(of X)

(*) It's also useful to think about all possible ordered pairs as a set isomorphic to the set of all functions $\underline{2} \to X$.

Def: Generalised elements: Let $c \in \text{Obj}\,\mathcal{C}$. Given an object b, a generalised element of b of shape c is a morphism $e : c \to b$.

(*) Note that a set is defined by its generalised elements of shape 1.

↓

This is not true for all categories, as 1 may not exist.

In general, an object in the category is determined by its generalised elements of all shapes.

(*) With morphisms also in the picture, we can transform a generalised element of x into a generalised element of y.

Let $f : x \longrightarrow y$ be a morphism.

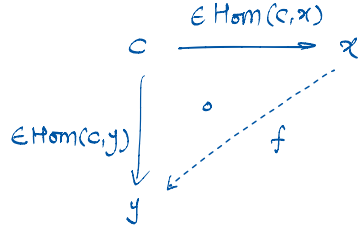
Let c be an object.

Then, we can obtain the sets of generalised elements $\text{Hom}(c,x)$ and $\text{Hom}(c,y)$, and because of f above, we also obtain a function

$$f \circ * \; : \; \text{Hom}(c,x) \longrightarrow \text{Hom}(c,y)$$

$$* \longmapsto f \circ *$$

That is, we get this commuting diagram.



This can be written as a proposition as :

Proposition Fix a shape c. Then, a morphism $f : x \to y$ induces a function from the generalised elements of x to the generalised elements of y.

Isomorphisms

Def : Isomorphism : Let $x, y \in \text{obj}(\mathcal{C})$. We say that a morphism $f : x \to y$ is an isomorphism if there exists $g : y \to x$ such that $g \circ f = \mathbb{1}_x$ and $f \circ g = \mathbb{1}_y$.

(*) Isomorphism in the category Set $\equiv$ bijection.

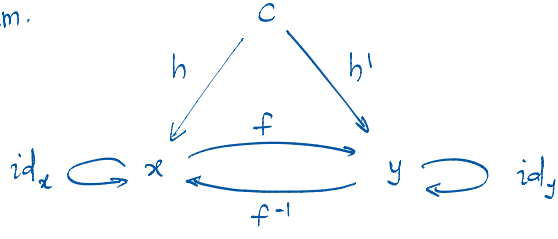
(*) For a fixed shape c and a morphism $x \to y$, the induced function as above is a bijection if f is an isomorphism.

Proposition Let $f : x \to y$ be an isomorphism. Then, for all objects c, we have a bijection

$$\text{Hom}(c, x) \underset{f^{-1} \circ *}{\overset{f \circ *}{\rightleftarrows}} \text{Hom}(c, y)$$

proof : We have the shown diagram.
f, f^{-1} as in the assumption.

$f \circ *$ sends h to $h' = f \circ h$
and $f^{-1} \circ *$ sends h' to
$h = f^{-1} \circ h'$.



□

Categories and Types Haskell

The Lambda Calculus:

* Central idea: write down mathematical functions using:
  1. Lambda abstraction
  2. Function application.

Def: **Lambda terms**: Sentences in the language of lambda calculus are called as lambda terms.

Equivalently, a valid lambda calculus expression is called as a lambda term.

Def: **Variables**: These are variable symbols, like x, y, z.

* Variables themselves are also considered as lambda terms.

* Given a lambda term A and variable x, lambda abstraction creates a new lambda term, $\lambda x.A$

"Any instance of x in A should be treated as a variable"

In a way, $\lambda x.A$ is a function with x as the input.

* Let A, B be two lambda terms. A function application creates a new lambda term AB.

  ⇓

  if A is a function in x, then we replace all instances of x in A by B.

"Composition"

And you also add parentheses

↳ more like the input/output relation.

(*) The two lambda terms are the same if one can be turned into the other by function composition.

eg:- $(\lambda x.\, xx)(zy) = (zy)(zy)$.

Breaking down the syntax:

$(\lambda x.\, xx)(zy) = (zy)(zy)$.

- $(\lambda$: we're working in λ-calculus
- x : what's the variable?
- xx : what to do with the var?
- (zy) : the input?
- $(zy)(zy)$: the output.

Above is equivalent to saying:

$f(x) := xx$

$f(zy) = (zy)(zy)$

eg:- $(\lambda x.\, (xy)x)(\lambda z.\, z) = ((\lambda z.\, z)\, y)(\lambda z.\, z)$

$= y\,(\lambda z.\, z)$

eg:- $\lambda x.\, x$ — "identity"

↳ change of var relation.

(*) Two lambda terms are considered the same if one can be obtained by the other just by changing the variable names.

$\lambda x.\, x = \lambda y.\, y$

(*) Church booleans: We want to model conditional logic in λ.

"if t then A else B"

$t = \text{prop}$.

Then, define:

$$\begin{aligned} \text{TRUE} &= \lambda x.(\lambda y.x) \\ \text{FALSE} &= \lambda x.(\lambda y.y) \\ \text{COND} &= \lambda x.x \end{aligned}$$

Then, "if t then A else B" $\equiv$ COND ((tA) B)

where t evaluates to TRUE or FALSE.

* The Y-combinator: Its a lambda term whose reduction does not terminate. Evaluation by substitution increases its length.

$$Y = \lambda f.\left((\lambda x.f(xx))(\lambda x.f(xx))\right)$$

Proposition $Yg = g(Yg)$. Thus, $Yg = g(g(\cdots(g(Yg)))$.

* The Y-combinator is used to express recursion in λ-calc.

Fun fact λ-calculus is Turing complete.

* Also, there is no restriction on what you can and cannot compose in λ-calculus.

For example, ouroboros = `\x -> x x` is allowed in λ-calc, but not languages like Haskell.

↳ x is fed to itself as an argument.

↓

Hard to reason!

Types

Haskell uses simply typed λ-calc.

* You can compose two functions only when the output type of the first matches the input type of the other.

* Integer – Arbitrary precision integers $= \mathbb{Z}$

  Int – Fixed size companion $= \mathbb{Z}/n\mathbb{Z}$ for suitable n.

  eg:- 64-bit implementation : -2^{63} to $2^{63}-1$.

  ↳ Usually preferred over Integers as they're more efficient.
  (Except when overflow is an issue)

Other types: String, Bool, Char

Def: **Polymorphic function**: Functions for which the definition works for inputs of arbitrary type.

eg:- `id :: forall a. a -> a` — the identity function.

`::` separator. Not composition.

* Types v/s Sets:

  Sets have elements but statement have types. Statements do not belong anywhere. they have types much like an "attribute"

* While declaring functions between types, much like morphisms, we declare its type signature, and then the implementation.

```
Square :: Integer -> Integer.        square x = x^2
```

(*) For all practical purposes, and only for practical purposes, you can denote a Haskell function as a mathematical function.

↳ This can however break down, for say,

$g :: Int \to Int$ by $g\ x = x + 1$.

Def: Infix, Outfix operators:

Infix : Written in between the arguments. $5 + 4$

Outfix : Written outside the arguments. $(+)\ 5\ 4$

(*) Composition of functions is itself a function.

$$(\cdot) :: (b \to c) \to (a \to b) \to a \to c$$

eg:- $f : a \to b$. $g : b \to c$. $g \cdot f : a \to c$

$$\underbrace{(b \to c)}_{g} \to \underbrace{(a \to b)}_{f} \to \underbrace{a}_{\text{input type}} \to \underbrace{c}_{\text{output type}}$$

(*) $\to$ associates to the right.

(*) $(g \cdot f)(x) \equiv (\cdot)\ g\ f = \backslash x \to g\ (f\ x)$