g) = Just (g ())
```

(gでできることは、単位値()に適用することだけです）. そして、実際、米田レンマの予測通り、これらは `Maybe ()` 型の2つの要素、つまり `Nothing` と `Just ()` に対応する。米田のレンマについては、また後ほど紹介します。

10.2 Beyond Naturality

2つのファンクタ（Constファンクタのエッジケースを含む）間のパラメトリック多相関は，常に自然変換です．標準的な代数的データ型はすべてファンクタなので，そのような型間の多 相関数はすべて自然な変換です．また，関数型も自由に使うことができ，それらは戻り値の型がファンクタである．これを用いて，（Readerファンクタのような）ファンクタを構築し，高次の関数である自然変換を定義することができる．しかし、関数型は、引数の型が共変でない。反変数的である。もちろん、contravariantなファンクタは等価である。

to covariant functors from the opposite category. Polymorphic functions between two contravariant functors are still natural transformations in the categorical sense, except that they work on functors from the opposite category to Haskell types.

You might remember the example of a contravariant functor we've looked at before:

```
newtype Op r a = Op (a -> r)
```

This functor is contravariant in `a`:

```
instance Contravariant (Op r) where
    contramap f (Op g) = Op (g . f)
```

We can write a polymorphic function from, say, `Op Bool` to `Op String`:

```
predToStr (Op f) = Op (\x -> if f x then "T" else "F")
```

But since the two functors are not covariant, this is not a natural transformation in **Hask**. However, because they are both contravariant, they satisfy the "opposite" naturality condition:

```
contramap f . predToStr = predToStr . contramap f
```

Notice that the function `f` must go in the opposite direction than what you'd use with `fmap`, because of the signature of `contramap`:

```
contramap :: (b -> a) -> (Op Bool a -> Op Bool b)
```

Are there any type constructors that are not functors, whether covariant or contravariant? Here's one example:

の共変ファンクタと等価です．2つのcontravariantファンクタの間の多相関数は，Haskellの型とは反対のカテゴリからのファンクタに作用することを除けば，カテゴリ的な意味での自然な変換であることに変わりはありません．前に見たcontravariant functorの例を覚えているかもしれません。

```
newtype Op r a = Op (a -> r)
```

このファンクタは次のような点で反変数的です。

```
インスタンス Contravariant ( Op r ) ここで
    contramap f (Op g) = Op (g . f)
```

例えばOp BoolからOp Stringへの多相関数を書くことができます．

```
predToStr (Op f) = Op (\x -> if f x then "T" else "F")
```

しかし、2つのファンクタは共変ではないので、これはHaskでは自然な変換ではあり ません。しかし、両者ともcontravariantであるため、「逆の」自然条件を満たす。

```
contramap f . predToStr = predToStr . contramap f
```

contramapのシグネチャのため，関数fはfmapで使うのとは逆方向でなければならないことに注意してください．

```
contramap :: (b -> a) -> (Op Bool a -> Op Bool b)
```

共変、反変にかかわらず、ファンクタではない型構成要素はありますか？ひとつ例を挙げます。

```
a -> a
```

This is not a functor because the same type a is used both in the negative (contravariant) and positive (covariant) position. You can't implement fmap or contramap for this type. Therefore a function of the signature:

```
(a -> a) -> f a
```

where f is an arbitrary functor, cannot be a natural transformation. Interestingly, there is a generalization of natural transformations, called dinatural transformations, that deals with such cases. We'll get to them when we discuss ends.

10.3 Functor Category

Now that we have mappings between functors — natural transformations — it's only natural to ask the question whether functors form a category. And indeed they do! There is one category of functors for each pair of categories, **C** and **D**. Objects in this category are functors from **C** to **D**, and morphisms are natural transformations between those functors.

We have to define composition of two natural transformations, but that's quite easy. The components of natural transformations are morphisms, and we know how to compose morphisms.

Indeed, let's take a natural transformation α from functor F to G. Its component at object a is some morphism:

$$\alpha_a :: Fa \to Ga$$


```
a -> a
```

これは同じ型aを負位置（contravariant）でも正位置（covariant）でも使っているので、ファンクタではありません。この型に対してfmapやcontramapを実装することはできま せん。したがって，シグネチャの関数

```
(a -> a) -> f a
```

ここで、fは任意のファンクタであり、自然変換にはなり得ない。興味深いことに、このようなケースを扱う、dinatural変換と呼ばれる自然変換の一般化 があります。この一般化については末尾で説明することにしよう。

10.3 Functor Category

ファンクタ間の写像（自然変換）ができたので，ファンクタがカテゴリを形成しているかどうかを問うのは当然である．そして，実際にそうなのです．□と□の各対のカテゴリにファンクタのカテゴリが1つ存在します。このカテゴリのオブジェクトは Lu_1D402 から Lu_1D403 へのファンクタであり、モルヒズムはそれらのファンクタの間の自然変換である。2つの自然変換の合成を定義しなければならないが、それは非常に簡単である。自然変換の成分はモルヒズムであり、モルヒズムの合成の仕方はわかっている。実際、ファンクタ ŭ から□への自然変換 ŭ を考えてみよう。オブジェクト Ll_1D44E でのその成分はあるモルヒズムです。

$$\alpha_a :: Fa \to Ga$$