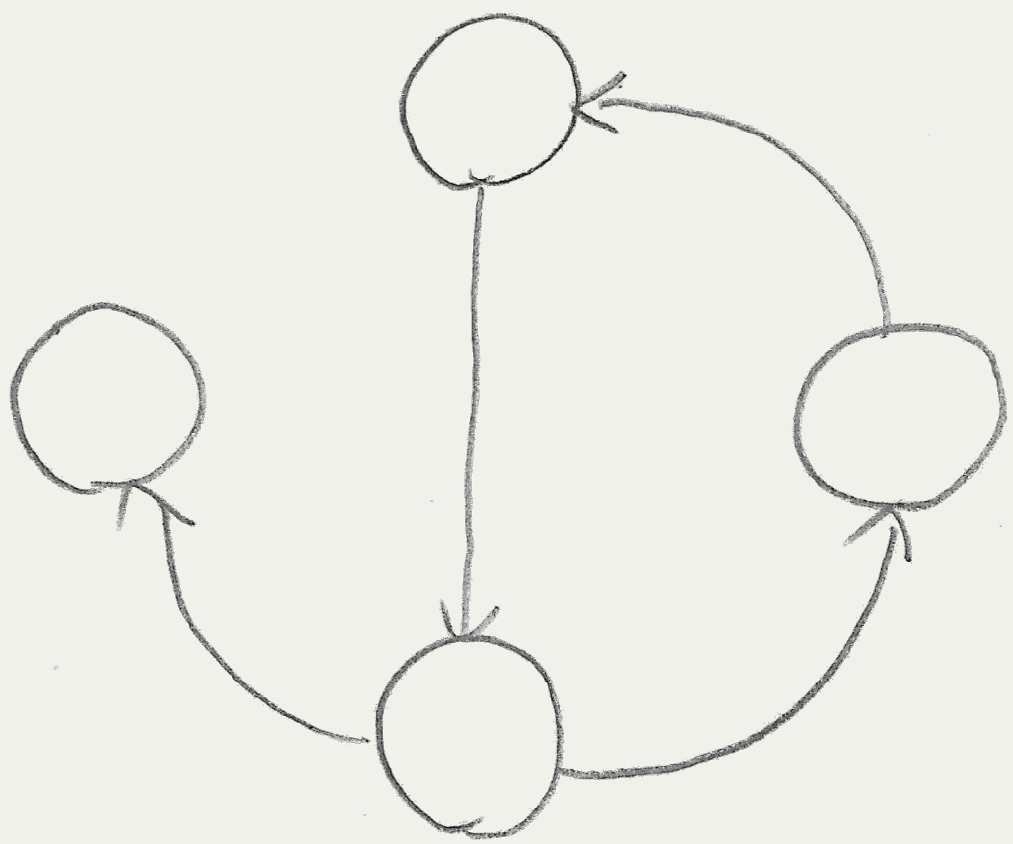
(Yet Another) Introduction to Category Theory

Rui Gonçalves

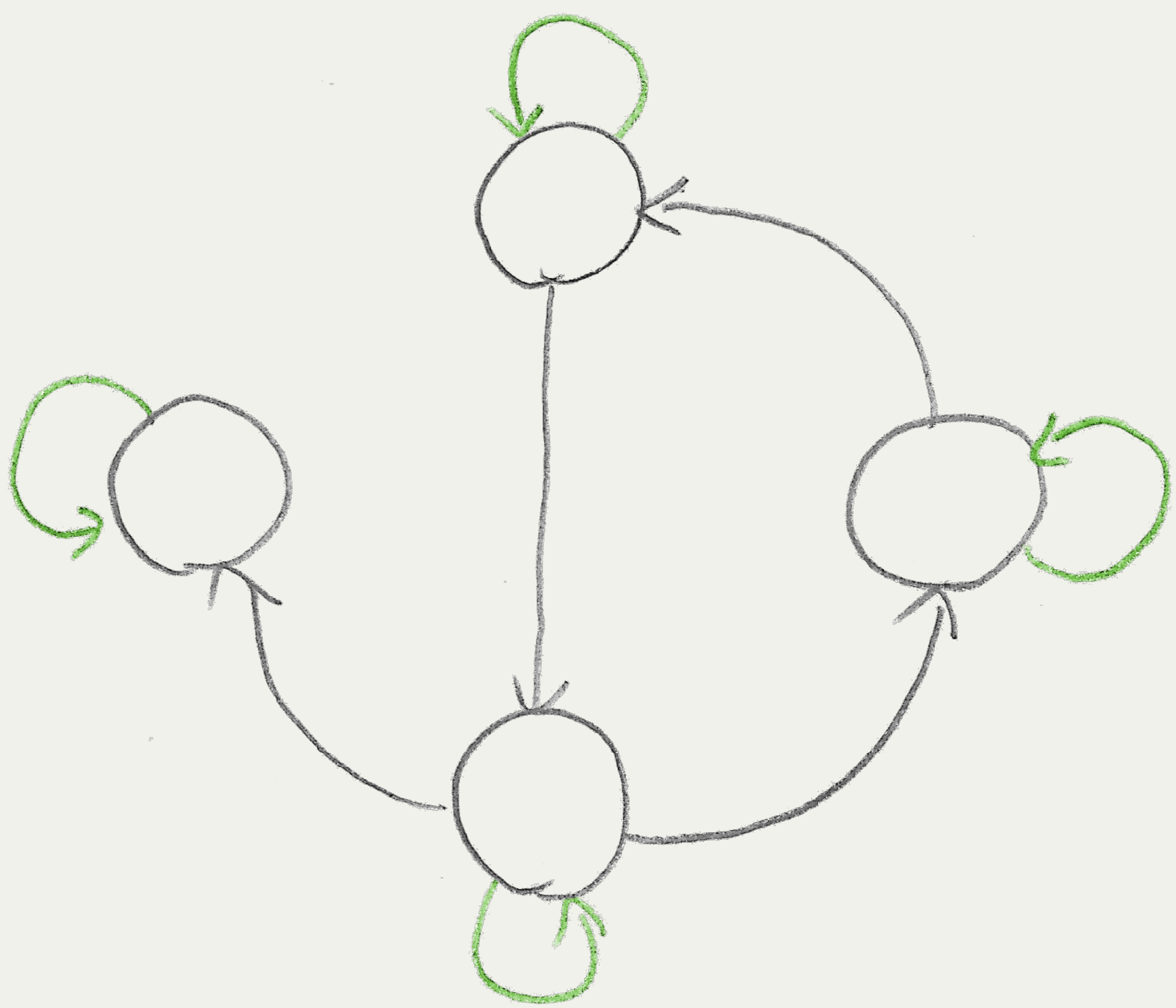
ShiftForward

June 20, 2017

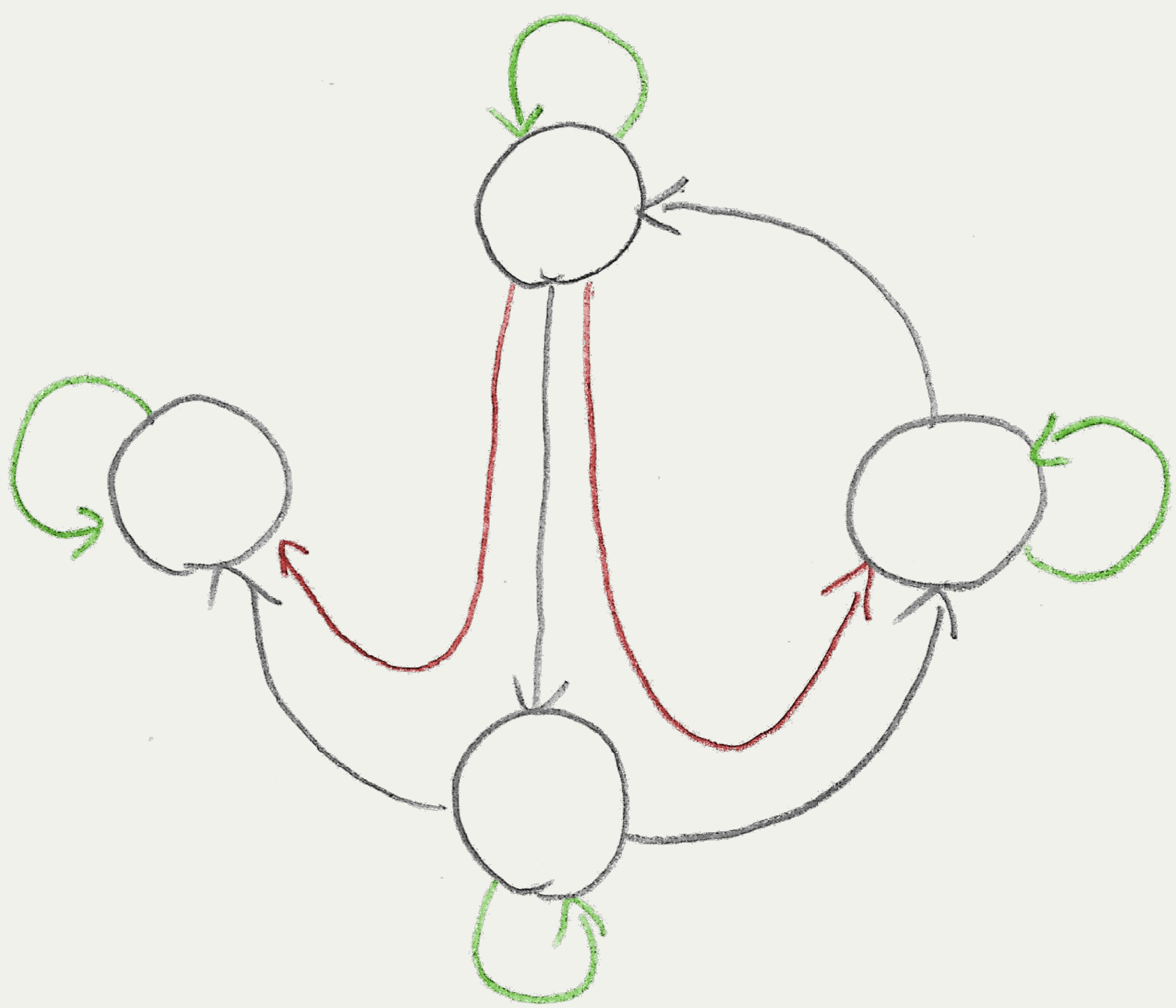
A Category



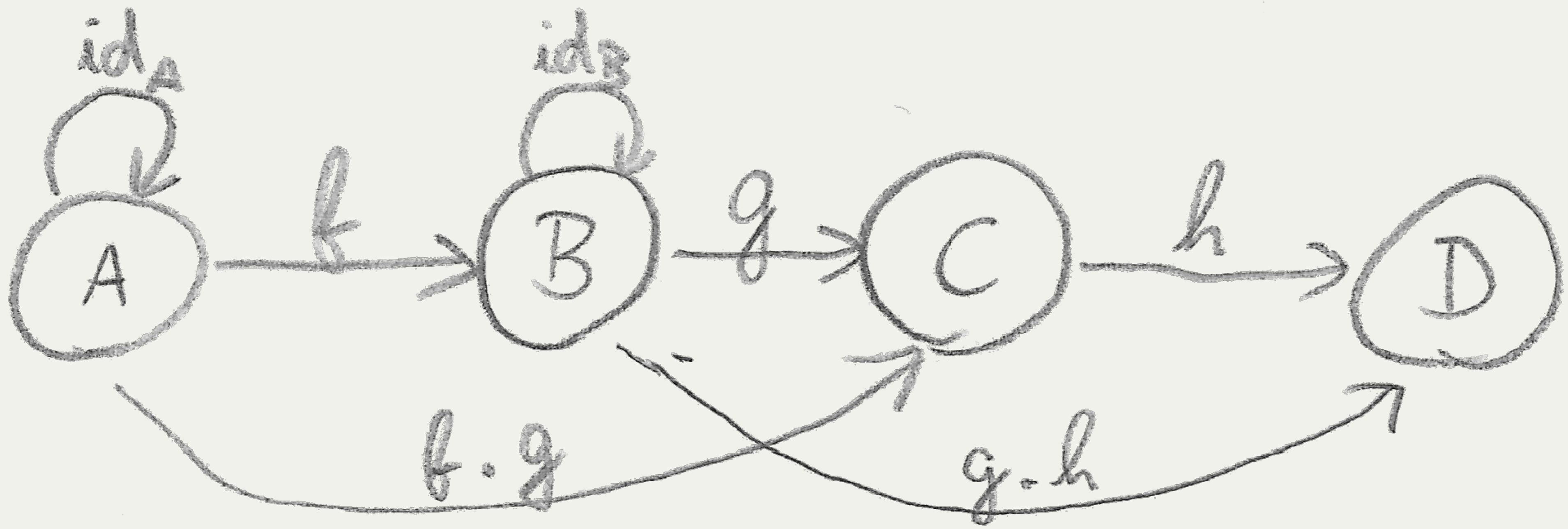
A Category



A Category



It's all about composition



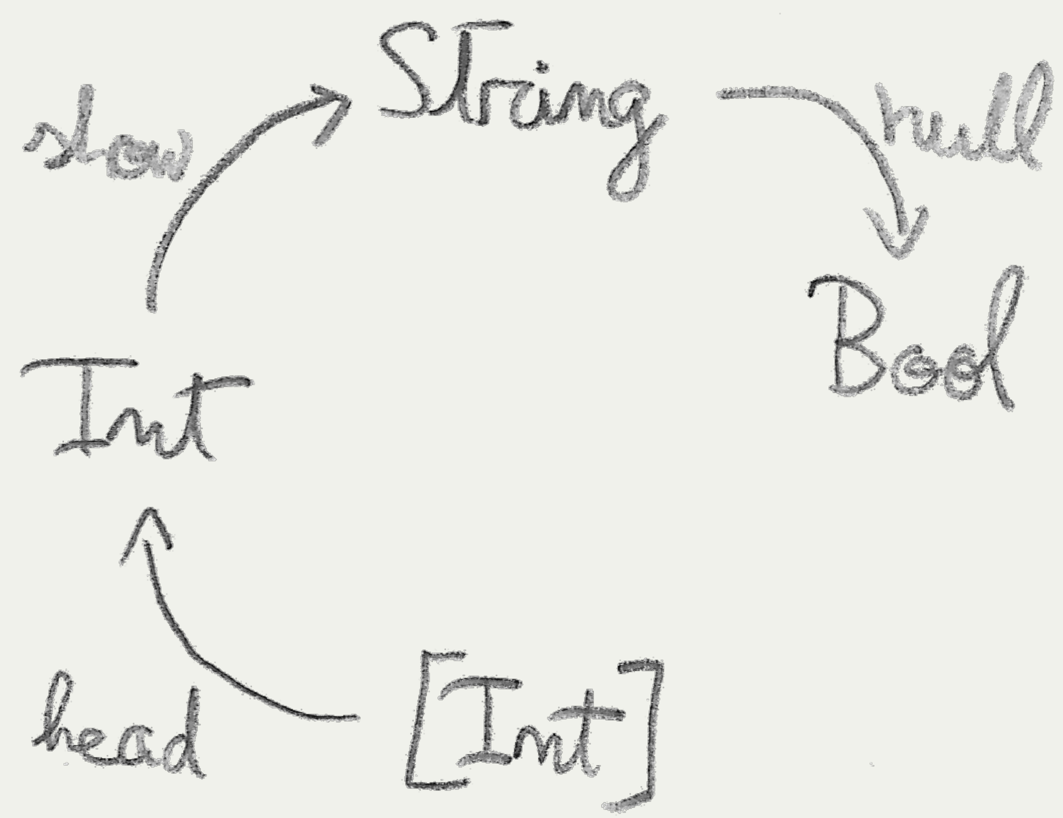
Set Category

$$
\begin{tikzcd}
 & \{0,2,4\} \arrow[dr, "g(x)=x+1"] & \\
\{0,1,2\} \arrow[ur, bend left, "f(x)=2x"] & & \{1,3,5\} \\
 & \mathbb{N} \arrow[ul, bend left, "h(x)=x \bmod 3"'] &
\end{tikzcd}
$$

Set Category

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

Hask Category



Hask Category

```haskell
id :: a -> a
id x = x

compose :: (b -> c) -> (a -> b) -> (a -> c)
compose g f x = g (f x)
-- compose g f = g . f
-- compose = (.)
```

More Categories

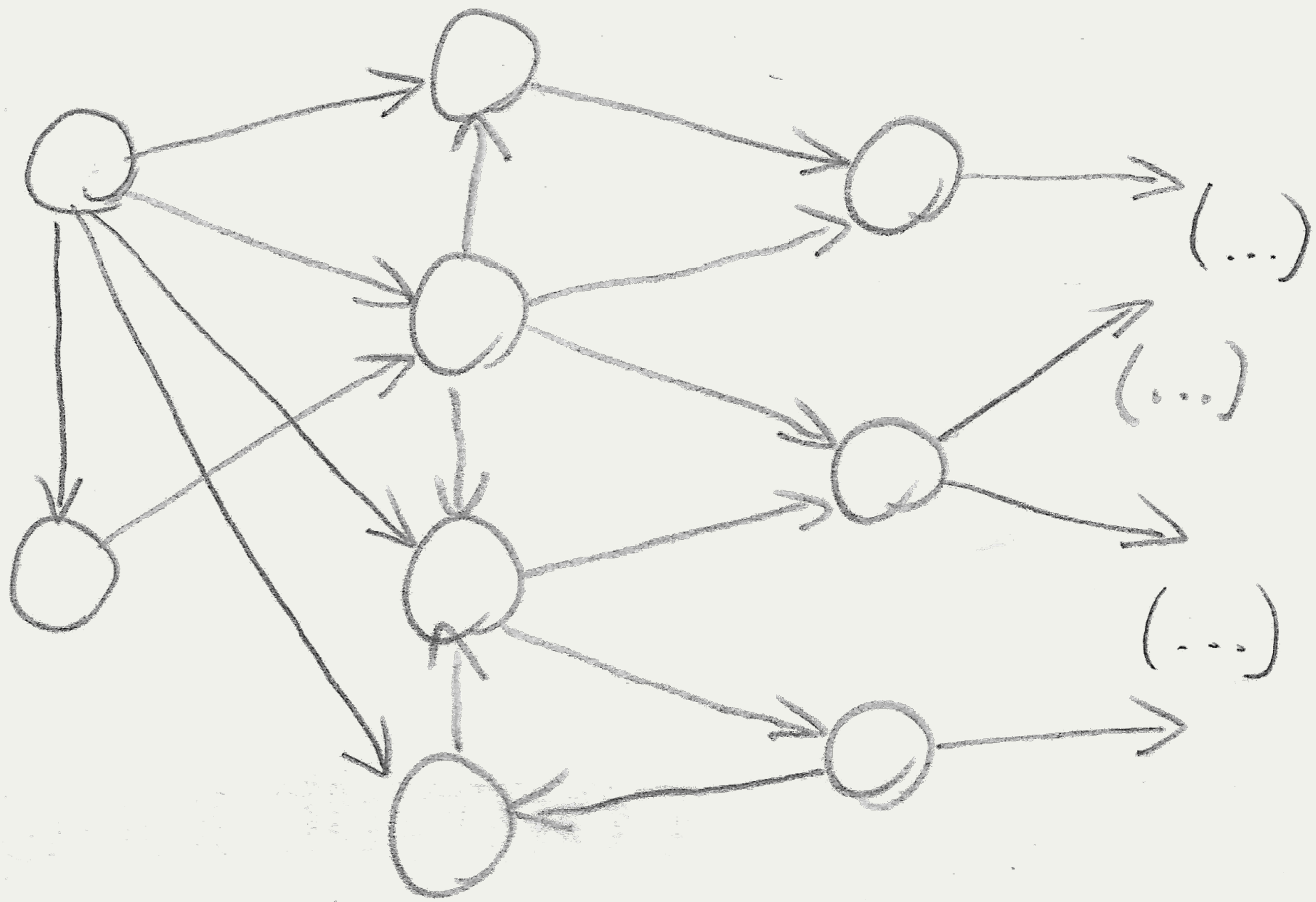
- No objects (**0**)
- Single object (**1**)
- Orders
- Vector spaces (**Vect**)
- ...

Universal Constructions

- Patterns of relationships between objects
- Allows reusing laws from one category in others
- A balance of precision and recall

1. pick a pattern and look for all its occurences
2. rank the hits and pick the best fit

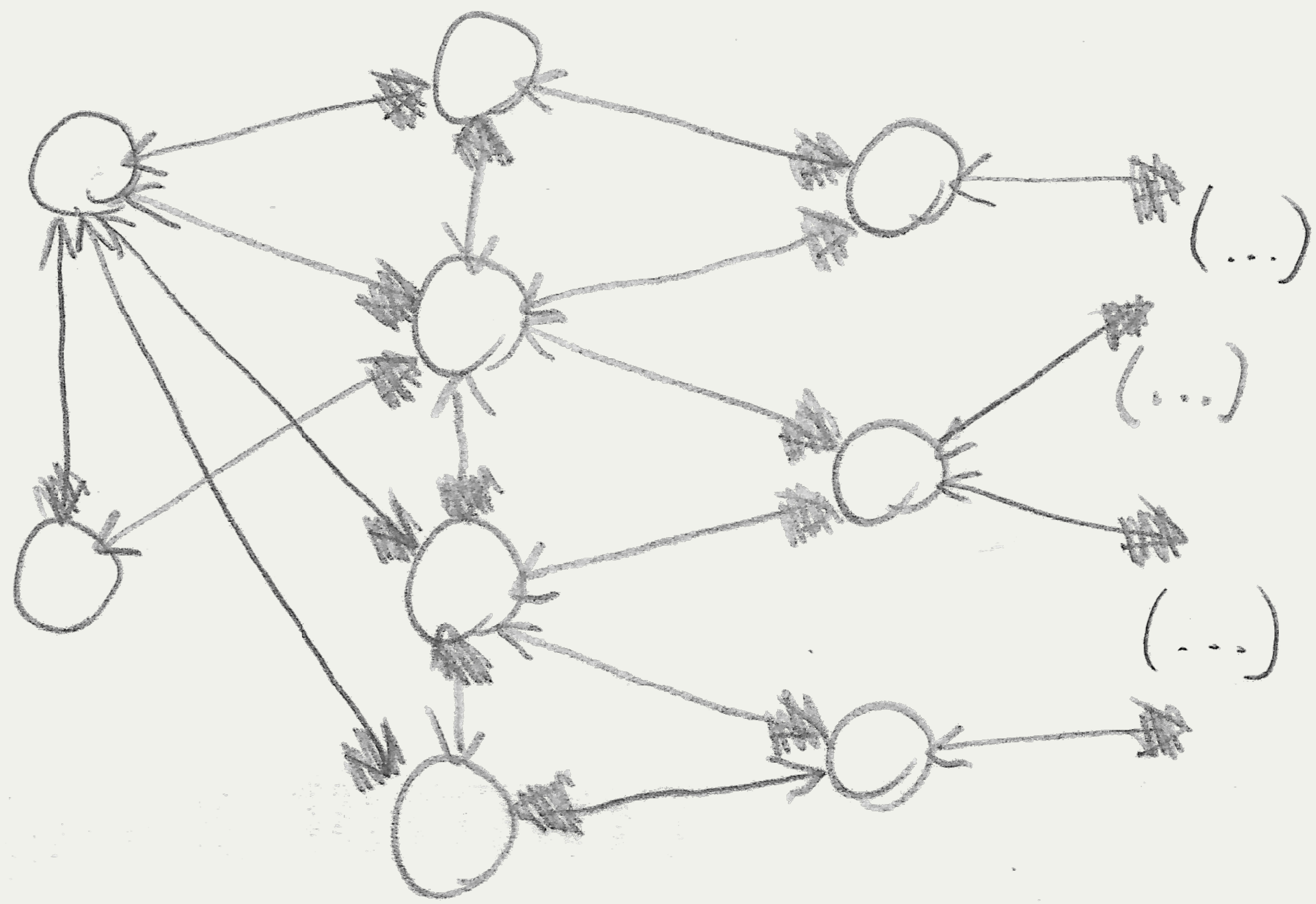
Initial Object



Initial Object

- The object is called 0
- The unique arrow $0 \to A$ is called 0_A
- The empty set is an initial object in **Set**
- Any empty type (such as `Void`) is an initial object in **Hask**

Terminal Object



Terminal Object

- The object is called 1
- The unique arrow $A \to 1$ is called 1_A
- Any singleton set is a terminal object in **Set**
- Any type with a single instance (such as `()`) is a terminal object in **Hask**

Duality

- $\mathbf{C^{op}}$ is a category obtained by keeping all objects and reversing the arrows of category **C**
- Automatically a category:
  - $id^{op} = id$
  - If $h = g \circ f$, then $h^{op} = f^{op} \circ g^{op}$

Duality

- Constructions in the opposite category are prefixed with "co" – coproducts, comonads, colimits...
- $\mathbf{C}^{\mathbf{op}}$ may be equal to **C** (not in **Set** and **Hask**)
- The terminal object is an initial object in the opposite category!

Isomorphism

- In universal constructions, the "shape" is the key
- It's enough to have a one-to-one mapping between objects
- In category theory, such a mapping is a pair of arrows, one being the *inverse* of the other

Isomorphism

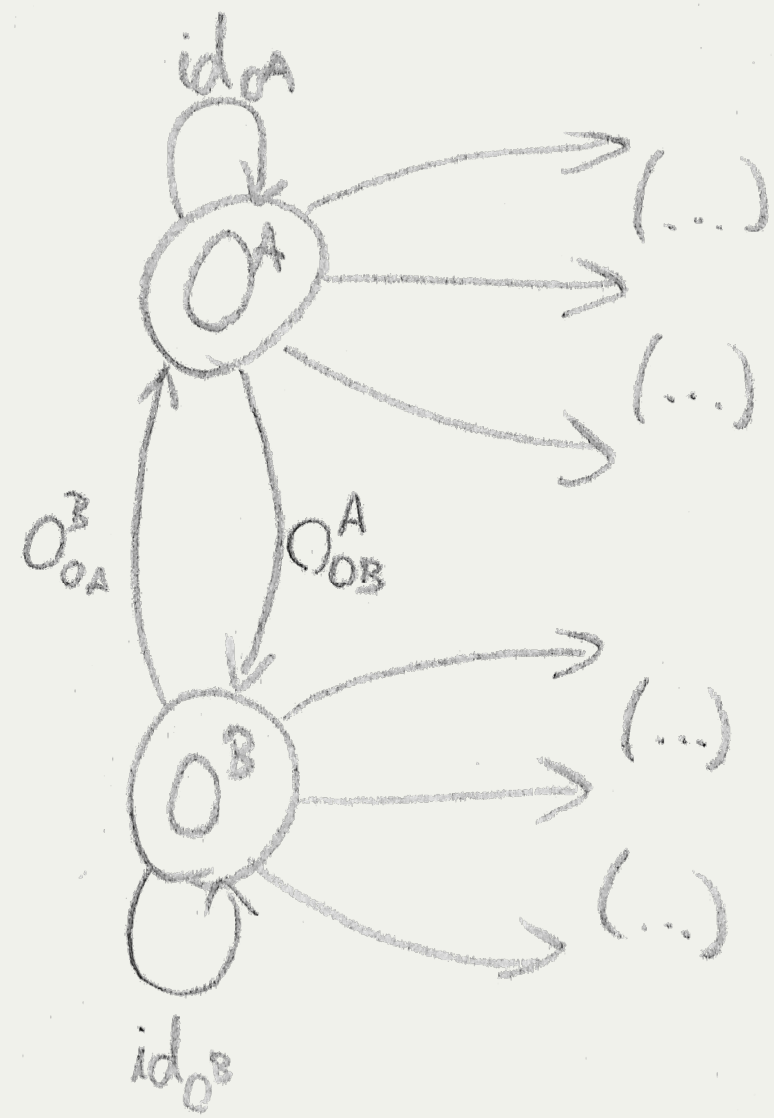
Isomorphism

- $f : A \to B$ is an isomorphism iff there is an arrow $f^{-1} : B \to A$ such that:
  - $f^{-1} \circ f = id_A$
  - $f \circ f^{-1} = id_B$

Isomorphism

- Two initial objects must be isomorphic:
  - Let 0^A and 0^B be two initial objects with arrows $0^A_C : 0^A \to C$ and $0^B_C : 0^B \to C$ for every C
  - $0^A_{0B} \circ 0^B_0 A$ is an arrow $0^A \to 0^A$
  - But there can be only one arrow from 0^A to any other object, and we already have id_A!
- No need to show the same thing for the terminal object!

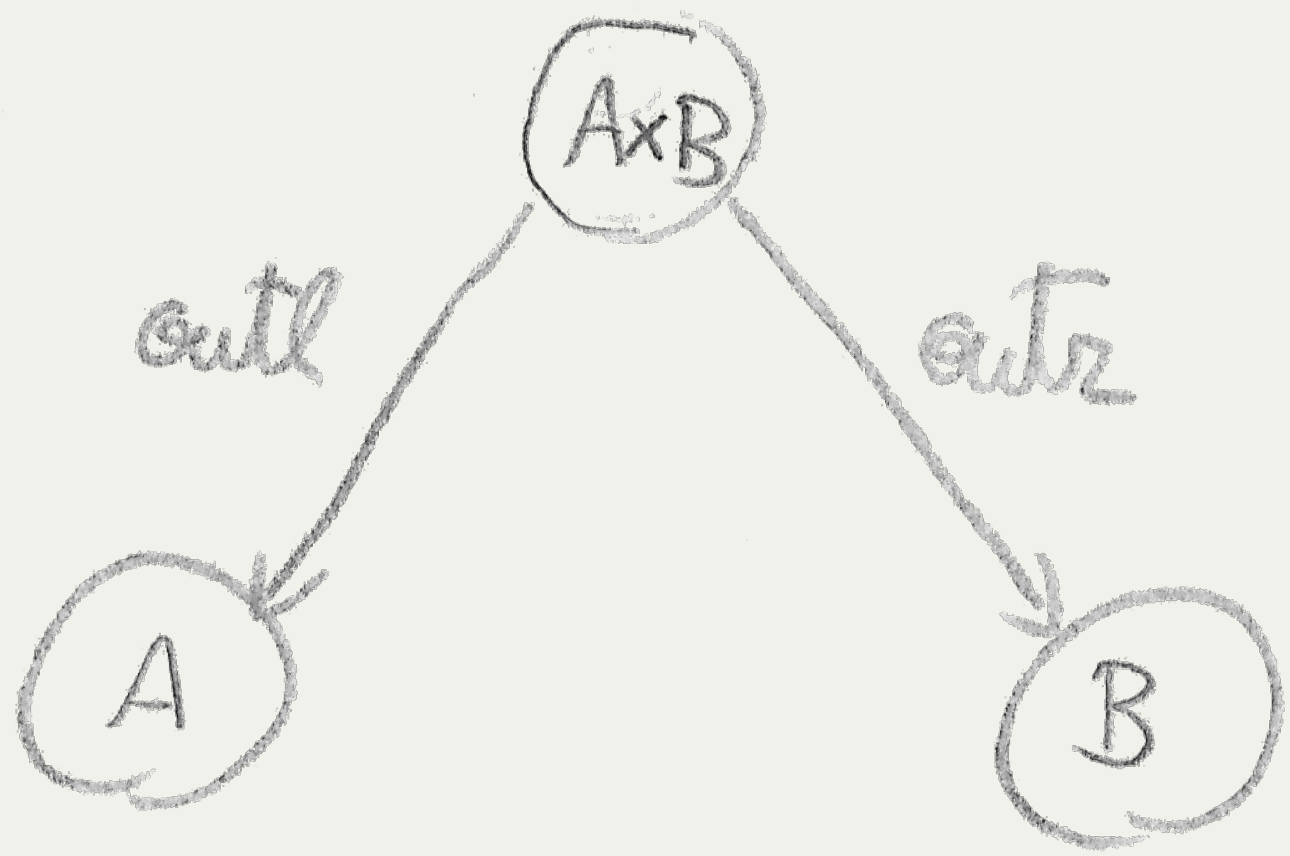
Isomorphism



Products

- Goal: generalize the notion of a cartesian product of sets
- First idea:
  - three objects: $A \times B$ is the product object of A and B
  - two arrows, $outl : A \times B \to A$ and $outr : A \times B \to B$, extracting the first and the second component

Products



Products

```
outl :: (Int, Bool) -> Int
outl (x, b) = x

outr :: (Int, Bool) -> Bool
outr (x, b) = b
```

Products

```
outl :: Int -> Int
outl x = x

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

Products

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

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

Products

- Need to rank candidates
- Idea: find an arrow between candidates that reconstructs one in function of the other
- Let P and Q be two candidates for a product with arrows $outl_P$, $outr_P$, $outl_Q$ and $outl_Q$:
  - P is "better" than Q iff there exists an **unique** arrow $m : Q \to P$ such that $outl_Q = outl_P \circ m$ and $outr_Q = outr_P \circ m$
  - Similiar to factorization in mathematics

Products

- Is `(Int, Bool)` better than `Int`?

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

- Is `Int` better than `(Int, Bool)`?

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

Products

- Is `(Int, Bool)` better than `(Int, Int, Bool)`?

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

- Is `(Int, Int, Bool)` better than `(Int, Bool)`?

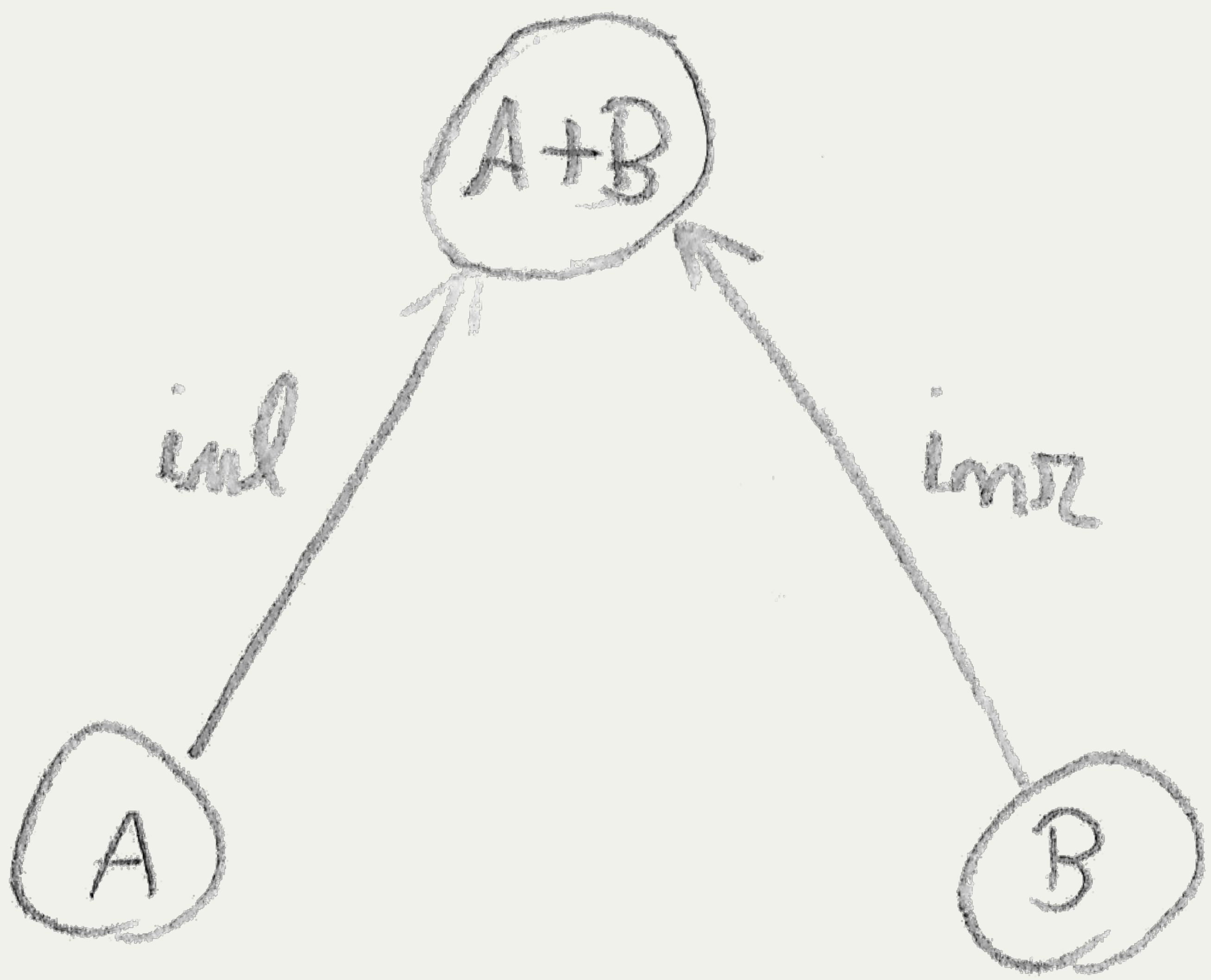
```
m :: (Int, Bool) -> (Int, Int, Bool)
-- m (x, b) = (x, ???, b)
```

Products

- With the factorization condition, every time a product exists it is unique up to a unique isomorphism
- `(a, b)` is the best match in **Hask** – A *factorizer* can show it:

```
factorizer :: (c -> a) -> (c -> b) -> (c -> (a, b))
factorizer outl outr = \x -> (outl x, outr x)
-- factorizer outl outr x = (outl x, outr x)
```

Coproducts



Coproducts

- By duality, when it exists, a coproduct is unique up to unique isomorphism
- `Either a b` is the best match in **Hask**:

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

factorizer :: (a -> c) -> (b -> c) -> (Either a b -> c)
factorizer inl inr (Left a) = inl a
factorizer inl inr (Right b) = inr b
```

Algebra of Data Types

- Most data structures in programming are built using products and coproducts
- Many properties are composable: equality, comparison, conversions...
- Treating data structures by their shape paves the way for automatic derivation

Algebra of Data Types

- These types are isomorphic in **Hask**:

```
(String, Int, Bool)         -- String * Int * Bool

((String, Int), Bool)       -- (String * Int) * Bool

(String, (Int, Bool))       -- String * (Int * Bool)

data Contact = Contact {    -- String * Int * Bool
  name :: String,
  age :: Int,
  gender :: Bool
}

(Int, Bool, String)         -- Int * Bool * String

(Int, Bool, String, ())     -- Int * Bool * String * 1
```

Algebra of Data Types

- These types are isomorphic in **Hask**:

```
Either (Either String Int) Bool     -- (String + Int) + Bool

Either String (Either Int Bool)     -- String + (Int + Bool)

data JsValue =                      -- String + Int + Bool
  JsString String |
  JsNumber Int |
  JsBoolean Bool

Either (Either Int Bool) String     -- (Int + Bool) + String

data JsValue2 =                     -- String + Int + Bool + 0
  JsString String | JsNumber Int |
  JsBoolean Bool | Void
```

Algebra of Data Types

- **Set** and **Hask** are monoidal categories:
  - With the product as binary operation and the terminal object as unit
  - With the coproduct as binary operation and the initial object as unit
- It can be shown that every category that *has* products and a terminal object is also a monoidal category (and the dual proposition for coproducts)

Algebra of Data Types

- These types are isomorphic in **Hask**:

```
-- String * (Int + Bool)
(String, Either Int Bool)

-- (String * Int) + (String * Bool)
Either (String, Int) (String, Bool)
```

Algebra of Data Types

- These types are isomorphic in **Hask**:

```
-- String * 0
(String, Void)

-- 0
Void
```

Algebra of Data Types

- **Set** and **Hask** form a *semiring* under the product and coproduct
- Not every category with product and coproduct monoids is a semiring

Algebra of Data Types

Numbers	**Types**
0	`Void`
1	`()`
$a + b$	`Either a b = Left a | Right b`
$a \times b$	`(a, b)` or `Pair a b = Pair a b`
$2 = 1 + 1$	`data Bool = True | False`
$1 + a$	`data Maybe = Nothing | Just a`

Algebra of Data Types

- Other mathematical concepts with meaning in type theory: exponentials, infinite sums, derivatives

What's More?

- Functors
- Natural transformations
- Function objects
- Everything else :)

Thank you!

- https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface
- *Algebra of Programming*, Richard Bird and Oege de Moor (1997)