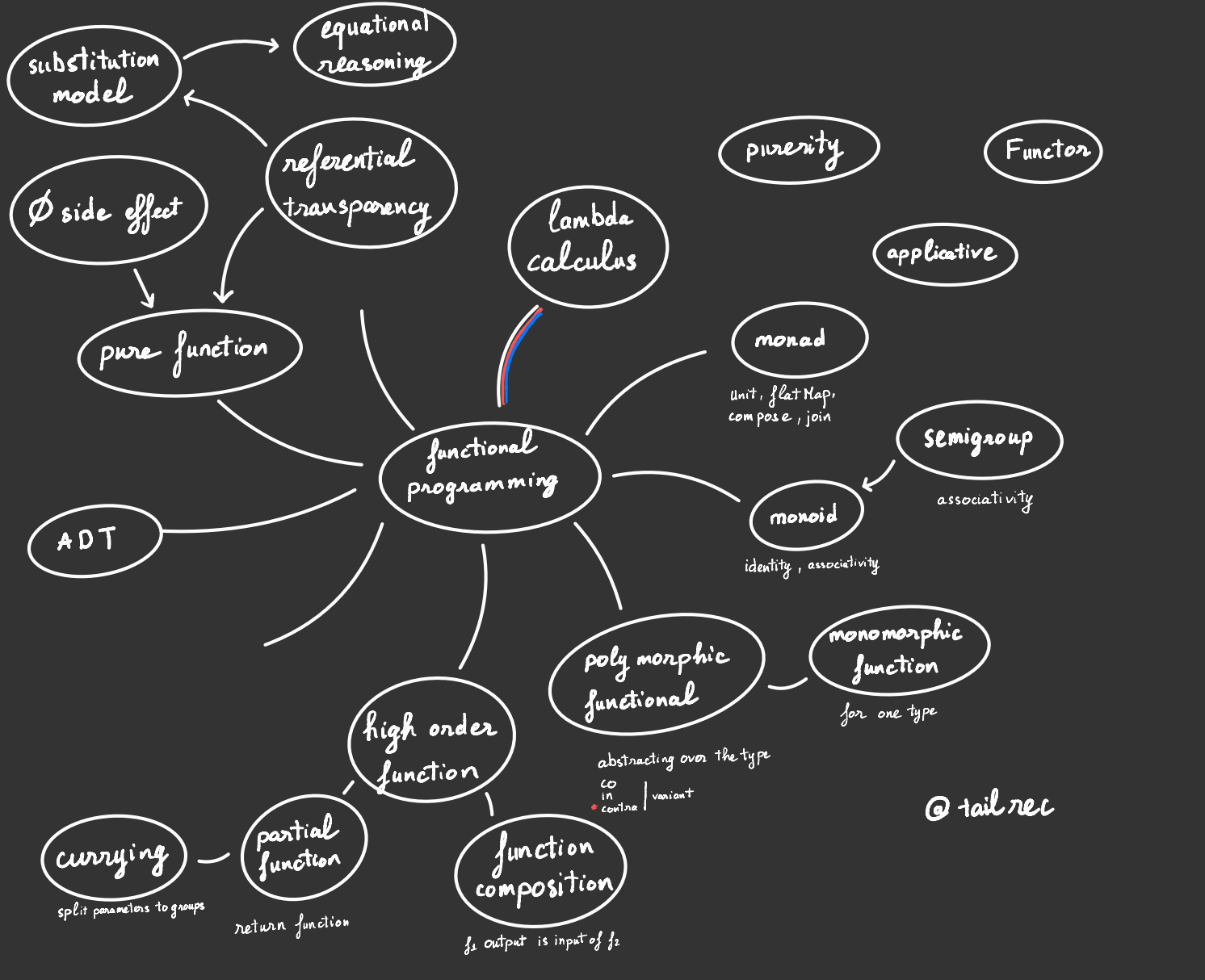
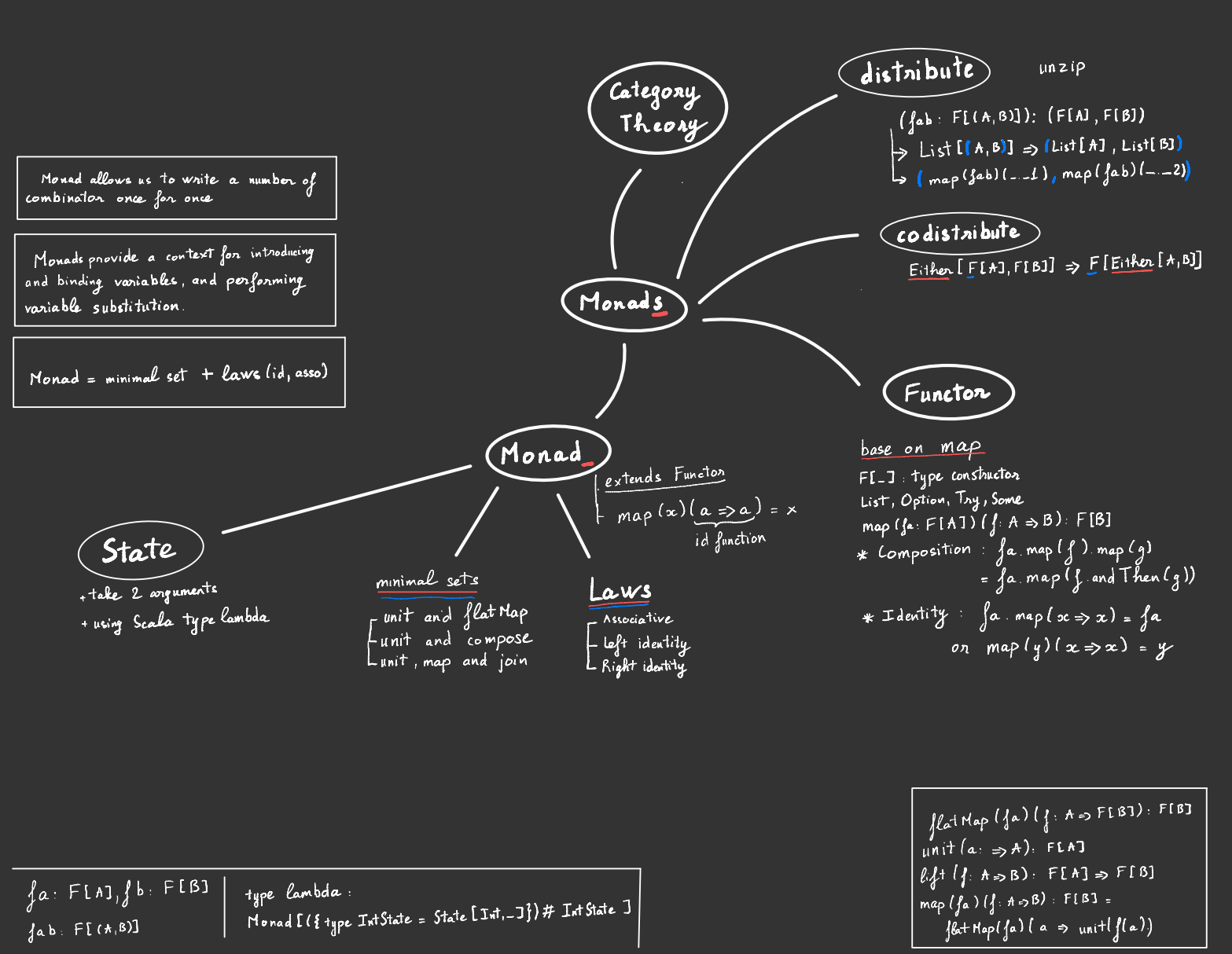
My mindmap

for

functional programming

in Scal


substitution model
equational reasoning
referential transparency
Ø side effect
pure function
lambda calculus
purerity
Functor
applicative
monad
unit, flatMap, compose, join
functional programming
semigroup
associativity
monoid
identity, associativity
ADT
poly morphic functional
monomorphic function
for one type
abstracting over the type
co
in
contra
variant
high order function
currying
split parameters to groups
partial function
return function
function composition
f1 output is input of f2
@tailrec


Category Theory
Monads
Monad
distribute
unzip
(fab: F[(A,B)]): (F[A], F[B])
List[(A,B)] ⇒ (List[A], List[B])
(map(fab)(_._1), map(fab)(_._2))
codistribute
Either[F[A], F[B]] ⇒ F[Either[A,B]]
Functor
base on map
F[_]: type constructor
List, Option, Try, Some
map(fa: F[A])(f: A ⇒ B): F[B]
* Composition: fa.map(f).map(g) = fa.map(f.andThen(g))
* Identity: fa.map(x ⇒ x) = fa
or map(y)(x ⇒ x) = y
Monad allows us to write a number of combinator once for once
Monads provide a context for introducing and binding variables, and performing variable substitution.
Monad = minimal set + laws (id, asso)
extends Functor
map(x)(a ⇒ a) = x
id function
State
+ take 2 arguments
+ using Scala type lambda
minimal sets
unit and flatMap
unit and compose
unit, map and join
Laws
Associative
Left identity
Right identity
fa: F[A], fb: F[B]
fab: F[(A,B)]
type lambda:
Monad[({type IntState = State[Int,_]})#IntState]
flatMap(fa)(f: A ⇒ F[B]): F[B]
unit(a: ⇒A): F[A]
lift(f: A ⇒ B): F[A] ⇒ F[B]
map(fa)(f: A ⇒ B): F[B] =
flatMap(fa)(a ⇒ unit(f(a)))

Semigroup

associativity
(combine or operation)

+ identity
(zero or empty)

Monoid

foldable
parallel
monoid homomorphic
compose
fuse traversals

Foldable

Traverse

Identity

Validated

Eval

Todo:

map(fa)(A ⇒ B): fb
↓
lift, fproduct, compose

Functor

+ ap(F[A ⇒ B])(fa): fb

Apply

+ pure(A): fa

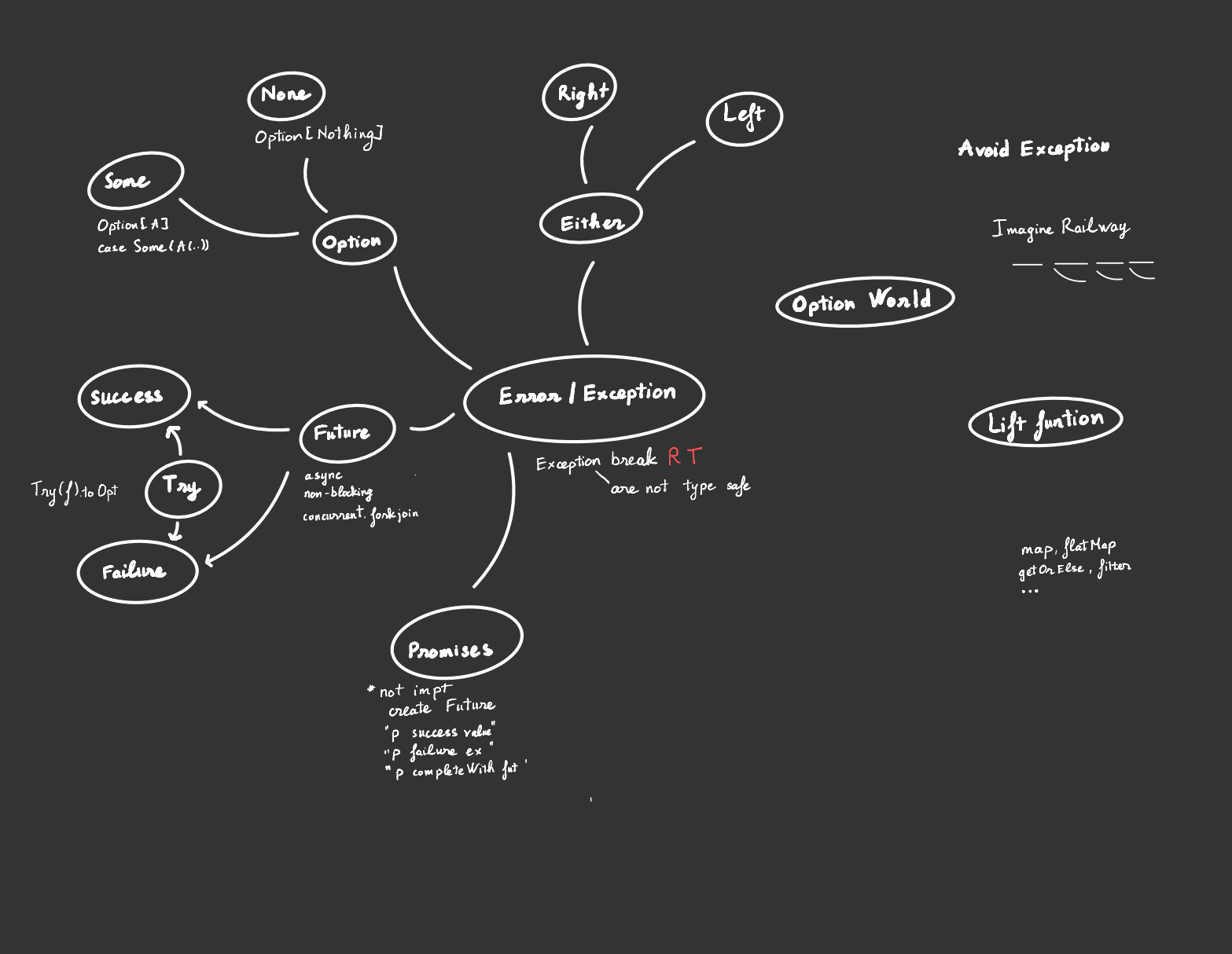
Applicative

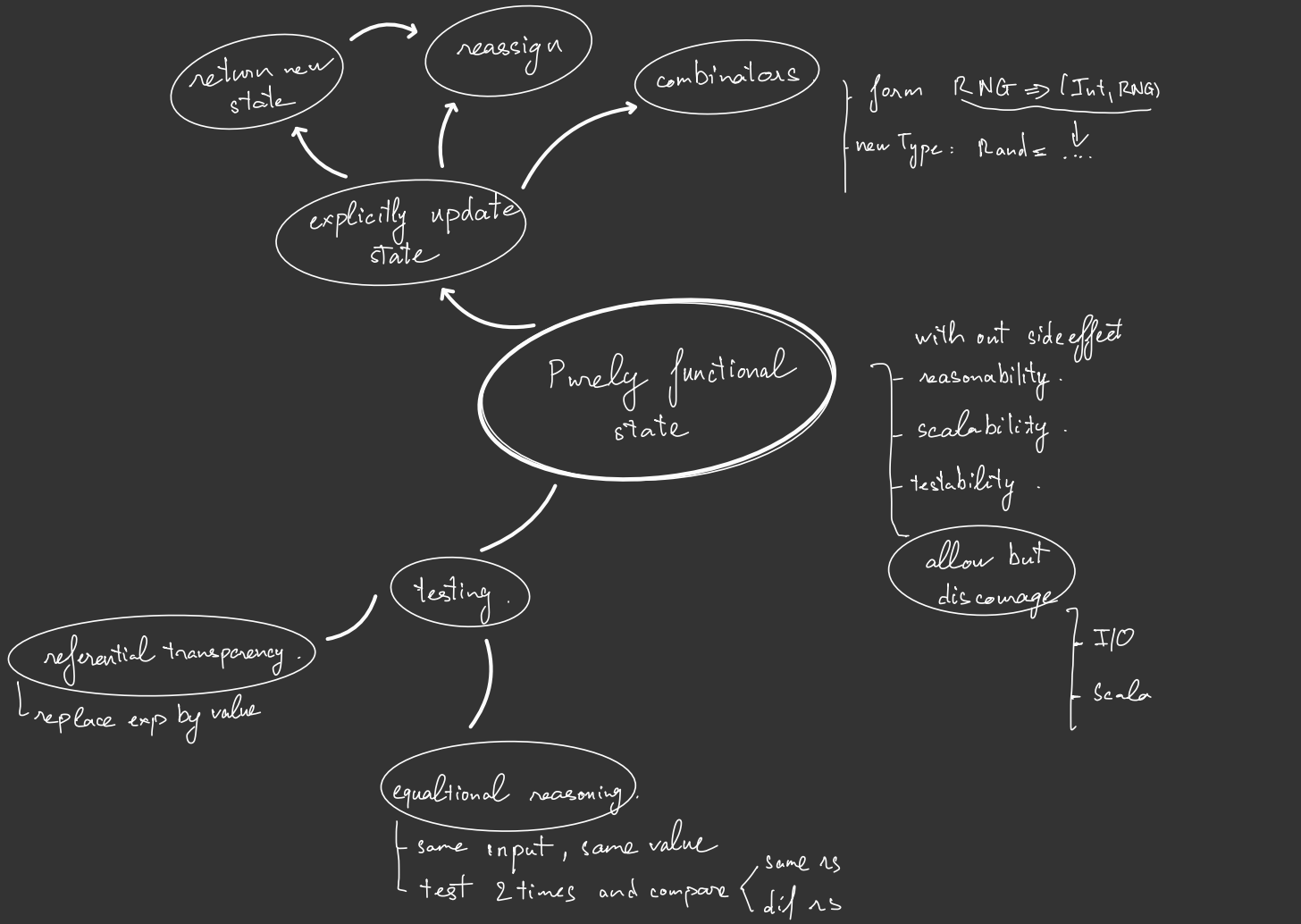
+ flatMap(fa)(A ⇒ F[B]): fb

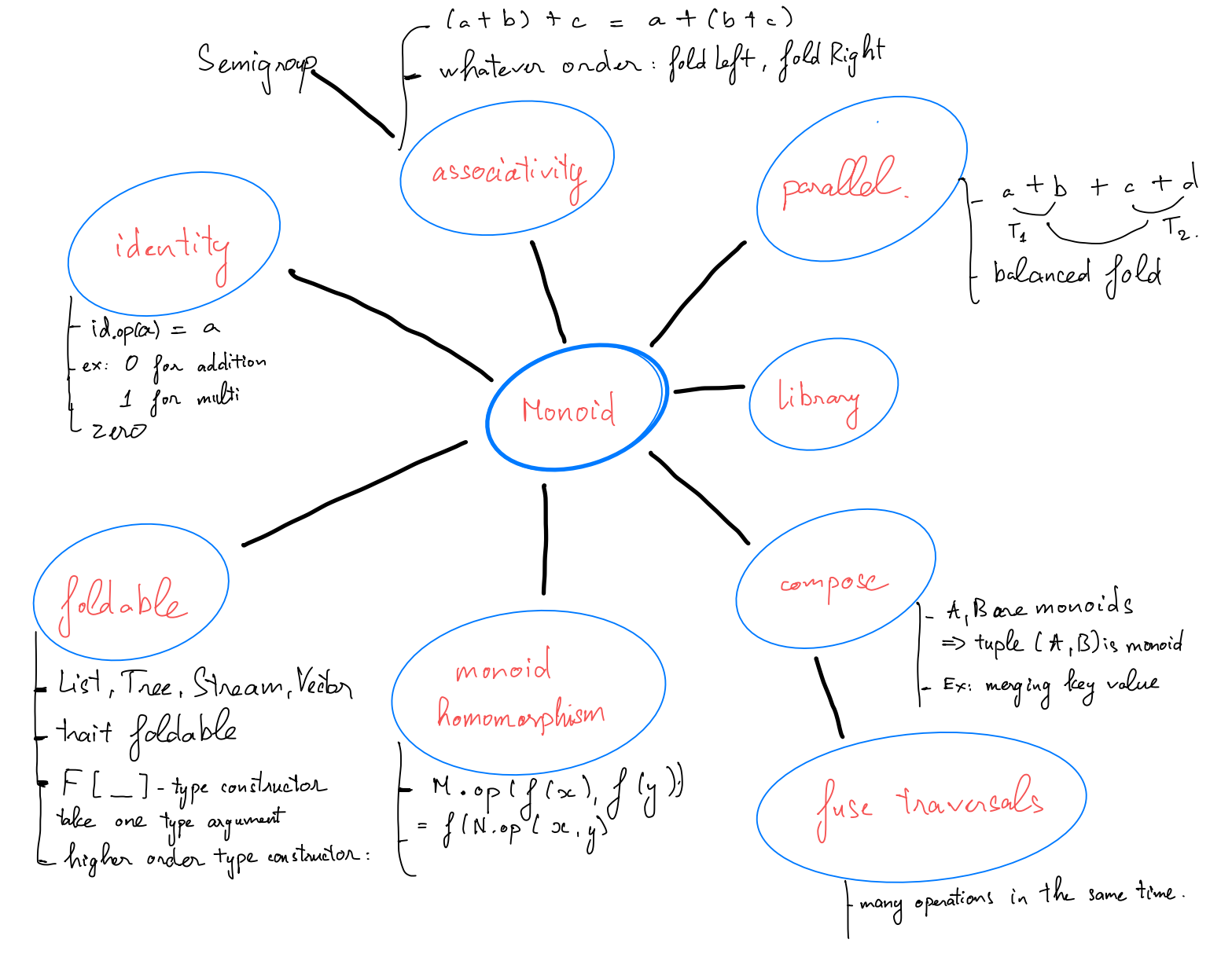
Monad

Function K

Function K


None
Option[Nothing]
Some
Option[A]
case Some(A(..))
Option
Right
Left
Either
Avoid Exception
Imagine Railway
Option World
Success
Future
Try
Try(f) to Opt
Failure
async
non-blocking
concurrent. fork join
Error / Exception
Exception break RT
are not type safe
Lift Junction
map, flatMap
getOrElse, filter
...
Promises
* not impt
create Future
"p success value"
"p failure ex"
"p completeWith fut"


return new state
reassign
combinators
form RNG ⇒ (Int, RNG)
new Type: Rand = ...
explicitly update state
Purely functional state
with out side effect
reasonability.
scalability.
testability.
allow but discourage
I/O
Scala
testing.
referential transparency.
replace exp by value
equational reasoning.
same input, same value
test 2 times and compare
same rs
dif rs


Semigroup
(a + b) + c = a + (b + c)
whatever order : fold left, fold Right
associativity
identity
id.op(a) = a
ex: 0 for addition
1 for multi
zero
parallel
a + b + c + d
T1
T2
balanced fold
Monoid
library
foldable
List, Tree, Stream, Vector
trait foldable
F[_] - type constructor
take one type argument
higher order type constructor :
monoid
Homomorphism
M.op(f(x), f(y))
= f(N.op(x, y))
compose
A, B are monoids
=> tuple (A, B) is monoid
Ex: merging key value
fuse traversals
many operations in the same time.