5

Products and Coproducts

The Ancient Greek playwright Euripides once said: "Every man is like the company he is wont to keep." We are defined by our relationships. Nowhere is this more true than in category theory. If we want to single out a particular object in a category, we can only do this by describing its pattern of relationships with other objects (and itself). These relationships are defined by morphisms.

There is a common construction in category theory called the *universal construction* for defining objects in terms of their relationships. One way of doing this is to pick a pattern, a particular shape constructed from objects and morphisms, and look for all its occurrences in the category. If it's a common enough pattern, and the category is large, chances are you'll have lots and lots of hits. The trick is to establish some kind of ranking among those hits, and pick what could be considered the best fit.

5

Products and Coproducts

古代ギリシャの劇作家、エウリピデスはこう言いました。"人は皆、付き合う人のようなものだ"私たちは、人間関係によって定義されるのです。カテゴリー理論ほど、このことが当てはまるものはない。あるカテゴリーで特定の対象を選び出したい場合，他の対象（およびそれ自身）との関係のパターンを記述することによってのみ，これを行うことができる．これらの関係はモルヒズムによって定義される。カテゴリー理論には、オブジェクトをその関係で定義するための普遍構文と呼ばれる一般的な構成がある。この方法の1つは、あるパターン（オブジェクトとモルヒズムから構成される特定の形）を選び、カテゴリーにそれがすべて出現しているかどうかを調べることである。もしそのパターンが十分に一般的で、かつカテゴリが大きければ、たくさんのパターンがヒットする可能性があります。コツは、これらのヒットの間にある種のランキングを確立し、最も適合してい ると考えられるものを選ぶことです。

This process is reminiscent of the way we do web searches. A query is like a pattern. A very general query will give you large *recall*: lots of hits. Some may be relevant, others not. To eliminate irrelevant hits, you refine your query. That increases its *precision*. Finally, the search engine will rank the hits and, hopefully, the one result that you're interested in will be at the top of the list.

5.1 Initial Object

The simplest shape is a single object. Obviously, there are as many instances of this shape as there are objects in a given category. That's a lot to choose from. We need to establish some kind of ranking and try to find the object that tops this hierarchy. The only means at our disposal are morphisms. If you think of morphisms as arrows, then it's possible that there is an overall net flow of arrows from one end of the category to another. This is true in ordered categories, for instance in partial orders. We could generalize that notion of object precedence by saying that object a is "more initial" than object b, if there is an arrow (a morphism) going from a to b. We would then define *the* initial object as one that has arrows going to all other objects. Obviously there is no guarantee that such an object exists, and that's okay. A bigger problem is that there may be too many such objects: The recall is good, but precision is lacking. The solution is to take a hint from ordered categories — they allow at most one arrow between any two objects: there is only one way of being less-than or equal-to another object. Which leads us to this definition of the initial object:

> The **initial object** is the object that has one and only one morphism going to any object in the category.

この作業は、ウェブ検索の方法を思い起こさせる。クエリというのはパターンのようなものです。非常に一般的なクエリであれば、たくさんのヒッ トが返されます。その中には、関連性のあるものもあれば、そうでないものもある。無関係なヒットを排除するために、クエリを絞り込む。そうすることで、精度が上がります。最後に、検索エンジンはヒットした検索結果を順位付けし、うまくいけば、あなたが興味を持っている結果がリストの一番上に来るでしょう。

5.1 Initial Object

最も単純な形状は、単一のオブジェクトです。明らかに、この形状のインスタンスは、与えられたカテゴリのオブジェクトの数だけ存在します。その中から選択するのは大変なことです。そこで、ある種のランキングを作り、その上位に位置するオブジェクトを見つける必要がある。そのための唯一の手段がモルヒズムである。モルヒズムを矢印と考えると、カテゴリの端から端まで全体的に矢印の純流れがある可能性がある。これは順序付きカテゴリ、例えば部分順序で言えることである。オブジェクトの優先順位の概念を一般化して、 から L1_1D44F に向かう矢印（形態素）がある場合、オブジェクト はオブジェクト L1_1D44F よりも「初期」であると言うことができるだろう。そして、他のすべてのオブジェクトに向かう矢印を持つものを、初期オブジェクトと定義することになる。もちろん、そのようなオブジェクトが存在する保証はないし、それはそれでかまわない。より大きな問題は、そのようなオブジェクトが多すぎる可能性があることである。再現率は良いのですが、精度が足りません。解決策は順序付きカテゴリからヒントを得ることである。それらは任意の2つのオブジェクトの間に最大1つの矢印を許す：他のオブジェクトより小さいか等しいかの方法は1つだけである。つまり、他のオブジェクトより小さいか等しいかのどちらかの方法しかないのである。

> 初期オブジェクトとは、カテゴリの任意のオブジェクトに向かう1つだけのモルヒズムを持つオブジェクトである。

However, even that doesn't guarantee the uniqueness of the initial object (if one exists). But it guarantees the next best thing: uniqueness *up to isomorphism*. Isomorphisms are very important in category theory, so I'll talk about them shortly. For now, let's just agree that uniqueness up to isomorphism justifies the use of "the" in the definition of the initial object.

Here are some examples: The initial object in a partially ordered set (often called a *poset*) is its least element. Some posets don't have an initial object — like the set of all integers, positive and negative, with less-than-or-equal relation for morphisms.

In the category of sets and functions, the initial object is the empty set. Remember, an empty set corresponds to the Haskell type `Void` (there is no corresponding type in C++) and the unique polymorphic function from `Void` to any other type is called `absurd`:

```
absurd :: Void -> a
```

It's this family of morphisms that makes `Void` the initial object in the category of types.

しかし、これでも初期オブジェクトの一意性は保証されない（もし存在するならば）。しかし、それは次善の策として、同型までの一意性を保証している。同型性については，カテゴリ理論で非常に重要なので，すぐに説明する．とりあえず、同型までの一意性が、初期オブジェクトの定義に「the」を使うことを正当化することだけは同意しておこう。以下はその例である。部分順序集合（しばしばポセットと呼ばれる）における初期対象は、その最小要素である。例えば、正負の全整数の集合で、モーフィズムが小なりか大なりの関係である場合、初期対象を持たないポゼットがある。集合と関数のカテゴリでは、初期オブジェクトは空集合である。空集合は Haskell の Void 型に対応し（C++ には対応する型がない）、Void 型から他の型へのユニークな多相関数は absurd と呼ばれることを覚えておいてください。

```
absurd :: Void -> a
```

このモルヒズムの一群が、Void を型のカテゴリの最初のオブジェクトにしているのです。

5.2 Terminal Object

Let's continue with the single-object pattern, but let's change the way we rank the objects. We'll say that object a is "more terminal" than object b if there is a morphism going from b to a (notice the reversal of direction). We'll be looking for an object that's more terminal than any other object in the category. Again, we will insist on uniqueness:

> The **terminal object** is the object with one and only one morphism coming to it from any object in the category.

And again, the terminal object is unique, up to isomorphism, which I will show shortly. But first let's look at some examples. In a poset, the terminal object, if it exists, is the biggest object. In the category of sets, the terminal object is a singleton. We've already talked about singletons — they correspond to the `void` type in C++ and the unit type `()` in Haskell. It's a type that has only one value — implicit in C++ and explicit in Haskell, denoted by `()`. We've also established that there is one and only one pure function from any type to the unit type:

5.2 Terminal Object

単一オブジェクトのパターンを続けながら、オブジェクトのランク付けの方法を変えてみましょう。　から　に向かう形態素がある場合、オブジェクト　はオブジェクト　より「より末端」だと言うことにします（方向が逆であることに注意してください）。私たちは、カテゴリ内の他のどのオブジェクトよりも終端的なオブジェクトを探すことになる。ここでも、一意であることを主張する。

> 終端オブジェクトとは、カテゴリのどのオブジェクトからも1つだけ来るモルヒズムを持つオブジェクトのことです。

そしてまた、末端オブジェクトは同型性までは一意であり、それはまもなく示すことにします。しかし、まず、いくつかの例を見てみよう。集合において、終端オブジェクトは、もし存在するならば、最大のオブジェクトである。集合のカテゴリでは、終端オブジェクトはシングレットである。シングルトンについてはすでに話したが、C++のvoid型やHaskellのunit型()に相当するものである。C++では暗黙的に、Haskellでは明示的に、()で示される一つの値のみを持つ型です。また、任意の型から単位型への純粋な関数が1つだけ存在することも確認しました。

```
unit :: a -> ()
unit _ = ()
```

so all the conditions for the terminal object are satisfied.

Notice that in this example the uniqueness condition is crucial, because there are other sets (actually, all of them, except for the empty set) that have incoming morphisms from every set. For instance, there is a Boolean-valued function (a predicate) defined for every type:

```
yes :: a -> Bool
yes _ = True
```

But `Bool` is not a terminal object. There is at least one more `Bool`-valued function from every type (except `Void`, for which both functions are equal to `absurd`):

```
no :: a -> Bool
no _ = False
```

Insisting on uniqueness gives us just the right precision to narrow down the definition of the terminal object to just one type.

5.3 Duality

You can't help but to notice the symmetry between the way we defined the initial object and the terminal object. The only difference between the two was the direction of morphisms. It turns out that for any category $\mathbf{C}$ we can define the *opposite category* $\mathbf{C}^{op}$ just by reversing all the arrows. The opposite category automatically satisfies all the requirements of a category, as long as we simultaneously redefine composition. If original morphisms $f :: a \to b$ and $g :: b \to c$ composed to


```
unit :: a -> ()
unit _ = ()
```

ということで、終端オブジェクトの条件はすべて満たされています。この例では、一意性の条件が重要であることに注意してください。なぜなら、すべての集合から入ってくる形態素を持つ他の集合（実際には、空集合を除くすべての集合）が存在するからです。たとえば、すべての型に対して定義されたブール値関数(述語)が存在する。

```
yes :: a -> Bool
yes _ = True
```

しかし、`Bool`は終端オブジェクトではありません。すべての型から少なくともう一つブール値関数がある（ただし、`Void`は両関数とも不条理に等しい）。

```
no :: a -> Bool
no _ = False
```

一意性にこだわることで、ターミナル・オブジェクトの定義をたった一つの型に絞り込むための、ちょうど良い精度が得られます。

5.3 Duality

初期オブジェクトと終端オブジェクトの定義の仕方が対称的であることに気づかれることでしょう。両者の違いは、モルヒズムの方向性だけである。どんなカテゴリー　に対しても、矢印をすべて逆にするだけで、反対のカテゴリー　を定義できることがわかる。反対側のカテゴリは、同時に合成を再定義する限り、自動的にカテゴリのすべての要件を満たす。元のモルヒズム ☒ と ☒ → ☒ が合成されて

$h :: a \to c$ with $h = g \circ f$, then the reversed morphisms $f^{op} :: b \to a$ and $g^{op} :: c \to b$ will compose to $h^{op} :: c \to a$ with $h^{op} = f^{op} \circ g^{op}$. And reversing the identity arrows is a (pun alert!) no-op.

Duality is a very important property of categories because it doubles the productivity of every mathematician working in category theory. For every construction you come up with, there is its opposite; and for every theorem you prove, you get one for free. The constructions in the opposite category are often prefixed with "co", so you have products and coproducts, monads and comonads, cones and cocones, limits and colimits, and so on. There are no cocomonads though, because reversing the arrows twice gets us back to the original state.

It follows then that a terminal object is the initial object in the opposite category.

5.4 Isomorphisms

As programmers, we are well aware that defining equality is a nontrivial task. What does it mean for two objects to be equal? Do they have to occupy the same location in memory (pointer equality)? Or is it enough that the values of all their components are equal? Are two complex numbers equal if one is expressed as the real and imaginary part, and the other as modulus and angle? You'd think that mathematicians would have figured out the meaning of equality, but they haven't. They have the same problem of multiple competing definitions for equality. There is the propositional equality, intensional equality, extensional equality, and equality as a path in homotopy type theory. And then there are the weaker notions of isomorphism, and even weaker of equivalence.

The intuition is that isomorphic objects look the same — they have the same shape. It means that every part of one object corresponds to

□ → □ with □ = □ □ , then the reversed morphisms □ □ □ □ → □ and □ □□ □ → □ will compose to □□ □ → □ with □□ = □ □□ □ □□ . そして、同一性の矢印を逆にすることは、（シャレにならない！）ダメである。双対性は，圏論に携わる数学者の生産性を2倍にしてくれるので，圏論の非常に重要な性質である．あなたが思いつく構成には，その反対がある．また，あなたが証明する定理には，その反対がある．反対側のカテゴリーの構成は，しばしば「co」という接頭辞を持つので，積とコプロダクト，モナドとコモナド，コーンとココーン，極限とコリミットなどがある．しかし、矢印を2回逆にすると元の状態に戻ってしまうので、コモナドは存在しない。ということは、終端オブジェクトは逆のカテゴリの初期オブジェクトであることになる。

5.4 Isomorphisms

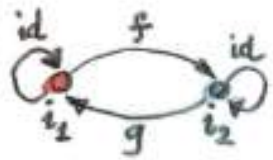
プログラマーとして、等号の定義が簡単でないことはよく承知している。2つのオブジェクトが等しいとはどういう意味だろうか？メモリ上で同じ場所を占めなければならないのか（ポインタの等価性）？それとも、すべての構成要素の値が等しいだけでよいのでしょうか？2つの複素数は、一方が実部と虚部で表され、他方がモジュラスと角度で表されるなら、等しいのだろうか？数学者なら等しいことの意味を理解していると思うだろうが、そうではない。等号の定義が複数あり、競合しているという同じ問題を抱えている。命題的等式、内包的等式、外延的等式、ホモトピー型理論におけるパスとしての等式などがある。さらに、同型性という弱い概念や、同値性というさらに弱い概念もある。直感的には、同型のものは同じに見える、つまり同じ形をしている、ということです。これは、ある物体のすべての部分が、以下のものに対応することを意味します。

some part of another object in a one-to-one mapping. As far as our instruments can tell, the two objects are a perfect copy of each other. Mathematically it means that there is a mapping from object a to object b, and there is a mapping from object b back to object a, and they are the inverse of each other. In category theory we replace mappings with morphisms. An isomorphism is an invertible morphism; or a pair of morphisms, one being the inverse of the other.

We understand the inverse in terms of composition and identity: Morphism g is the inverse of morphism f if their composition is the identity morphism. These are actually two equations because there are two ways of composing two morphisms:

```
f . g = id
g . f = id
```

When I said that the initial (terminal) object was unique up to isomorphism, I meant that any two initial (terminal) objects are isomorphic. That's actually easy to see. Let's suppose that we have two initial objects i_1 and i_2. Since i_1 is initial, there is a unique morphism f from i_1 to i_2. By the same token, since i_2 is initial, there is a unique morphism g from i_2 to i_1. What's the composition of these two morphisms?

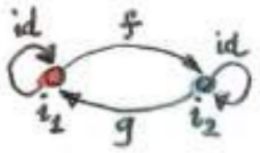


All morphisms in this diagram are unique.

ある物体のあらゆる部分が、別の物体のある部分と一対一の対応関係にあることを意味する。我々の観測機器が見る限り、この2つの物体は互いに完全にコピーされている。数学的には、物体　L1_1D44E から物体　への写像があり、物体□から物体　L1_1D44E に戻る写像があり、これらは互いに逆であることを意味します。カテゴリー理論では、写像を形態素に置き換える。同型とは、反転可能な形態素のことであり、形態素の組は、一方が他方の逆である。逆は合成と恒等式で理解される。モルヒズム　L1454　はモルヒズム　L1453　の逆で、それらの合成が同一モルヒズムである場合。2つのモルヒズムの合成には2つの方法があるので、これらは実際には2つの式である。

```
f . g = id
g . f = id
```

初期（終端）オブジェクトは同型まで一意であると言ったのは、任意の2つの初期（終端）オブジェクトが同型であるという意味です。それは、実は簡単にわかる。2つの初期物体□ 1と□ 2があるとします。□ 1 は初期なので、□ 1 から □ 2 への一意なモルヒズム　　が存在する。同じ意味で、□ 2 は初期なので、□ 2 から □ 1 への一意なモルヒズム　　が存在します。この2つのモルヒズムの合成はどうなるか？



この図のモルヒズムはすべて一意である。

The composition $g \circ f$ must be a morphism from i_1 to i_1. But i_1 is initial so there can only be one morphism going from i_1 to i_1. Since we are in a category, we know that there is an identity morphism from i_1 to i_1, and since there is room for only one, that must be it. Therefore $g \circ f$ is equal to identity. Similarly, $f \circ g$ must be equal to identity, because there can be only one morphism from i_2 back to i_2. This proves that f and g must be the inverse of each other. Therefore any two initial objects are isomorphic.

Notice that in this proof we used the uniqueness of the morphism from the initial object to itself. Without that we couldn't prove the "up to isomorphism" part. But why do we need the uniqueness of f and g? Because not only is the initial object unique up to isomorphism, it is unique up to *unique* isomorphism. In principle, there could be more than one isomorphism between two objects, but that's not the case here. This "uniqueness up to unique isomorphism" is the important property of all universal constructions.

5.5 Products

The next universal construction is that of a product. We know what a Cartesian product of two sets is: it's a set of pairs. But what's the pattern that connects the product set with its constituent sets? If we can figure that out, we'll be able to generalize it to other categories.

All we can say is that there are two functions, the projections, from the product to each of the constituents. In Haskell, these two functions are called `fst` and `snd` and they pick, respectively, the first and the second component of a pair:

合成 ☒ は ☒ 1 から ☒ 1 へのモルヒズムでなければならない。しかし、☒ 1 は初期なので、☒ 1 から ☒ 1 に向かうモルヒズムは1つしかありえない。ここはカテゴリーなので、☒ 1 から ☒ 1 への同一性モルヒズムがあることが分かっており、1つしか存在する余地がないので、それがそれでなければならないのです。したがって、 は恒等式と等しい。同様に、 は☒ 2から☒ 2に戻る形態は1つしかありえないので、同一性に等しくなければならない。このことから、 と は互いの逆でなければならないことが証明される。したがって、任意の2つの初期オブジェクトは同型である。この証明で、初期オブジェクトからそれ自身へのモルヒズムの一意性を使ったことに注目しよう。これがないと、「同型まで」の部分が証明できないからです。しかし、なぜ ☒ と ☒ の一意性が必要なのでしょうか？なぜなら、初期オブジェクトは同型性まで一意であるだけでなく、一意な同型性まで一意であるからです。原理的には、2つのオブジェクトの間には複数の同型性があり得ますが、ここではそうではありません。この「一意同型までの一意性」こそ、すべての普遍構文の重要な性質です。

5.5 Products

次の普遍構文は、積の構文です。二つの集合のデカルト積が何であるかは知っている：それはペアの集合である。しかし、積の集合とそれを構成する集合をつなぐパターンは何だろうか？それが分かれば、他のカテゴリーにも一般化できる。言えることは、積から各構成要素への投影という2つの関数があることだ。Haskellでは、この2つの関数はfstとsndと呼ばれ、それぞれペアの第1成分と第2成分を選びます。

```haskell
fst :: (a, b) -> a
fst (x, y) = x
```

```haskell
snd :: (a, b) -> b
snd (x, y) = y
```

Here, the functions are defined by pattern matching their arguments: the pattern that matches any pair is `(x, y)`, and it extracts its components into variables `x` and `y`.

These definitions can be simplified even further with the use of wildcards:

```haskell
fst (x, _) = x
snd (_, y) = y
```

In C++, we would use template functions, for instance:

```haskell
template<class A, class B> A
fst(pair<A, B> const & p) {
    return p.first;
}
```

Equipped with this seemingly very limited knowledge, let's try to define a pattern of objects and morphisms in the category of sets that will lead us to the construction of a product of two sets, a and b. This pattern consists of an object c and two morphisms p and q connecting it to a and b, respectively:

```haskell
p :: c -> a
q :: c -> b
```



```haskell
fst :: (a, b) -> a
fst (x, y) = x
```

```haskell
snd :: (a, b) -> b
snd (x, y) = y
```


ここで，関数は引数のパターンマッチングによって定義されます．任意のペアにマッチするパターンは（x，y）であり，その成分を変数 x と y に抽出します．これらの定義は，ワイルドカードを使うことでさらに簡略化することができる．

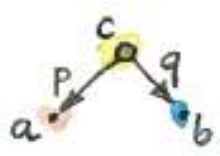

```haskell
fst (x, _) = x
snd (_, y) = y
```


C++では，例えばテンプレート関数を使うことになる．


```haskell
template<class A, class B> A
fst(pair<A, B> const & p) {
    return p.first;
}
```


このように非常に限られた知識しかないので、2つの集合　と　の積の構築につながる、集合のカテゴリにおけるオブジェクトとモルヒズムのパターンを定義してみよう。このパターンは、物体　L1_1D450と、それを□と□にそれぞれ接続する二つのモルヒズム□から構成されている。


```haskell
p :: c -> a
q :: c -> b
```



All *c*s that fit this pattern will be considered candidates for the product. There may be lots of them.

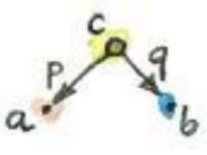
For instance, let's pick, as our constituents, two Haskell types, `Int` and `Bool`, and get a sampling of candidates for their product.

Here's one: `Int`. Can `Int` be considered a candidate for the product of `Int` and `Bool`? Yes, it can — and here are its projections:

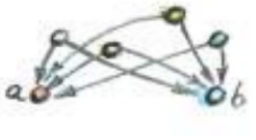
```
p :: Int -> Int
p x = x

q :: Int -> Bool
q _ = True
```

That's pretty lame, but it matches the criteria.

Here's another one: `(Int, Int, Bool)`. It's a tuple of three elements, or a triple. Here are two morphisms that make it a legitimate candidate (we are using pattern matching on triples):



このパターンに適合するすべての□が積の候補とみなされる。たくさんある場合もある。



例えば、HaskellのIntとBoolという2つの型を構成要素として選び、その積の候補のサンプリングをしてみましょう。これがその一つです。Int 。Int は Int と Bool の積の候補と見なせるでしょうか？はい、できます。その射影を示します。

```
p :: Int -> Int
p x = x

q :: Int -> Bool
q _ = True
```

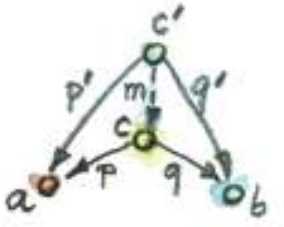
これはかなりいい加減なものですが、条件には合っています。もうひとつ、(Int, Int, Bool) というのがあります。これは3つの要素からなるタプル、つまりトリプルです。これは正当な候補となる2つの形態素です（私たちはトリプルのパターンマッチを使っています）。

```
p :: (Int, Int, Bool) -> Int
p (x, _, _) = x

q :: (Int, Int, Bool) -> Bool
q (_, _, b) = b
```

You may have noticed that while our first candidate was too small — it only covered the `Int` dimension of the product; the second was too big — it spuriously duplicated the `Int` dimension.

But we haven't explored yet the other part of the universal construction: the ranking. We want to be able to compare two instances of our pattern. We want to compare one candidate object c and its two projections p and q with another candidate object c' and its two projections p' and q'. We would like to say that c is "better" than c' if there is a morphism m from c' to c — but that's too weak. We also want its projections to be "better," or "more universal," than the projections of c'. What it means is that the projections p' and q' can be reconstructed from p and q using m:

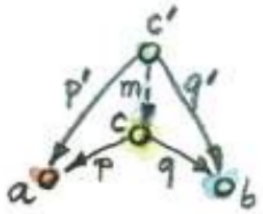
```
p' = p . m
q' = q . m
```




```
p :: (Int, Int, Bool) -> Int
p (x, _, _) = x

q :: (Int, Int, Bool) -> Bool
q (_, _, b) = b
```

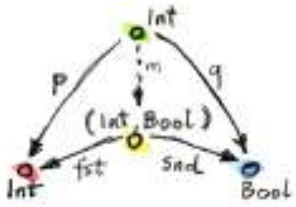

最初の候補は小さすぎて、積のInt次元しかカバーしていないことにお気づきでしょうか。2番目の候補は大きすぎて、Int次元が誤って重複しています。しかし、我々はまだ普遍的な構造の他の部分、すなわちランキングを調査していません。私たちはパターンの2つのインスタンスを比較することができます。ある候補オブジェクトŭとその2つの射影ŭと L145D を、別の候補オブジェクト L1_1D450′ とその2つの射影⊠と L145E′ と比較したいのです。もし⊠から⊠への形態素があれば、⊠は⊠より「良い」、と言いたいところだが、それでは弱すぎる。また、その投影は ⊠ の投影よりも「より良い」、つまり「より普遍的」であってほしいのです。その意味は、投影 ⊠ と L1_1D45E′ が ⊠ と⊠ を使って再構築できることである。


```
p' = p . m
q' = q . m
```

Another way of looking at these equations is that *m factorizes* p' and q'. Just pretend that these equations are in natural numbers, and the dot is multiplication: *m* is a common factor shared by p' and q'.

Just to build some intuitions, let me show you that the pair `(Int, Bool)` with the two canonical projections, `fst` and `snd` is indeed *better* than the two candidates I presented before.



The mapping `m` for the first candidate is:

```
m :: Int -> (Int, Bool)
m x = (x, True)
```

Indeed, the two projections, `p` and `q` can be reconstructed as:

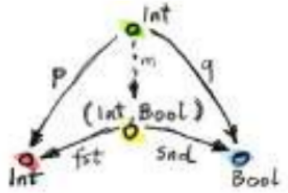
```
p x = fst (m x) = x
q x = snd (m x) = True
```

The `m` for the second example is similarly uniquely determined:

```
m (x, _, b) = (x, b)
```

We were able to show that `(Int, Bool)` is better than either of the two candidates. Let's see why the opposite is not true. Could we find some `m'` that would help us reconstruct `fst` and `snd` from `p` and `q`?

これらの方程式を別の見方をすると、 は と L1_145E′ を因数分解することになる。これらの方程式が自然数であり、ドットが乗算であると仮定すれば、 L1_145A は ′ と ′ に共通する因数である。直感を養うために、2つの正準投影、fst と snd を持つ（Int, Bool）のペアが、前に示した2つの候補よりも本当に良いことをお見せしましょう。



第一候補の写像mは

```
m :: Int -> (Int, Bool)
m x = (x, True)
```

確かに、2つの射影、pとqは次のように再構成できる。

```
p x = fst (m x) = x
q x = snd (m x) = True
```

第二の候補の写像mも同様に一意に決まる。

```
m (x, _, b) = (x, b)
```

(Int, Bool)が2つの候補のどちらよりも優れていることを示すことができた。では、なぜその逆が成り立たないのかを見てみよう。pとqからfstとsndを再構成するのに役立つ何らかのm'を見つけること はできないだろうか？

```
fst = p . m'
snd = q . m'
```

In our first example, q always returned True and we know that there are pairs whose second component is False. We can't reconstruct snd from q.

The second example is different: we retain enough information after running either p or q, but there is more than one way to factorize fst and snd. Because both p and q ignore the second component of the triple, our m' can put anything in it. We can have:

```
m' (x, b) = (x, x, b)
```

or

```
m' (x, b) = (x, 42, b)
```

and so on.

Putting it all together, given any type c with two projections p and q, there is a unique m from c to the Cartesian product (a, b) that factorizes them. In fact, it just combines p and q into a pair.

```
m :: c -> (a, b)
m x = (p x, q x)
```

That makes the Cartesian product (a, b) our best match, which means that this universal construction works in the category of sets. It picks the product of any two sets.

Now let's forget about sets and define a product of two objects in any category using the same universal construction. Such a product doesn't always exist, but when it does, it is unique up to a unique isomorphism.


```
fst = p . m'
snd = q . m'
```

最初の例では、qは常にTrueを返しており、第2成分がFalseであるペアが存在することが分かっている 。qからsndを復元することはできない。2番目の例では、pかqのどちらかを実行した後に十分な情報を保持しているが、fstとsndを因数分解する方法は1つだけではないのである。pもqもトリプルの第2成分を無視するので、我々のm'はそこに何でも入れることができる。私たちは持つことができる。

```
m' (x, b) = (x, x, b)
```

or

```
m' (x, b) = (x, 42, b)
```

といった具合です。以上をまとめると、任意の型cと2つの射影pとqがあれば、それらを因数分解するcからデカルト積（a, b）までの一意のmが存在することになる。実際には、pとqを一組にまとめるだけである。

```
m :: c -> (a, b)
m x = (p x, q x)
```

これは、デカルト積(a, b)が最もよくマッチすることを意味し、この普遍的な構成が集合のカテゴリで機能することを意味する。つまり、この普遍的な構成は集合のカテゴリで機能するのである。ここで、集合のことは忘れて、同じ普遍構文を使って、任意のカテゴリーにある2つのオブジェクトの積を定義してみよう。このような積は常に存在するわけではないが、存在するときは一意な同型性まで一意である。

> A **product** of two objects *a* and *b* is the object *c* equipped with two projections such that for any other object *c′* equipped with two projections there is a unique morphism *m* from *c′* to *c* that factorizes those projections.

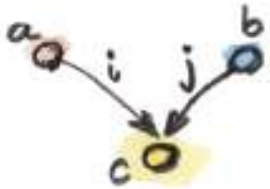
A (higher order) function that produces the factorizing function `m` from two candidates is sometimes called the *factorizer*. In our case, it would be the function:

```
factorizer :: (c -> a) -> (c -> b) -> (c -> (a, b))
factorizer p q = \x -> (p x, q x)
```

5.6 Coproduct

Like every construction in category theory, the product has a dual, which is called the coproduct. When we reverse the arrows in the product pattern, we end up with an object *c* equipped with two *injections*, `i` and `j`: morphisms from *a* and *b* to *c*.

```
i :: a -> c
j :: b -> c
```



> 2つの対象□と□の積は、2つの射影を備えた他の任意の対象□に対して、それらの射影を分解する □から□への一意の形態素が存在する、そのような射影を備えた対象の□である。

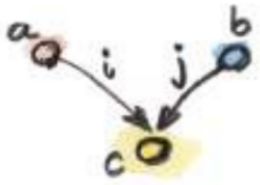
2つの候補から因数分解関数mを生成する（高次の）関数を因数分解器と呼ぶことがある。我々の場合、それは関数となる。

```
factorizer :: (c -> a) -> (c -> b) -> (c -> (a, b))
factorizer p q = \x -> (p x, q x)
```

5.6 Coproduct

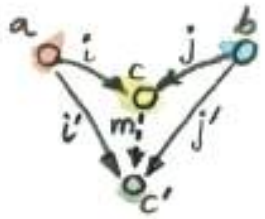
カテゴリ理論におけるすべての構成と同様に、積には双対があり、それはコプロックと呼ばれる。積のパターンの矢印を逆にすると、2つの注入、iとj：□と□から□へのモーフィズムを備えたオブジェクト□に行き着きます。

```
i :: a -> c
j :: b -> c
```



The ranking is also inverted: object c is "better" than object c' that is equipped with the injections i' and j' if there is a morphism m from c to c' that factorizes the injections:

```
i' = m . i
j' = m . j
```



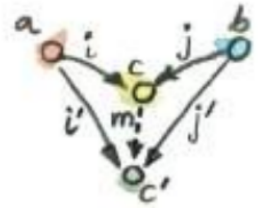
The "best" such object, one with a unique morphism connecting it to any other pattern, is called a coproduct and, if it exists, is unique up to unique isomorphism.

> A **coproduct** of two objects a and b is the object c equipped with two injections such that for any other object c' equipped with two injections there is a unique morphism m from c to c' that factorizes those injections.

In the category of sets, the coproduct is the *disjoint union* of two sets. An element of the disjoint union of a and b is either an element of a or an element of b. If the two sets overlap, the disjoint union contains two copies of the common part. You can think of an element of a disjoint union as being tagged with an identifier that specifies its origin.

順位も逆転します：オブジェクト □ は、□ から □ へのモルヒズムが存在し、その注入を分解する場合、注入 □′ と □ を備えるオブジェクト L1_1D450′ より「良い」のです。

```
i' = m . i
j' = m . j
```



このようなオブジェクトのうち、他のどのパターンにも接続する一意なモルヒズムを持つ「最良の」オブジェクトをコプロダクトと呼び、それが存在すれば、一意な同型性まで一意である。

> 2つのオブジェクト L1_1D44E と □ の共積は、2つの注入を備えた他のオブジェクト □ に対して、それらの注入を分解する □ から □′ への一意な形態素が存在するようなオブジェクト □ を指します。

集合のカテゴリにおいて、共積は2つの集合の不連続和である。 L1_1D44E と L1_1D44F の不一致和の要素は L1_1D44E の要素か □ の要素のどちらかである。2つの集合が重なっている場合、不連続和は共通部分の2つのコピーを含む。不連続和の要素は、その起源を指定する識別子でタグ付けされていると考えることができます。

For a programmer, it's easier to understand a coproduct in terms of types: it's a tagged union of two types. C++ supports unions, but they are not tagged. It means that in your program you have to somehow keep track which member of the union is valid. To create a tagged union, you have to define a tag — an enumeration — and combine it with the union. For instance, a tagged union of an `int` and a `char const *` could be implemented as:

```
struct Contact {
    enum { isPhone, isEmail } tag;
    union { int phoneNum; char const * emailAddr; };
};
```

The two injections can either be implemented as constructors or as functions. For instance, here's the first injection as a function `PhoneNum`:

```
Contact PhoneNum(int n) {
    Contact c;
    c.tag = isPhone;
    c.phoneNum = n;
    return c;
}
```

It injects an integer into `Contact`.

A tagged union is also called a *variant*, and there is a very general implementation of a variant in the boost library, `boost::variant`.

In Haskell, you can combine any data types into a tagged union by separating data constructors with a vertical bar. The `Contact` example translates into the declaration:

```
data Contact = PhoneNum Int | EmailAddr String
```


プログラマーにとって、型という観点から共産物を理解することは簡単です：それは、2つの型のタグ付き和集合です。C++はユニオンをサポートしていますが、ユニオンはタグ付きではありません。つまり、プログラムの中で、どのメンバが有効な和集合であるかを、何らかの方法で追跡しなければならないのです。タグ付き論理和を作るには、タグ（列挙）を定義して、それを論理和に組み合わ せなければならない。たとえば、intとchar const *のタグ付き結合は、次のように実装できます。

```
struct Contact {
    enum { isPhone, isEmail } tag;
    union { int phoneNum; char const * emailAddr; };
};
```

この2つのインジェクションは、コンストラクタとして実装することも、関数として実装することも可能です。例えば、最初のインジェクションは関数 PhoneNum として実装します。

```
Contact PhoneNum(int n) {
    Contact c;
    c.tag = isPhone;
    c.phoneNum = n;
    return c;
}
```

これは、Contact に整数をインジェクションします。タグ付き結合はバリアントとも呼ばれ、boost ライブラリにはバリアントの非常に一般的な実装である boost::variant が存在します。Haskellでは、データコンストラクタを縦棒で区切ることで、任意のデータ型をタグ付きユニオンに結合することができます。Contactの例では、次のような宣言に変換されます。

```
data Contact = PhoneNum Int | EmailAddr String
```

Here, PhoneNum and EmailAddr serve both as constructors (injections), and as tags for pattern matching (more about this later). For instance, this is how you would construct a contact using a phone number:

```
helpdesk :: Contact
helpdesk = PhoneNum 2222222
```

Unlike the canonical implementation of the product that is built into Haskell as the primitive pair, the canonical implementation of the coproduct is a data type called Either, which is defined in the standard Prelude as:

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

It is parameterized by two types, a and b and has two constructors: Left that takes a value of type a, and Right that takes a value of type b.

Just as we've defined the factorizer for a product, we can define one for the coproduct. Given a candidate type c and two candidate injections i and j, the factorizer for Either produces the factoring function:

```
factorizer :: (a -> c) -> (b -> c) -> Either a b -> c
factorizer i j (Left a)  = i a
factorizer i j (Right b) = j b
```

5.7 Asymmetry

We've seen two sets of dual definitions: The definition of a terminal object can be obtained from the definition of the initial object by reversing the direction of arrows; in a similar way, the definition of the coproduct can be obtained from that of the product. Yet in the category of sets

ここで、PhoneNumとEmailAddrはコンストラクタ（インジェクション）とパターンマッチのためのタグ（これについては後で説明します）の両方の役割を果たします。たとえば、電話番号を使ってコンタクトを作成する場合は、このようになります。

```
helpdesk :: Contact
helpdesk = PhoneNum 2222222
```

Haskell にプリミティブなペアとして組み込まれている積の正規の実装とは異なり、共積の正規の実装は Either というデータ型で、標準の Prelude で次のように定義されています。

```
data Either a b = 左 a | 右 b
```

これは a と b の2つの型によってパラメータ化され、2つのコンストラクタを備えています。左は a 型の値、右は b 型の値を受け取ります。積の因数分解を定義したように、共積の因数分解も定義することができます。候補型cと2つの候補注入口iとjが与えられると、Eitherの因数分解器は因数分解関数を生成する。

```
factorizer :: (a -> c) -> (b -> c) -> Either a b -> c
factorizer i j (Left a)  = i a
factorizer i j (Right b) = j b
```

5.7 Asymmetry

ここまで、2組の双対定義を見てきた。同様に、共積の定義は積の定義から得ることができる。しかし、集合のカテゴリでは

the initial object is very different from the final object, and coproduct is very different from product. We'll see later that product behaves like multiplication, with the terminal object playing the role of one; whereas coproduct behaves more like the sum, with the initial object playing the role of zero. In particular, for finite sets, the size of the product is the product of the sizes of individual sets, and the size of the coproduct is the sum of the sizes.

This shows that the category of sets is not symmetric with respect to the inversion of arrows.

Notice that while the empty set has a unique morphism to any set (the `absurd` function), it has no morphisms coming back. The singleton set has a unique morphism coming to it from any set, but it *also* has outgoing morphisms to every set (except for the empty one). As we've seen before, these outgoing morphisms from the terminal object play a very important role of picking elements of other sets (the empty set has no elements, so there's nothing to pick).

It's the relationship of the singleton set to the product that sets it apart from the coproduct. Consider using the singleton set, represented by the unit type `()`, as yet another — vastly inferior — candidate for the product pattern. Equip it with two projections `p` and `q`: functions from the singleton to each of the constituent sets. Each selects a concrete element from either set. Because the product is universal, there is also a (unique) morphism `m` from our candidate, the singleton, to the product. This morphism selects an element from the product set — it selects a concrete pair. It also factorizes the two projections:

```
p = fst . m
q = snd . m
```


は初期目的語と最終目的語が大きく異なり、副積は積と大きく異なる。後で見るが、積は乗算に似た振る舞いをし、終目的語が1の役割を果たすのに対し、共積は和に近い振る舞いをし、初目的語が0の役割を果たす。特に有限集合では、積の大きさは個々の集合の大きさの積であり、共積の大きさはそれらの大きさの和である。このことから、集合のカテゴリは、矢印の反転に関して対称ではないことがわかる。空集合は任意の集合への一意な形態素（不条理関数）を持つが、戻ってくる形態素はないことに注意。単層集合は、どの集合からも一意な形態素がやってきますが、（空集合を除く）すべての集合に向かう形態素も持っています。前に見たように、この終端オブジェクトからの出力形態素は、他の集合の要素を選ぶという非常に重要な役割を果たします（空集合は要素を持たないので、選ぶものがありません）。単体集合と積の関係で、共積とは別物になっているのです。単位型()で表される単子集合を、さらに別の（大きく劣る）積パターンの候補として使うことを考える。この集合には2つの射影pとqがある：シングルトンから各構成集合への関 数である。それぞれは、どちらかの集合から具体的な要素を選択する。積は普遍的なので、候補であるシングルトンから積への（一意な）モルヒズムmも存在する。このモルヒズムは積の集合から要素を選択する、つまり具体的なペアを選択する。それはまた、2つの投影を因数分解する。

```
p = fst . m
q = snd . m
```

When acting on the singleton value `()`, the only element of the singleton set, these two equations become:

```
p () = fst (m ())
q () = snd (m ())
```

Since `m ()` is the element of the product picked by `m`, these equations tell us that the element picked by `p` from the first set, `p ()`, is the first component of the pair picked by `m`. Similarly, `q ()` is equal to the second component. This is in total agreement with our understanding that elements of the product are pairs of elements from the constituent sets.

There is no such simple interpretation of the coproduct. We could try the singleton set as a candidate for a coproduct, in an attempt to extract the elements from it, but there we would have two injections going into it rather than two projections coming out of it. They'd tell us nothing about their sources (in fact, we've seen that they ignore the input parameter). Neither would the unique morphism from the coproduct to our singleton. The category of sets just looks very different when seen from the direction of the initial object than it does when seen from the terminal end.

This is not an intrinsic property of sets, it's a property of functions, which we use as morphisms in **Set**. Functions are, in general, asymmetric. Let me explain.

A function must be defined for every element of its domain set (in programming, we call it a *total* function), but it doesn't have to cover the whole codomain. We've seen some extreme cases of it: functions from a singleton set — functions that select just a single element in the codomain. (Actually, functions from an empty set are the real extremes.) When the size of the domain is much smaller than the size of the codomain, we often think of such functions as embedding the do-

単子集合の唯一の要素である単子値（）に作用させると、この二つの方程式は次のようになる。

```
p () = fst (m ())
q () = snd (m ())
```

m（）はmが選んだ積の要素であるから、これらの式から、最初の集合からpが選んだ要素p（）は、mが選んだ組の第1成分であることがわかる。同様に、q（）は第2成分に等しい。これは、積の要素が構成集合からの要素の組であるという理解と完全に一致している。共積にはこのような単純な解釈はない。共積の候補として、単項集合を試して、そこから要素を抽出することもできるが、その場合、そこから出る2つの投影ではなく、そこに入る2つの注入ができることになる。これでは、出所について何もわからない（実際、入力パラメータを無視することがわかった）。また、コプロダクトからシングルトンへの一意な形態素もないだろう。集合のカテゴリは、最初のオブジェクトの方向から見たときと、末端から見たときとで、まったく違って見えるだけです。これは集合の本質的な性質ではなく、□□でモルヒズムとして使う関数の性質です。一般に、関数は非対称です。説明しましょう。関数はそのドメイン集合のすべての要素に対して定義されなければなりませんが（プログラミングではこれを全関数と呼びます）、コードドメイン全体をカバーする必要はないのです。極端な例として、単一集合からの関数、つまり、コドメインの中のたった一つの要素を選択する関数を見たことがあると思います。（ドメインの大きさがコドメインの大きさよりもずっと小さい場合、このような関数は、ドーピングを埋め込むと考えることがよくあります。

main in the codomain. For instance, we can think of a function from a singleton set as embedding its single element in the codomain. I call them *embedding* functions, but mathematicians prefer to give a name to the opposite: functions that tightly fill their codomains are called *surjective* or *onto*.

The other source of asymmetry is that functions are allowed to map many elements of the domain set into one element of the codomain. They can collapse them. The extreme case are functions that map whole sets into a singleton. You've seen the polymorphic `unit` function that does just that. The collapsing can only be compounded by composition. A composition of two collapsing functions is even more collapsing than the individual functions. Mathematicians have a name for non-collapsing functions: they call them *injective* or *one-to-one*.

Of course there are some functions that are neither embedding nor collapsing. They are called *bijections* and they are truly symmetric, because they are invertible. In the category of sets, an isomorphism is the same as a bijection.

5.8 Challenges

1. Show that the terminal object is unique up to unique isomorphism.
2. What is a product of two objects in a poset? Hint: Use the universal construction.
3. What is a coproduct of two objects in a poset?
4. Implement the equivalent of Haskell `Either` as a generic type in your favorite language (other than Haskell).
5. Show that `Either` is a "better" coproduct than `int` equipped with two injections:

をコドメインに埋め込むと考えます。たとえば、単項集合からの関数は、その単項をコドメインに埋め込むと考えることができる。私はこれを埋め込み関数と呼んでいるが、数学者はその逆の名前をつけることを好む。コードメインをきっちり埋める関数は、サージェクトあるいはオンと呼ばれる。非対称性のもう一つの原因は、関数がドメイン集合の多くの要素をコドメインの一つの要素に写すことが許されることである。関数はドメイン集合の多くの要素をコードメインの1つの要素に写すことができる。極端な例としては、集合全体をシングルトンにマッピングする関数があります。多相ユニット関数はまさにそのような関数である。折りたたみ関数は、合成によってのみ複合化することができます。二つの崩れた関数の合成は、個々の関数よりもさらに崩れやすくなる。数学者は、非崩壊関数を、射影あるいは一対一と呼んでいる。もちろん，埋め込み型でもなく，畳み込み型でもない関数もある．これは両投影と呼ばれ，反転可能であるから，真に対称な関数である．集合のカテゴリでは，同型は双射と同じである．

5.8 Challenges

1．終端オブジェクトは一意的な同型性まで一意的であることを示せ。2．集合における2つの対象の積とは何か？ヒント：普遍構文を使ってください。3．ポゼットの2つの対象のコプロダクトとは？4．Haskell Eitherに相当するものを汎用型として好きな言語（Haskell以外）で 実装せよ。5．5．Eitherは、2つの注入を備えたintよりも「良い」共積であることを示せ。

```
int i(int n) { return n; }
int j(bool b) { return b ? 0: 1; }
```

Hint: Define a function

```
int m(Either const & e);
```

that factorizes `i` and `j`.

6. Continuing the previous problem: How would you argue that `int` with the two injections `i` and `j` cannot be "better" than `Either`?
7. Still continuing: What about these injections?

```
int i(int n) {
    if (n < 0) return n;
    return n + 2;
}

int j(bool b) { return b ? 0: 1; }
```

8. Come up with an inferior candidate for a coproduct of `int` and `bool` that cannot be better than `Either` because it allows multiple acceptable morphisms from it to `Either`.

5.9 Bibliography

1. The Catsters, Products and Coproducts[1] video.

[1] https://www.youtube.com/watch?v=upCSDIO9pjc


```
int i(int n) { return n; }
int j(bool b) { return b ? 0: 1; }
```

Hint: Define a function

```
int m(Either const & e);
```


であり、i と j を因数分解するものです。6. 6. 前問の続き：i と j の 2 つの注入を持つ int が Either よりも「良い」わけがないことを、どのように論証しますか？7. まだ続きます。これらの注入はどうでしょうか？


```
int i(int n) {
    if (n < 0) return n;
    return n + 2;
}

int j(bool b) { return b ? 0: 1; }
```


8. intとboolの共積で、EitherからEitherへのモルヒズムが複数許容されるため、Eitherよりも劣る候補を考えてみてください．

5.9 Bibliography

1. CatstersとProductとCoproduct 1のビデオ。

[1] https://www.youtube.com/watch?v=upCSDIO9pjc