(5)

Motivation

Consider the following C++ sketch:

```
string log = " ";

bool negate (bool x)
{
    log += "not!";
    return !x;
}
```

Hmm. This has side effects. Fix?

```
pair<bool, string> negate (bool x, string log)
{
  return make_pair(
              !x,
              log + "Not!"
          );
}
```

This is not efficient unfortunately.

Moreover, both violate SRP. Fix?

What if we modified the definition of composition?

Motivation (continued)

Our original composition is:

```
function<c(a)> compose(function<b(a)> f, function<c(b)> g)
{
    return [f, g](a x){
        auto p1 = f(x);
        auto p2 = g(p1);
        return p2;
}}
```

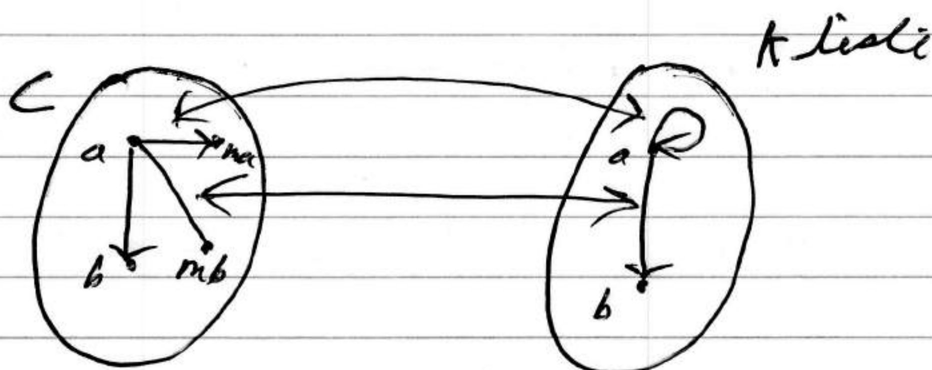
Our new composition function:

```
function<pair<c, string>(a)> compose(function<pair<b, string>(a)> f, function<pair<c, string>(b)> g
{
    return [f, g](a x){
        auto p1 = f(x);
        auto p2 = g(p1.first);
        return make_pair(
            p2.first,
            p1.second + p2.second
        );
}}
```

↑ This will work for any monoid!

This is a Kleisli Category ($a \rightarrow (b, string)$).

The Kleisli Category



$$a \longrightarrow (a, string) = m\, a.$$

So, we have identity, composition and associativity.

We will come back to this when we look at Monads later.

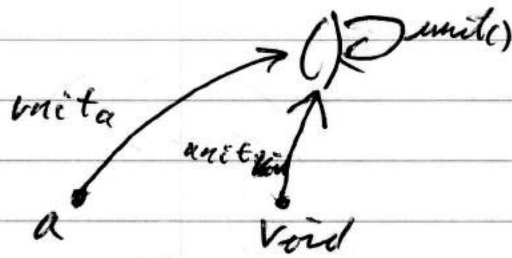
Up Next

In set theory, we define things in terms of elements.

How can we have the empty set, in category theory, if we can't talk about elements?

How can we define the Cartesian product?

Terminal Objects



For singleton sets, we always have the unit function to (). In fact, exactly one.

In category theory, we abstract the idea of a singleton set, to a:

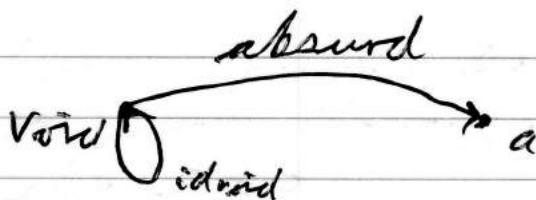
terminal object.

That is:

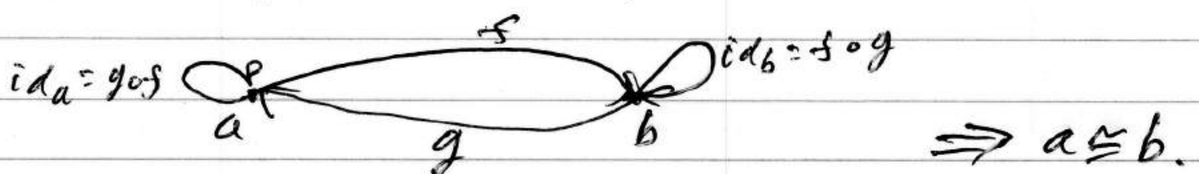
$$\forall a, \exists f :: a \to ()$$

$$\forall f :: a \to (),\ g :: a \to () \Rightarrow f = g$$

Similarly, an initial object corresponds to the empty set, where we can only have outgoing arrows, other than id.



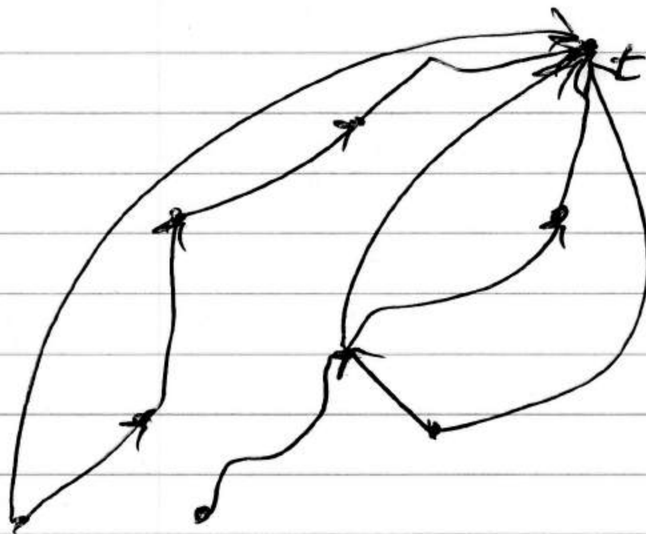
The terminal/initial objects are unique upto isomorphism:



Naming

Why are they called terminal and initial?

Think about the definitions.



There is always an arrow direct to a terminal object if connected.

When does this not exist?

Ordering on $\mathbb{Z}$. There is no longest object.

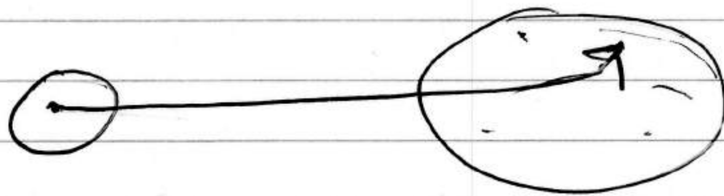
Ordering on $\mathbb{Z}_3$:

2 is a terminal, 0 is initial.

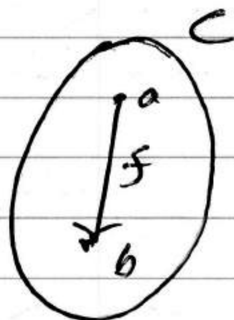
Remark

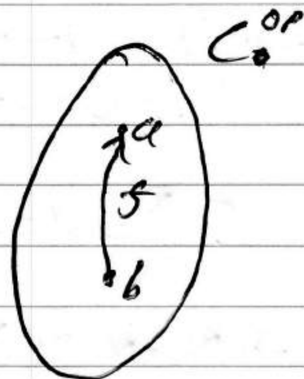
Note that we said we had to have unique incoming arrows from objects to terminals, but we said nothing about outgoing arrows.

A morphism from a terminal is like "picking an element":



Opposite Categories





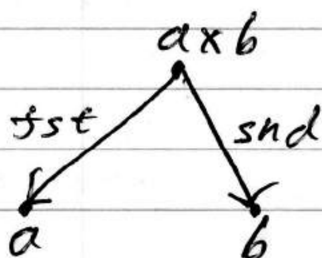
For every category C, we can construct the opposite category by reversing the arrows. Note that:

$$(g \circ f)^{op} = f^{op} \circ g^{op}.$$

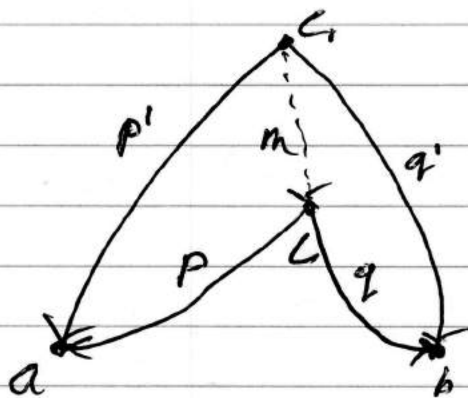
So, initial ~~categ~~ objects are terminals in the opposite category.

Products

Take two objects a and b, then we can describe their product using arrows:



Now, consider the following:



We say c is "better" than c' if a unique m exists s.t. $p \circ m = p'$, $q \circ m = q'$.

A "good" product is: (a, b)

i.e. $fst(a, _) = a$, $snd(_, b) = b$.

A "bad" product might take us to id only for example:

$$snd(_, b) = id$$

which is bad if b non-trivial.

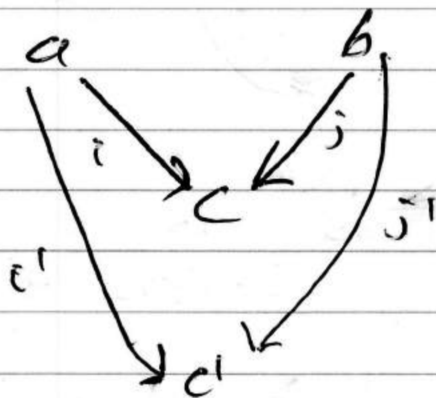
Products (continued)

In our example, c is a product, if for any other c, with p', q', then we have a unique $m: c' \to c$.

The category of sets has products for every pair of objects. This is not generally true.

Coproducts

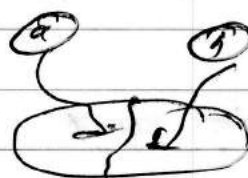
This time, we have:



injections rather than projections s.t.

$$i' = m \circ i, \quad j' = m \circ j.$$

As we know from POPL, products are pairs, and coproducts are sum types, where we have a disjoint union.



Example 1

The following types are not the same:

(a, b)   (b, a)

However, they are isomorphic, since we can swap:

swap p (snd p, fst p).

Example 2

Consider the following:

Either a Void ≅ a
Either Void a ≅ a

(a, Void) ≅ Void   (a, ()) ≅ a
(Void, a) ≅ Void   ((), a) ≅ a

(a, (b, c)) ≅ ((a, b), c)

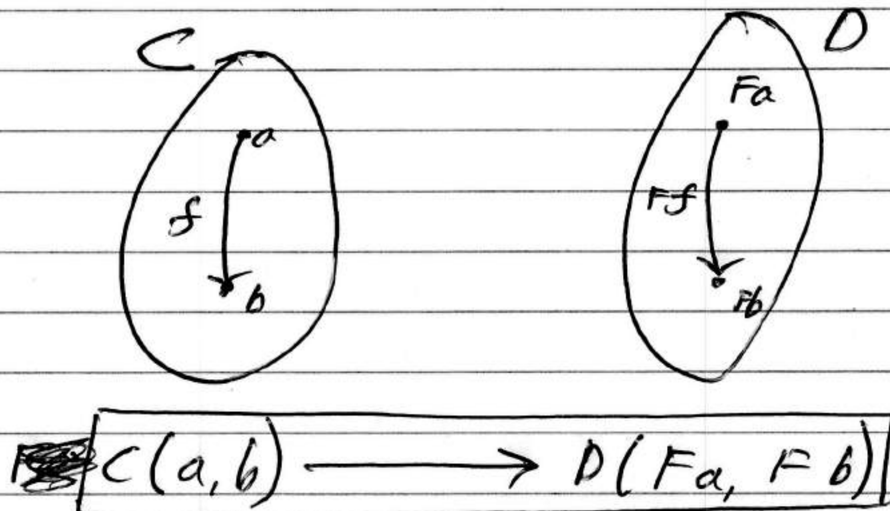
Either a (Either b c) ≅ Either (Either a b) c

(a, Either b c) ≅ Either (a, b) (a, c).

We have a semiring!

Functors

A functor is essentially a homomorphism of Categories.



for every homset.

We preserve structure, so:

$$F(g \circ f) = Fg \circ Ff$$

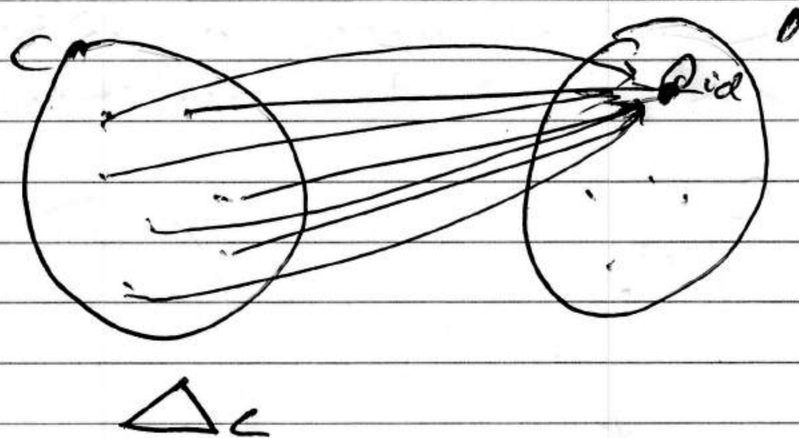
$$F(id_a) = id_{Fa}.$$

We say a functor is faithful when it is an injective mapping on the homsets. Notice we are saying nothing about the objects. They may be collapsed.

We say a functor is full if each homset map is surjective.

Fully faithful = bijective.

The Constant Functor

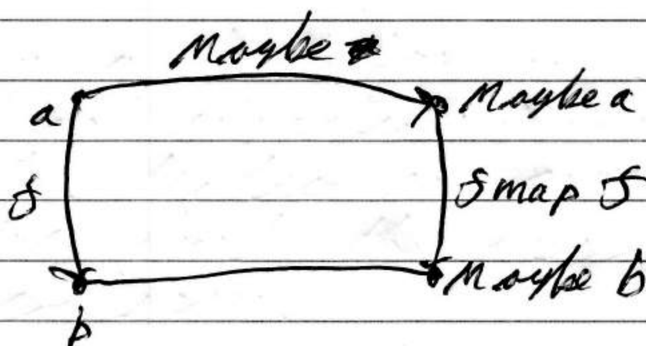


Δ_c

Endofunctors

These are common in programming languages, where the source and target are the same category, that category being types and functions.

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




```
fmap :: (a → b) → (Maybe a → Maybe b)

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

Example

```
class Eq a where
    (==) :: a -> a -> Bool
```

Recall that Haskell supports classes

We can define a Functor class:

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

Now consider a recursive definition of List:

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

Is this a functor?

```
instance Functor List where
    fmap _ Nil = Nil
    fmap f (Cons h t) = Cons (f h) (fmap f t)
```