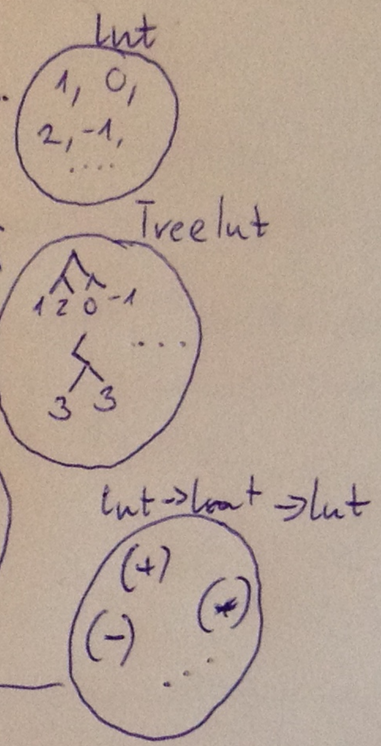
Type ~ Set

- set of „simple values" ←
- set of „complex values"
- set of „abstract complex values"
- set of functions



Typeclass ~ Algebraic structure

(Type, func) + rules = = typeclass

(S, •) + laws = = algebraic structure

e.g.
(Int, (+)) can be an instance of Semigroup typeclass, because
$\forall x,y,z :: Int :$
$(x+y)+z = x+(y+z)$

$(\mathbb{Z}, +)$ is semigroup, because
$\forall a,b,c \in \mathbb{Z}:$
$(a+b)+c = a+(b+c)$

$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

Concrete and Abstract types

"type functions" F : type → type e.g. Tree :: a → Tree a

type constructor („function")

abstract type

type parameter

abstract type: „encloses", „contains" or „gives a context to" other types

Int → Int ⇒ a → b
a → b ⇒ (→) a b
type cons. type params

concrete: doesn't have any unsubstituted (unbound) type parameters, e.g. Tree Int, Tree (Tree Int)

abstract: you have at least one unbound type parameter, e.g. Tree (Tree a)
unbound

Syntax of typeclasses

name of typeclass — denotes a type

```
class Semigroup a where
  (<>) :: a -> a -- Usually "a" is part of
                 -- your type-signatures
```

method

Laws (not enforced!!):

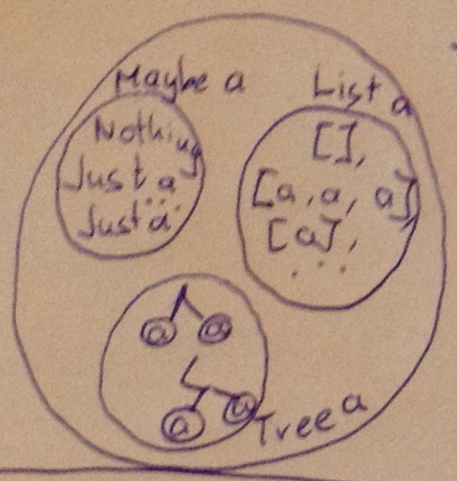
$\forall x,y,z :: a : (x <> y) <> z = x <> (y <> z)$

```
instance Semigroup Int where
  (<>) = (+)
```

$\forall x,y,z :: Int :$
$(x+y)+z = x+(y+z)$

Typeclasses of Abstract Types

typeclass ~"set of sets" (types)
((with methods and laws))



Typeclass for "container-like" or „context-like" abstract types

„API"

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a->b) -> f a -> f b
```

List a:

$a_1 \xrightarrow{pure} [a_1]$

$[fun :: a \to b] \langle * \rangle [a_1, a_2, a_3 \dots] \to [b_1, b_2 \dots]$

Applicative is more powerful than Functor

```
fmap f l = pure f <*> l
```

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

„API" — function — enforces „f" to be an abstract type, because it's applied to type parameter „a" („b").

Rules:

```
fmap id = id
fmap (g . h) =
  fmap g . fmap a
```

List a:

function

$[a_1, a_2, a_3, \dots] \xrightarrow{fmap(a \to b)} [b_1, b_2, b_3, \dots]$

```
class Applicative m => Monad' m where
  return :: a -> m' a   (same as pure)
  (>>=) :: m a -> (a -> m b) -> m b
```

„bind" operator

List a:

eg. $[b_{11}, b_{12}, \dots]$

$[a_1, a_2, a_3 \dots] >>= (a \to [b]) \to [b_{11}, b_{12}, \dots b_{21}, b_{22}, \dots]$