Type introduction illustrated

for Haskell newcomers

get over the Foldable

Takenobu T.

Rev. 0.00.0
WIP

“What is this description ?!“

```
foldr ::  Foldable t  =>  (a -> b -> b) -> b -> t a -> b
```

NOTE

- This shows one of the mental model.
- Please see also references.
- This is written for Haskell, especially later ghc7.10.

Contents

1. Introduction
  - Simple question
  - Type
  - Type class

2. Types
  - Type
  - Parametric polymorphism type
  - Type constructer

3. Type classes
  - Type class

Appendix I – various types
- Bool
- Char
- Int, Integer, Float
- List
- Maybe
- Either

Appendix II – various type classes
- Eq, Ord
- Num
- Foldable
- Monoid
- Functor, Applicative, Monad

References

1. Introduction

Values, Types, Type classes

Values

'a'

1.0

False

1

2

'h'

1.5

'5'

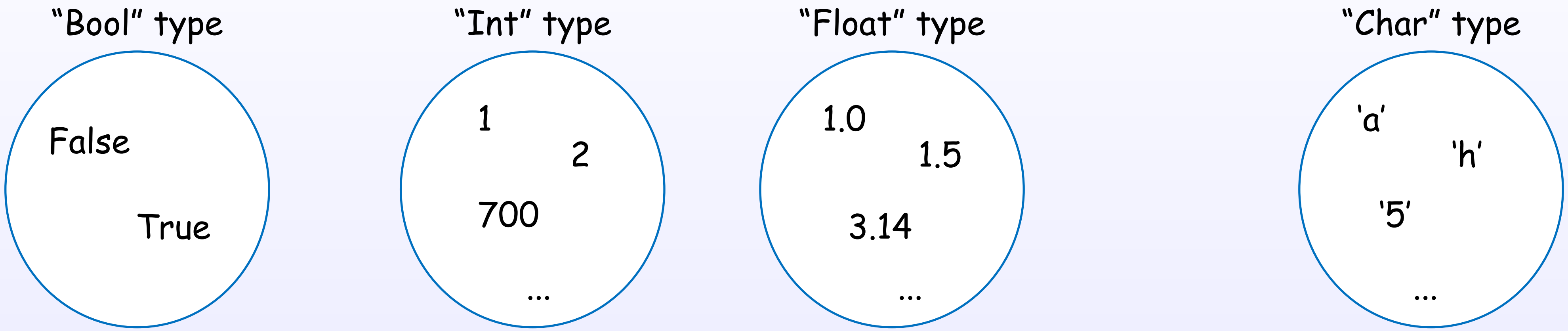
True

700

3.14

References : @@@

Types



"Bool" type

False

True

"Int" type

1

2

700

...

"Float" type

1.0

1.5

3.14

...

"Char" type

'a'

'h'

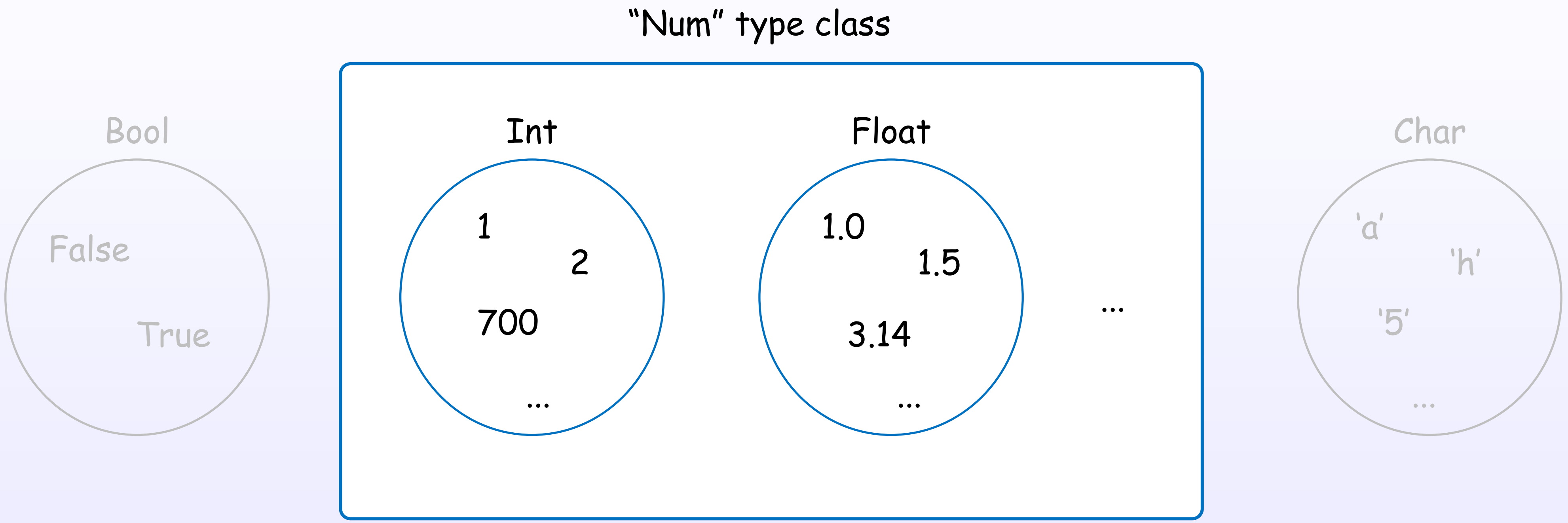
'5'

...

A type is a collection of values which has common property.

References : @@@

Type classes



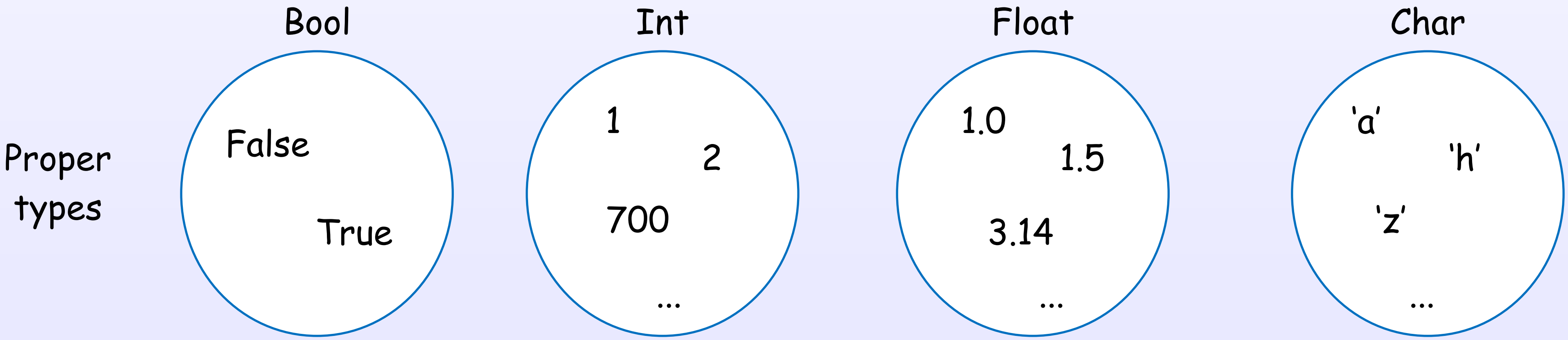
A type class is a collection of types which has common computations.

References : @@@

1. Introduction

Proper types, Quantified types

Proper types



Proper types

Bool

False

True

Int

1

2

700

...

Float

1.0

1.5

3.14

...

Char

'a'

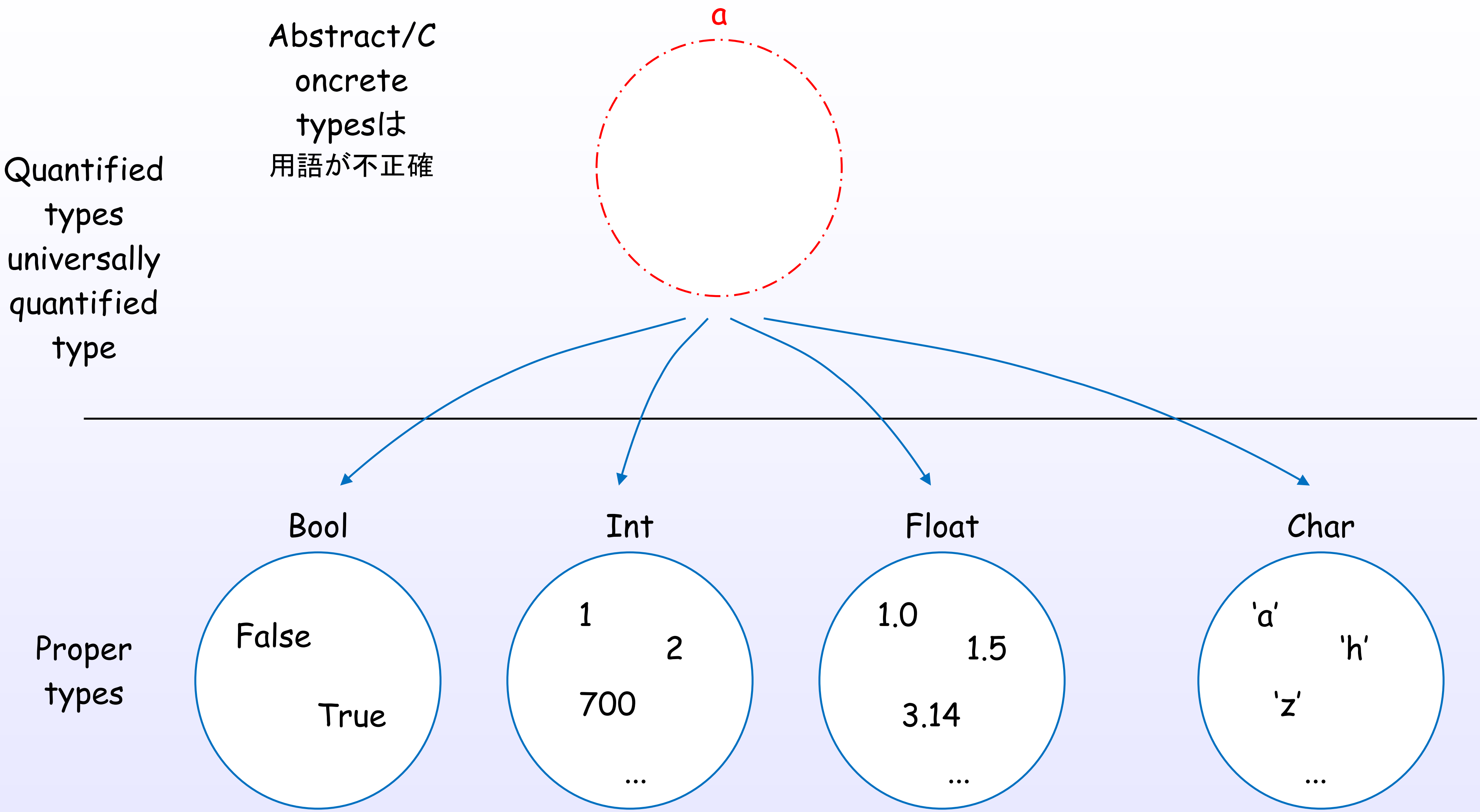
'h'

'z'

...

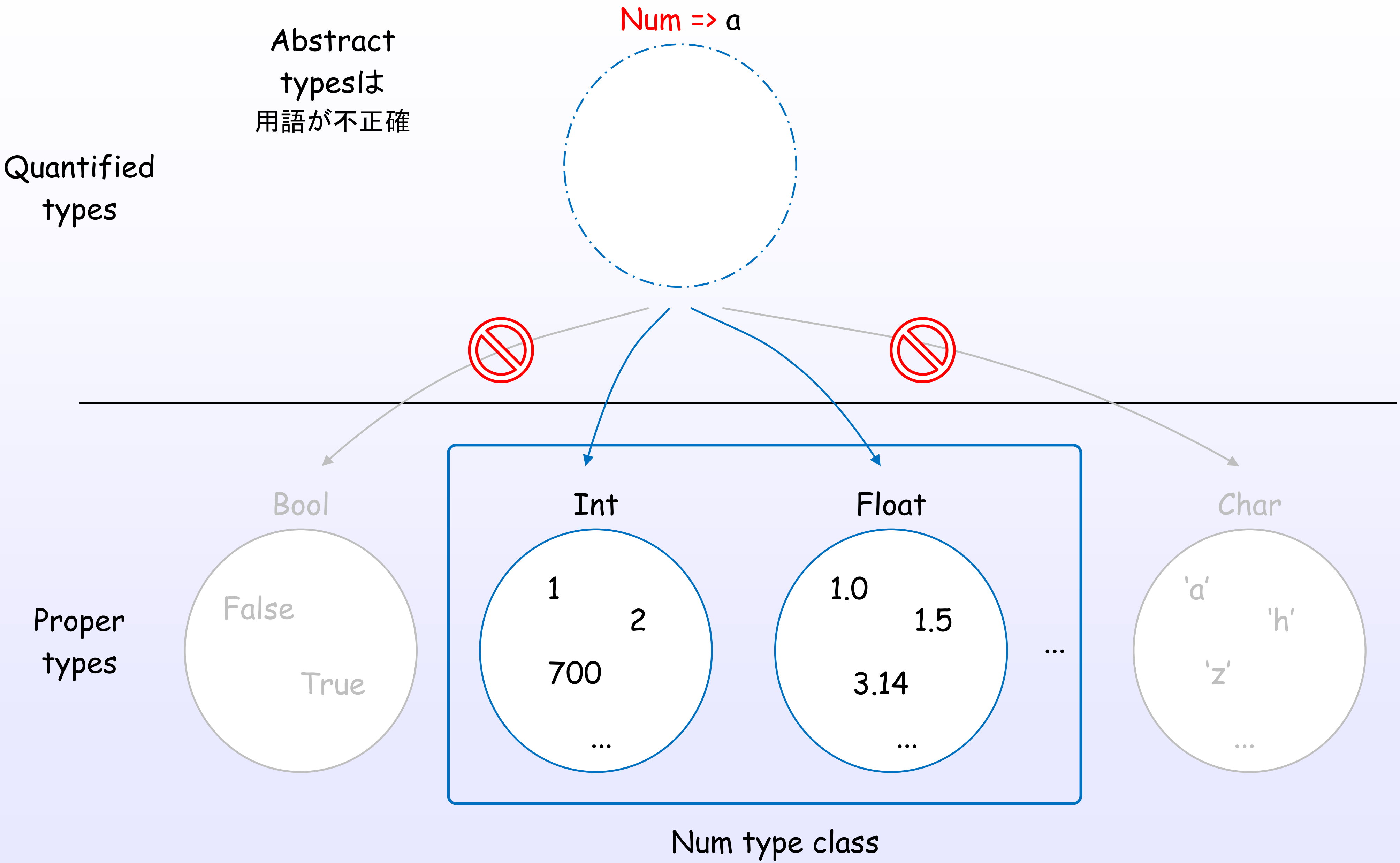
References : @@@

Quantified types


Abstract/Concrete typesは用語が不正確
a
Quantified types universally quantified type
Bool
Int
Float
Char
Proper types
False
True
1
2
700
...
1.0
1.5
3.14
...
'a'
'h'
'z'
...

References : @@@

Quantified types restrained with type classes



Abstract typesは用語が不正確

Quantified types

Num => a

Proper types

Bool

False

True

Int

1

2

700

...

Float

1.0

1.5

3.14

...

...

Char

'a'

'h'

'z'

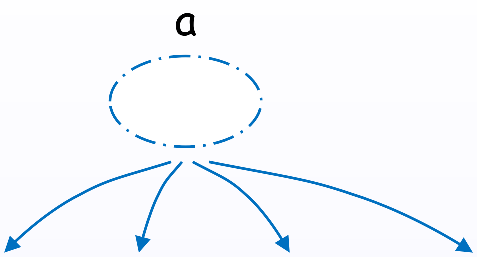
...

Num type class

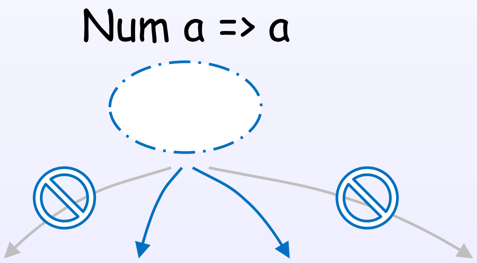
References : @@@

Quantified types

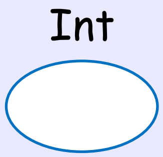
Universal
Quantified
types



Restricted
Quantified
types



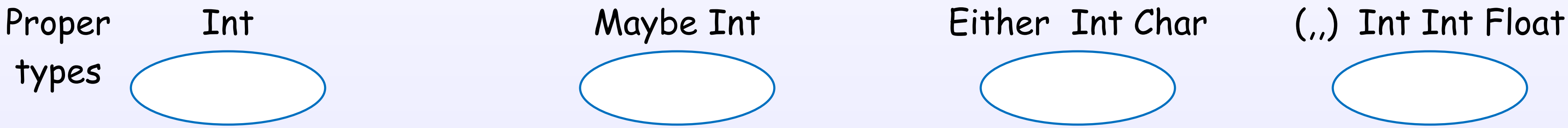
Types



References : @@@

Higher-order types

Higher-order types



Proper types

Int

Maybe Int

Either Int Char

(,,) Int Int Float

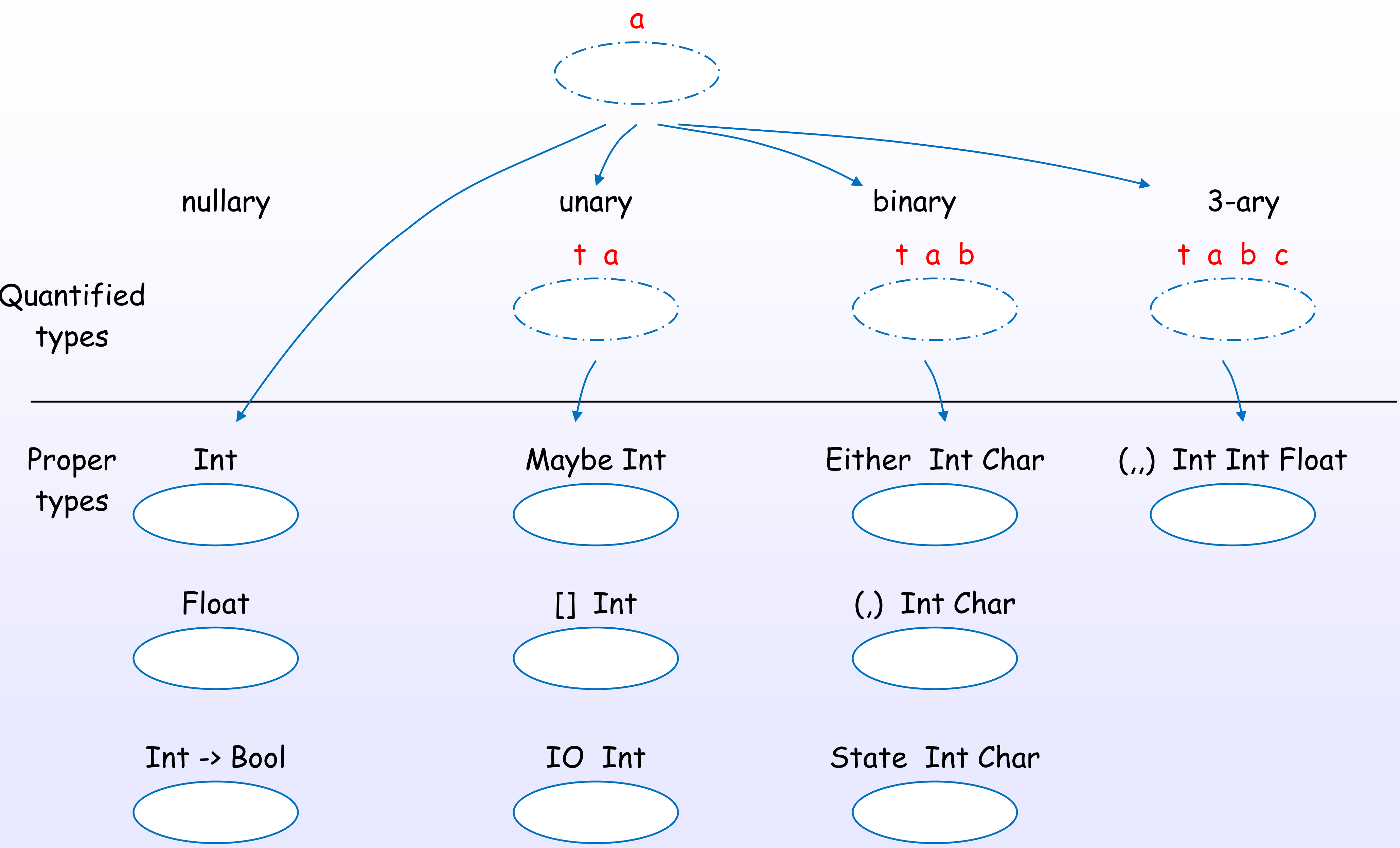
References : @@@

Higher-order types

	Higher-order types			
nullary	unary	binary	3-ary	...
Int	Maybe Int	Either Int Char	(,,) Int Int Float	
Float	[] Int	(,) Int Char		
Int -> Bool	IO Int	State Int Char		

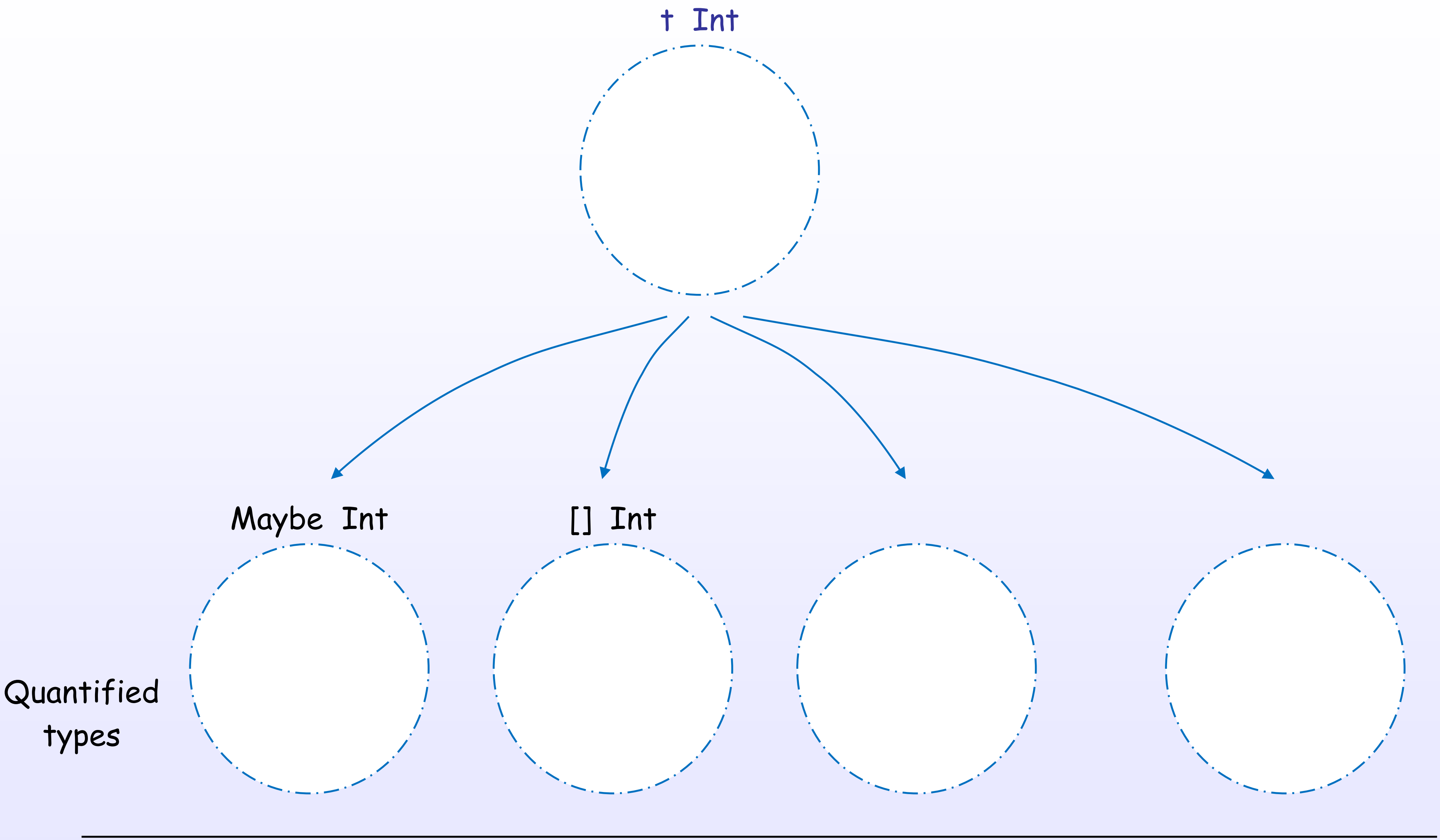
References : @@@

Higher-order types and Quantified types



References : @@@

higher-order and Quantified type



References : @@@

1. Introduction

summary temp

Higher-order types and Quantified types

	non higher-order types	Higher-order types
Quantified types	a	Maybe a [a] (a, b) t b :
Proper types	Int Char Float :	Maybe Int [Char] (Float, Int) :

References : @@@

1. Introduction

Simple question

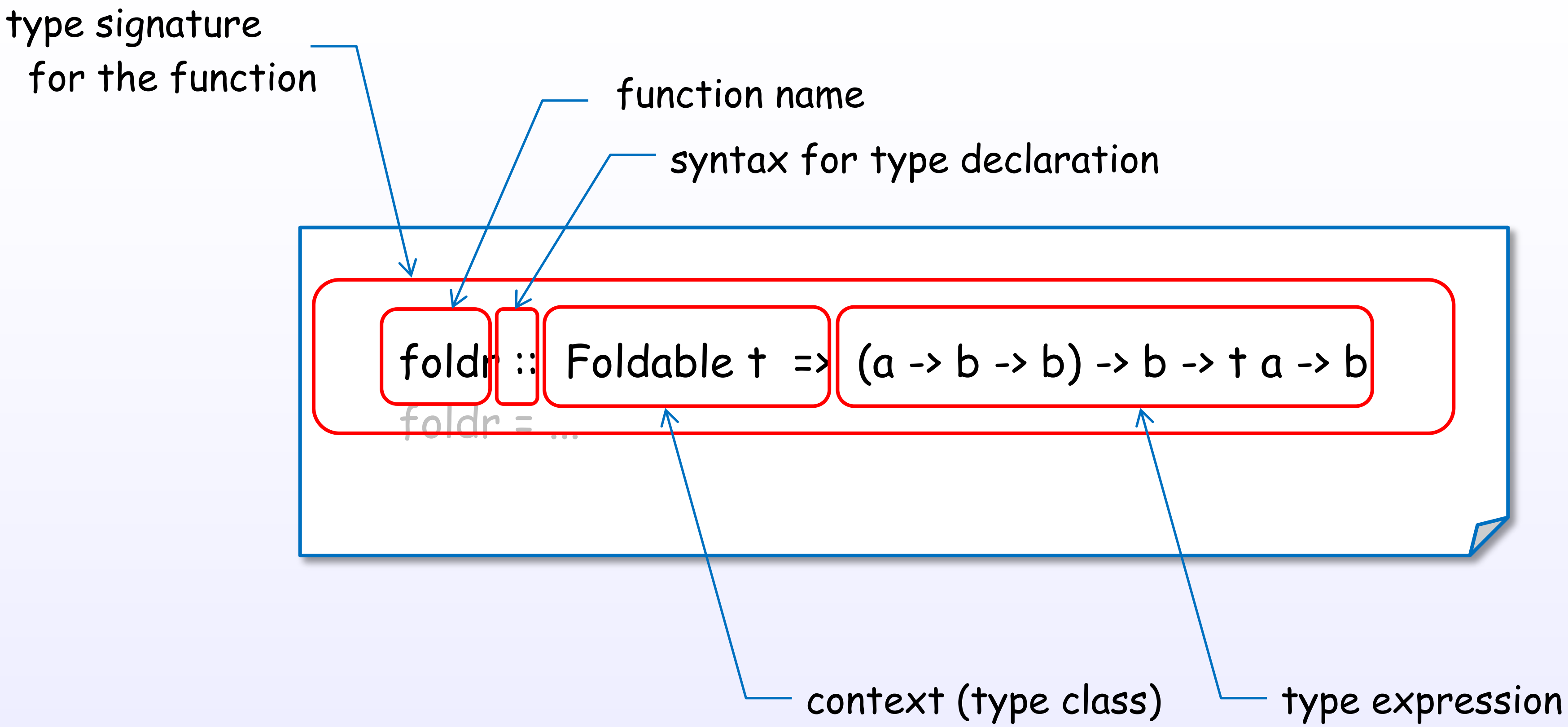
What is this ?!

```
foldr ::  Foldable t  =>  (a -> b -> b) -> b -> t a -> b
foldr = ...
```

?

References : @@@

What is this ?!



type signature for the function

function name

syntax for type declaration

```
foldr :: Foldable t  => (a -> b -> b) -> b -> t a -> b
foldr = ...
```

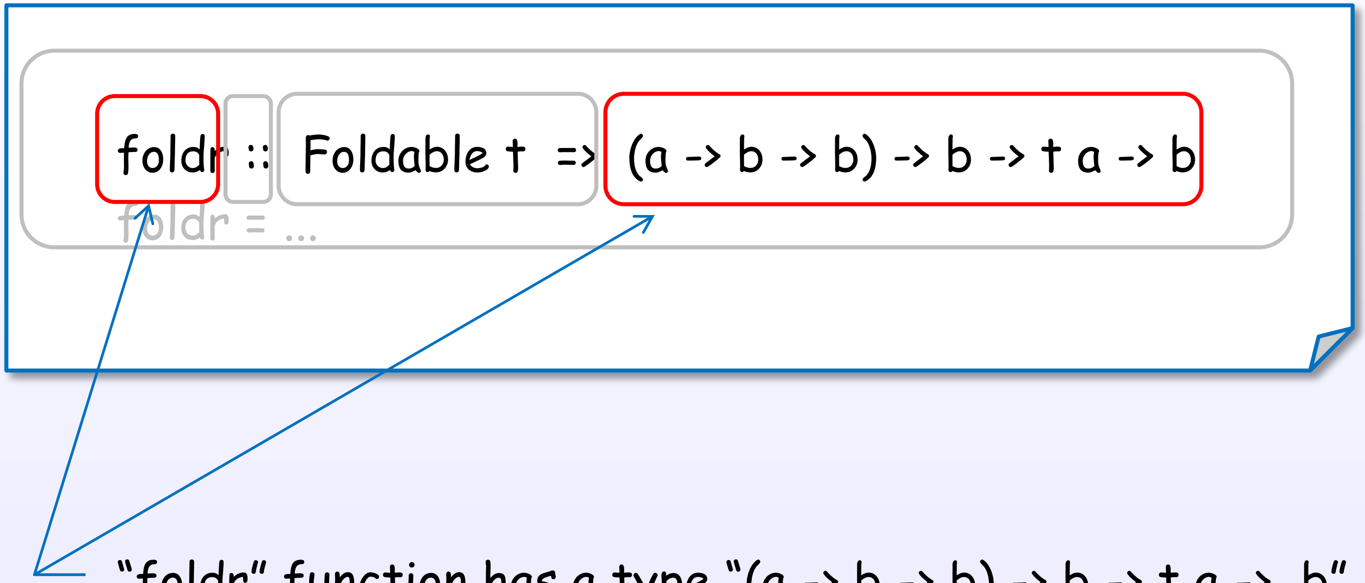
context (type class)

type expression

[H1] 4.1

References : @@@

What is this ?!

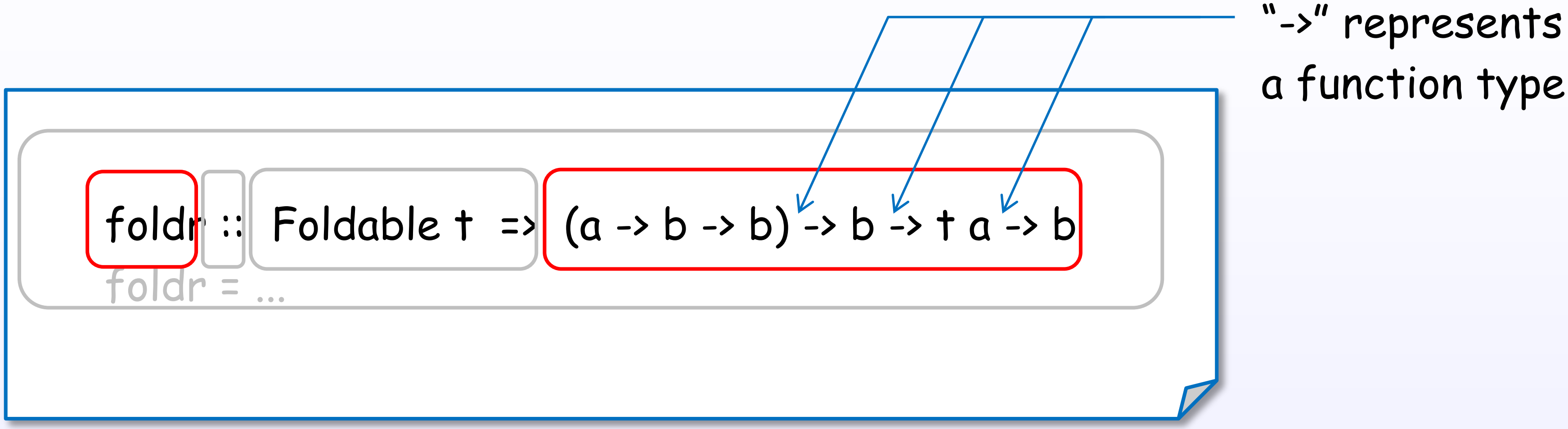


```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr = ...
```

“foldr” function has a type “(a -> b -> b) -> b -> t a -> b”.

References : @@@

What is this ?!



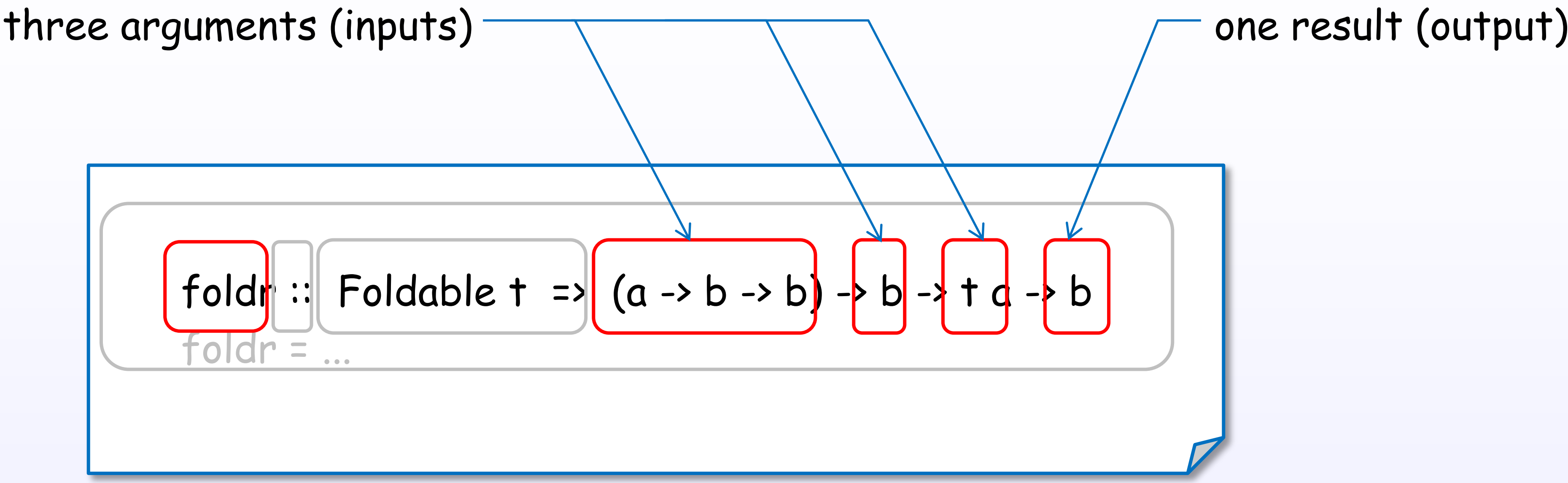
```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr = ...
```

“->” represents a function type

“foldr” is a function.

References : @@@

What is this ?!

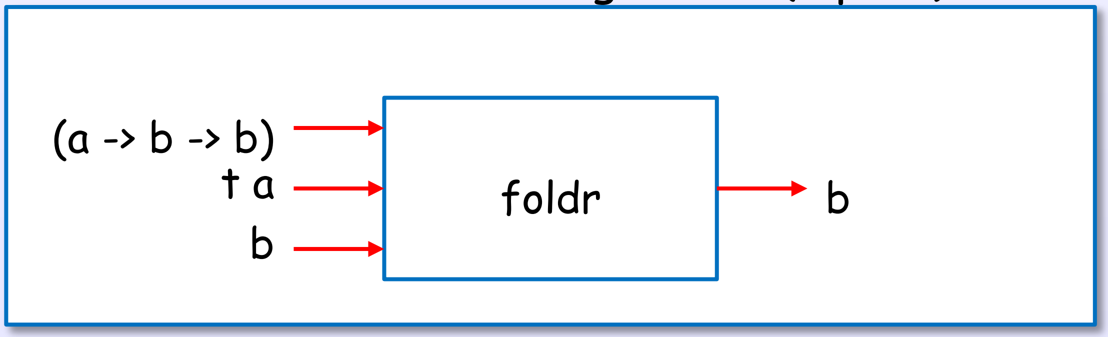


three arguments (inputs)

one result (output)

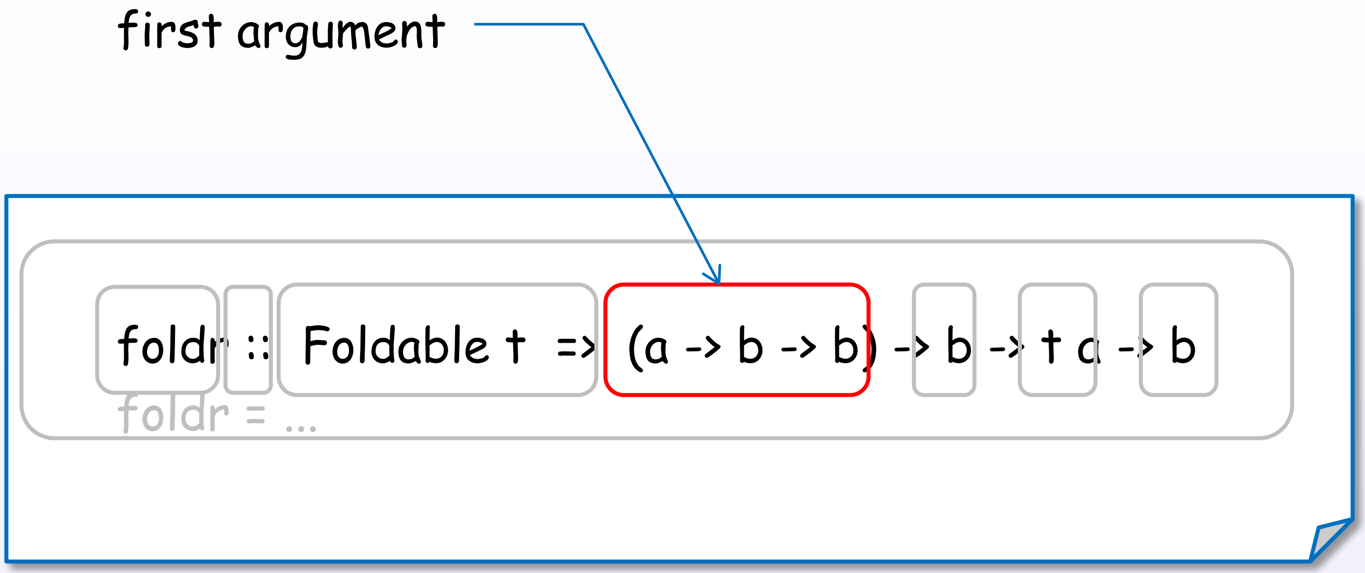
```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr = ...
```

“foldr” function has three arguments(inputs) and one result(output).



References : @@@

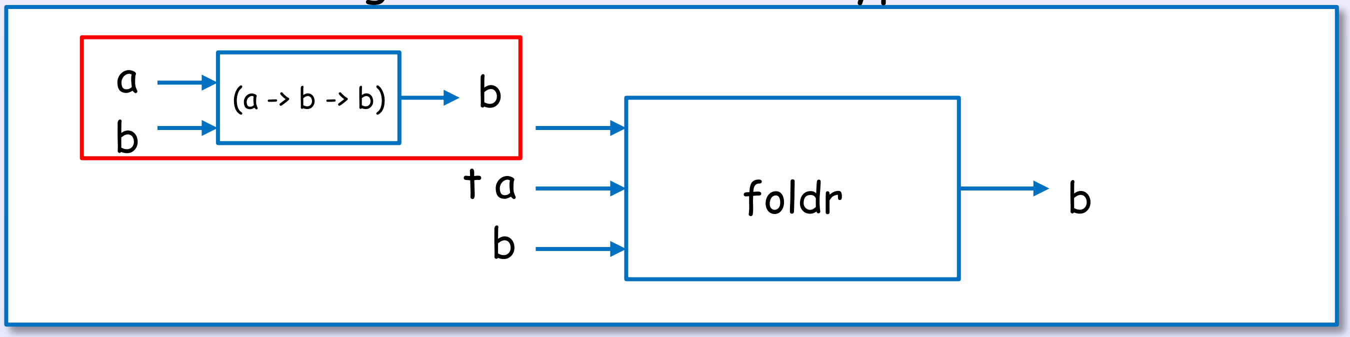
What is this ?!



first argument

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr = ...
```

First argument is a function type.



a

b

(a -> b -> b)

b

t a

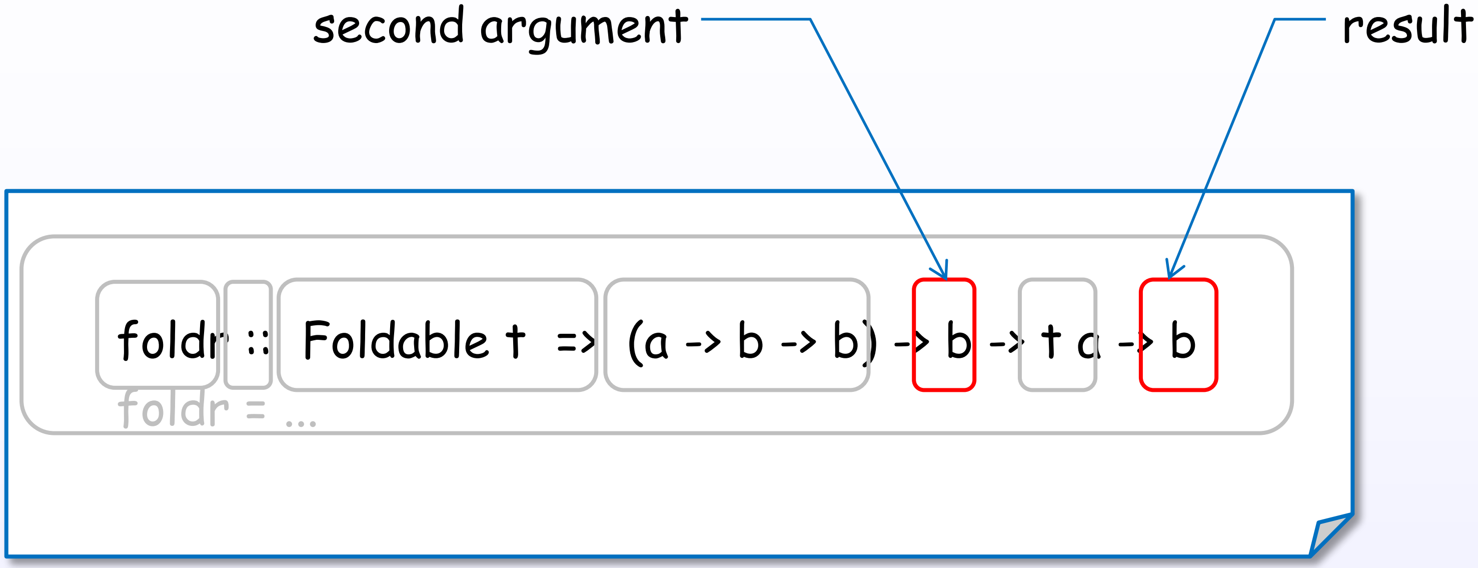
b

foldr

b

References : @@@

What is this ?!

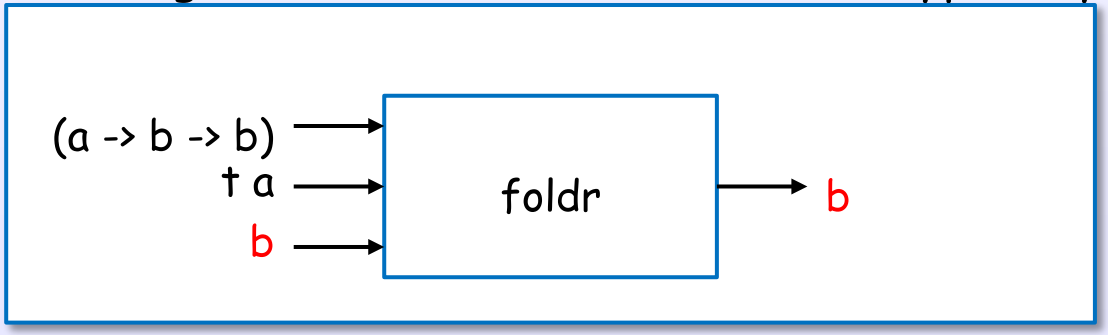


second argument

result

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr = ...
```

Second argument and the result are same type (any type “b”).



(a -> b -> b)

t a

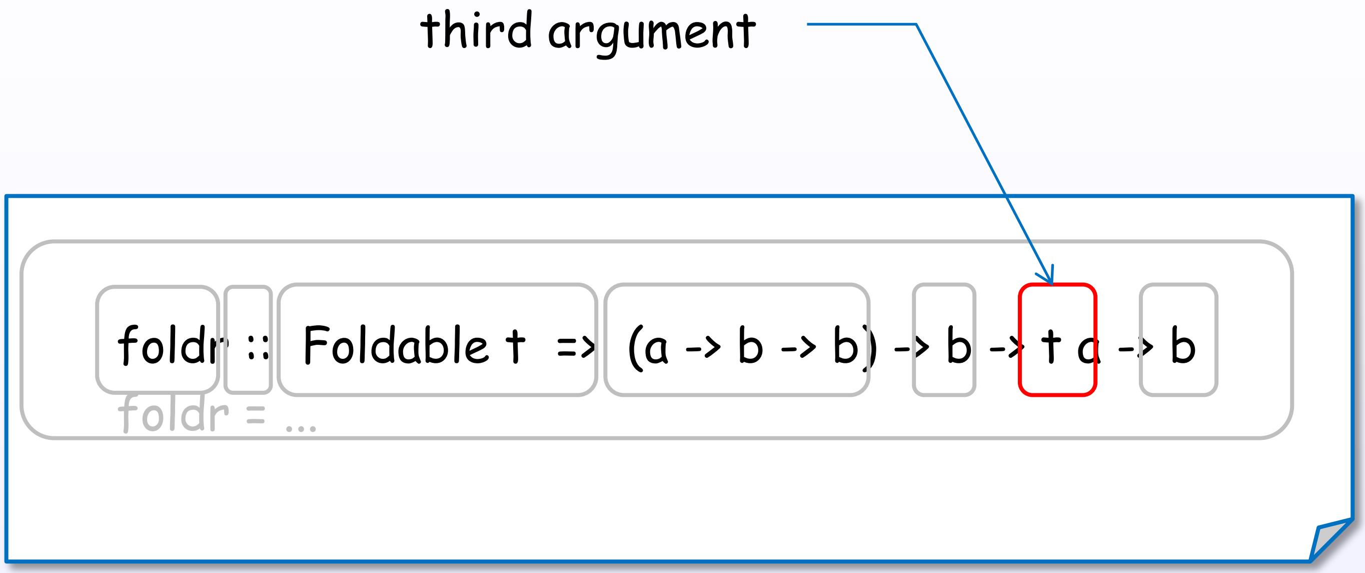
b

foldr

b

References : @@@

What is this ?!



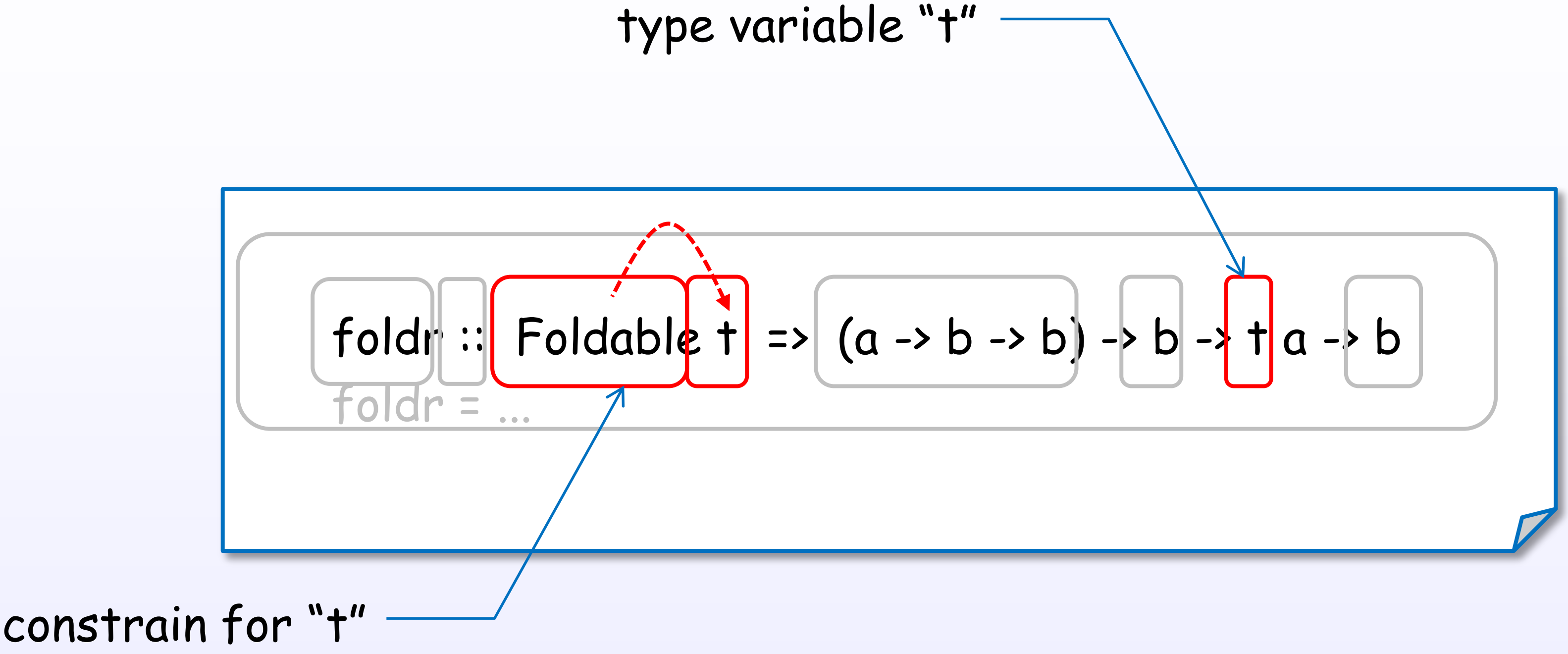
third argument

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr = ...
```

Third argument is a constructed type with type variable “t” and “a”.

References : @@@

What is this ?!



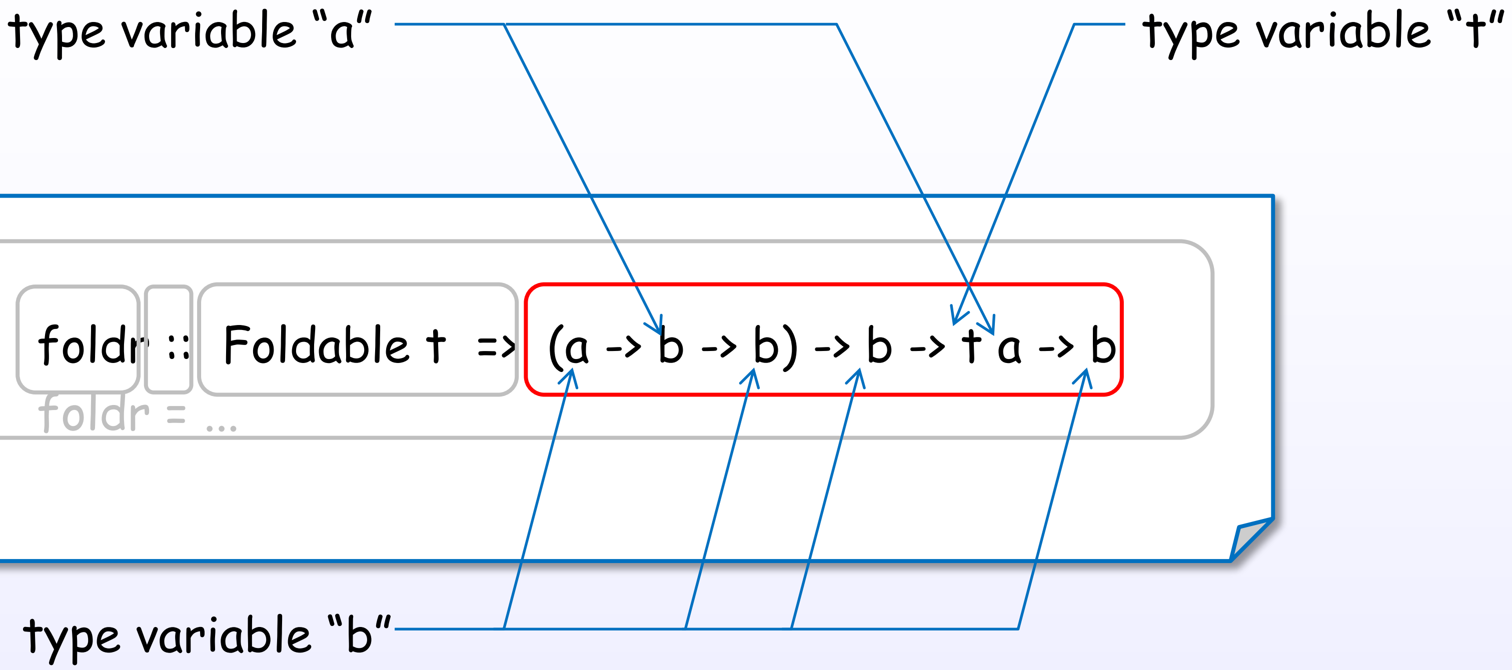
type variable “t”

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr = ...
```

constrain for “t”

The type variable “t” is belonged with “Foldable” type class.

What is this ?!



type variable “a”

type variable “t”

```
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr = ...
```

type variable “b”

“foldr” function has three type variables (“a”, “b” and “t”).
Type variable “a” and “b” is any type.
Type variable “t” is belonged with “Foldable” type class.

References : @@@

Value, Type, Type class

Type classes

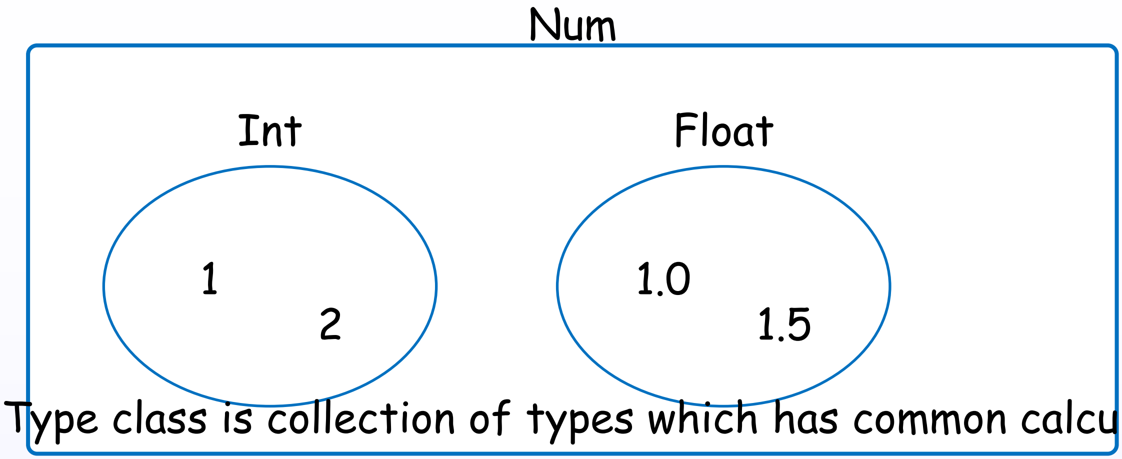
Types

Values

References : @@@

Value, Type, Type class

Type classes



Num

Int

1

2

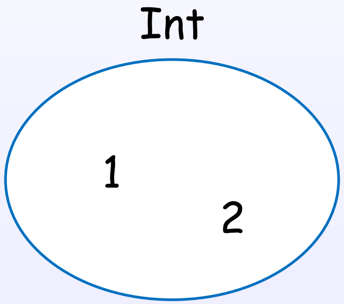
Float

1.0

1.5

Type class is collection of types which has common calculation.

Types



Int

1

2

Type is collection of values which has common property.

Values

1

2

A

References : @@@

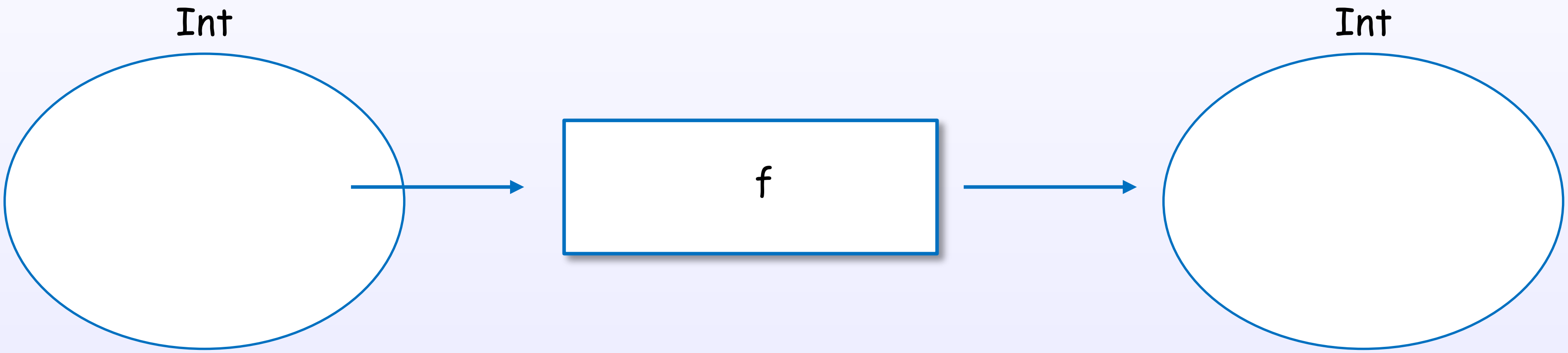
Value, Type, Type class

Type classes	a	Maybe a
Types	Num => a	Num => Maybe a
Values	Int	Maybe Int

References : @@@

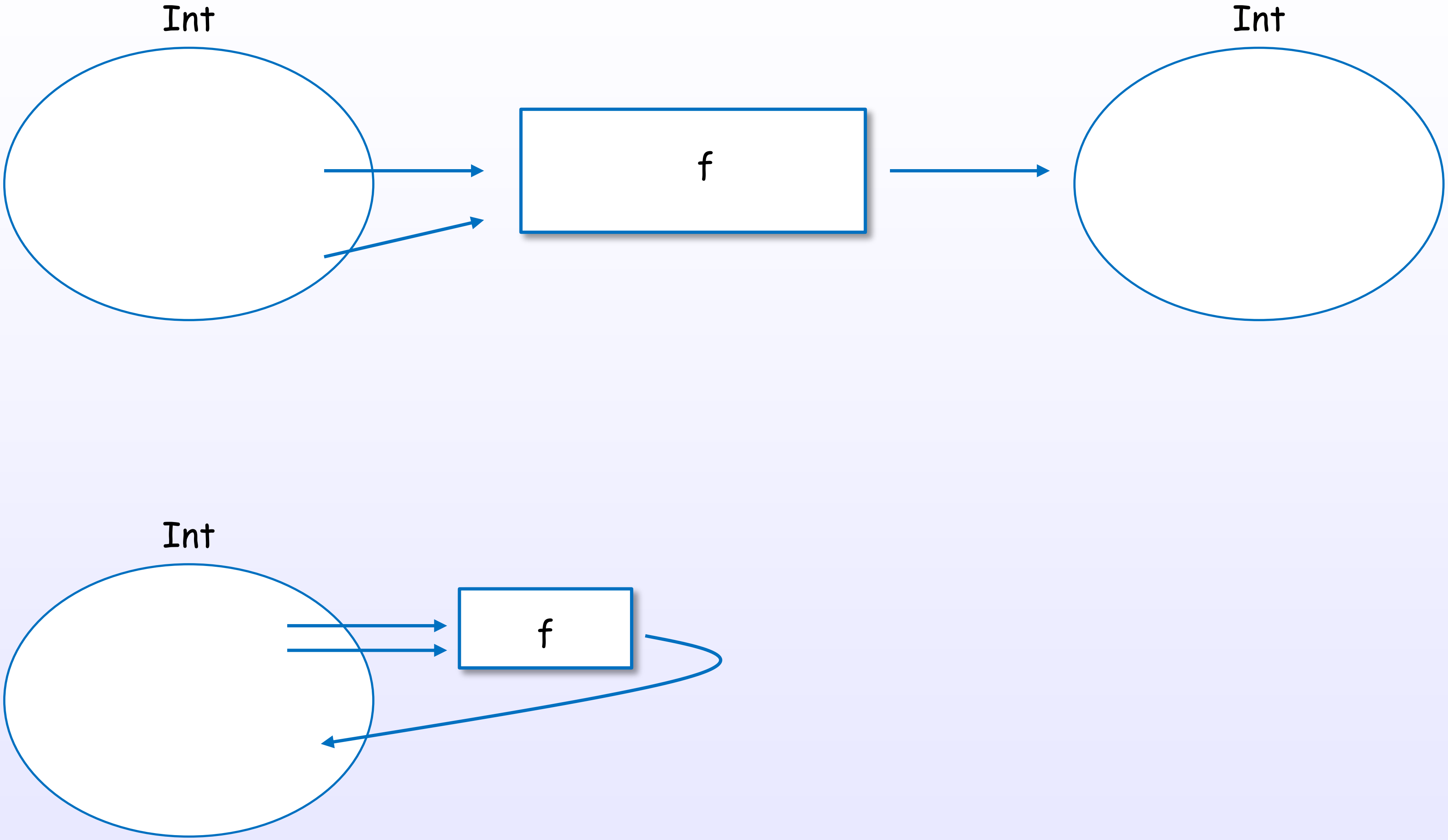
type

f :: Int -> Int



References : @@@

Each view



References : @@@

type

f :: a -> a

for all

?

All or One?

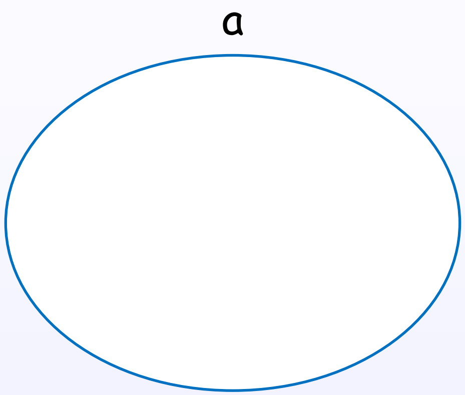
Are there intermediate?

Proper, specialize

f :: Int -> Int

References : @@@

type class

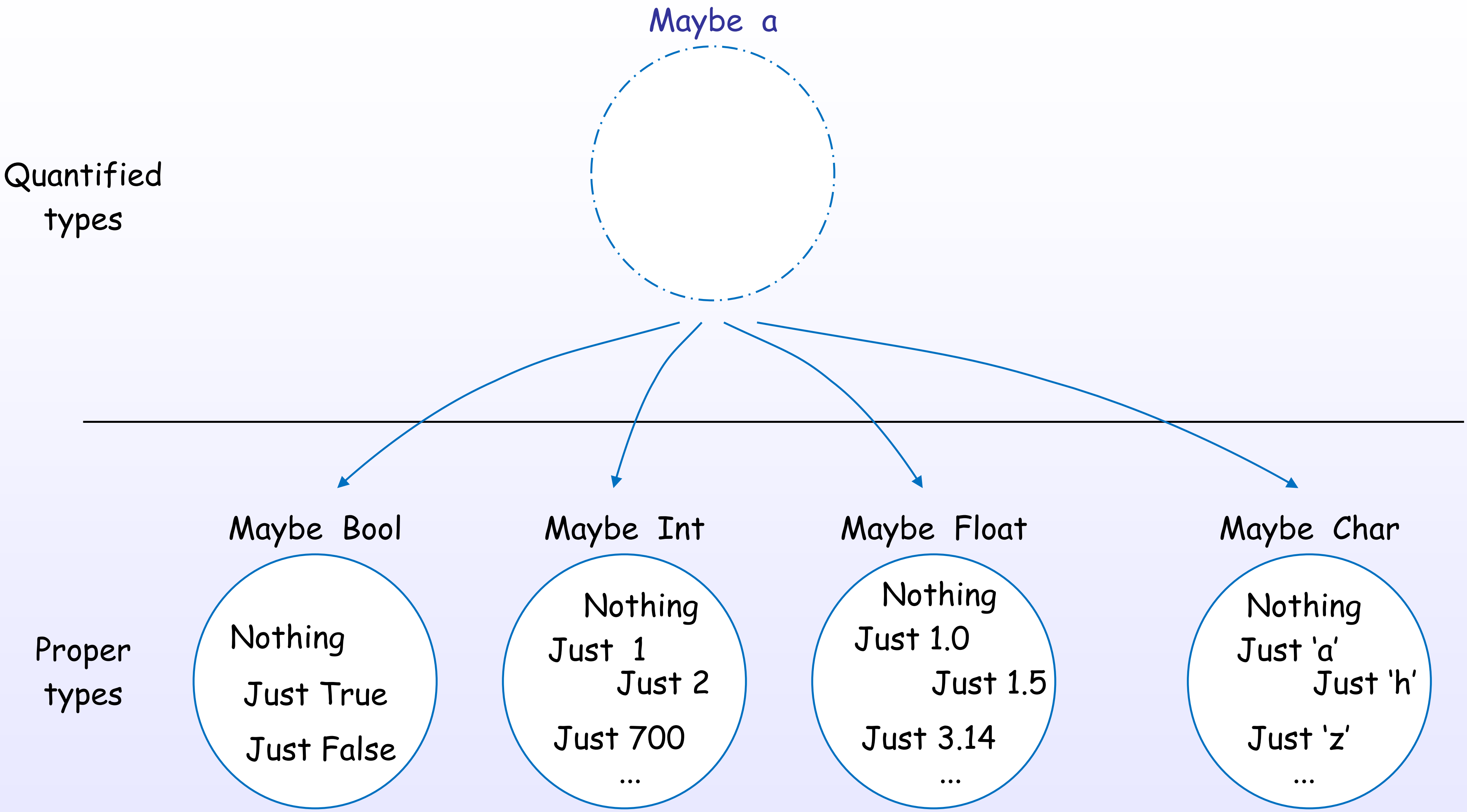


```
class Num
  (+) :: ...
```

References : @@@

Appendix I - various types

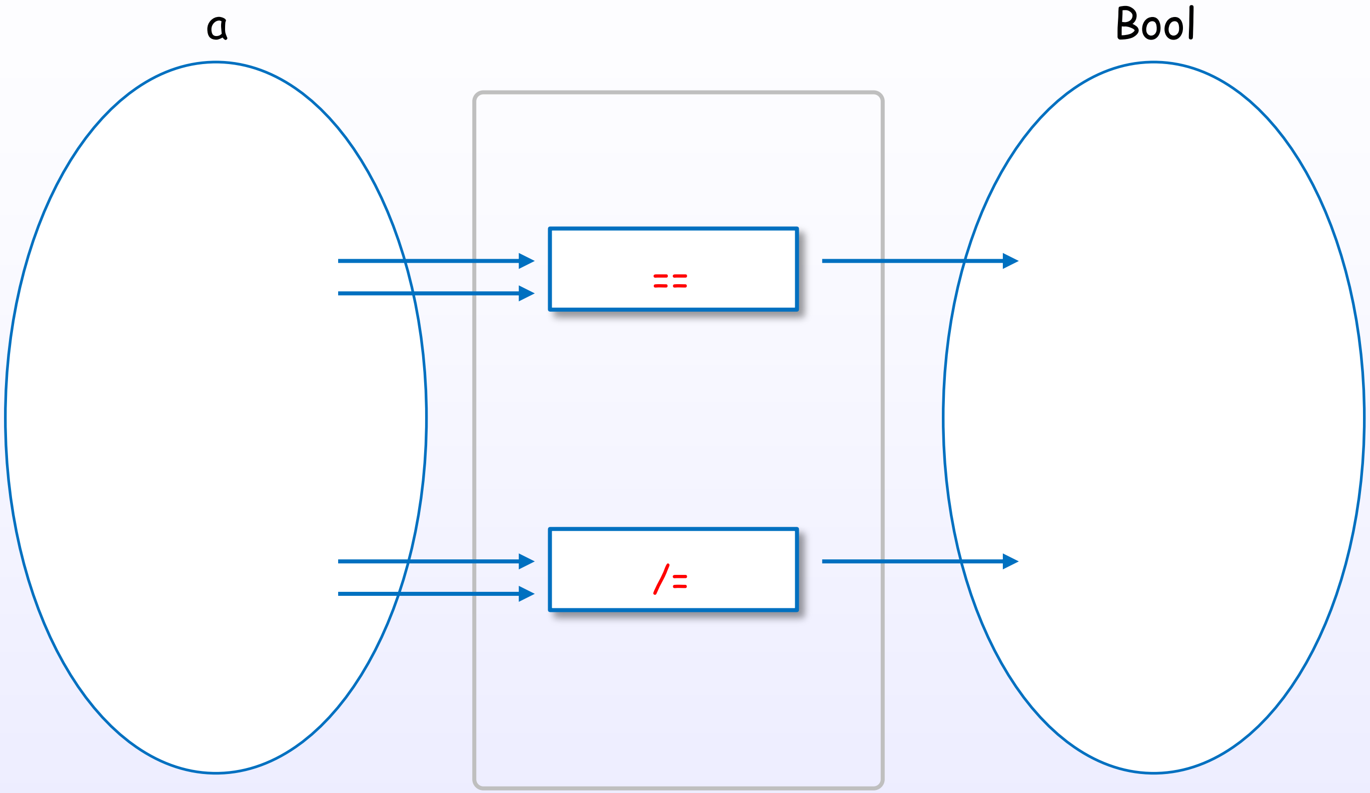
Maybe class



References : @@@

Appendix II - various type classes

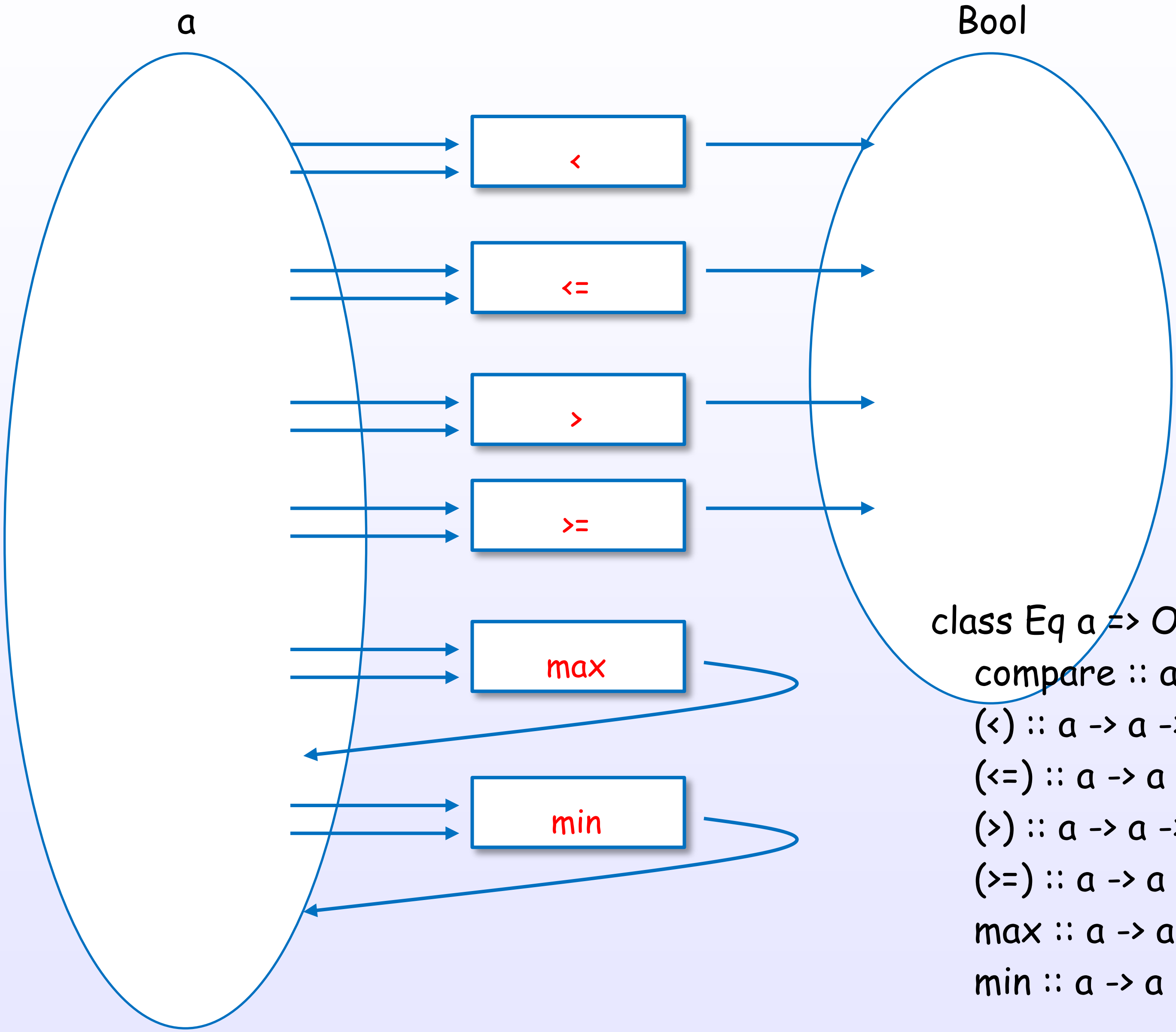
Eq class



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

References : @@@

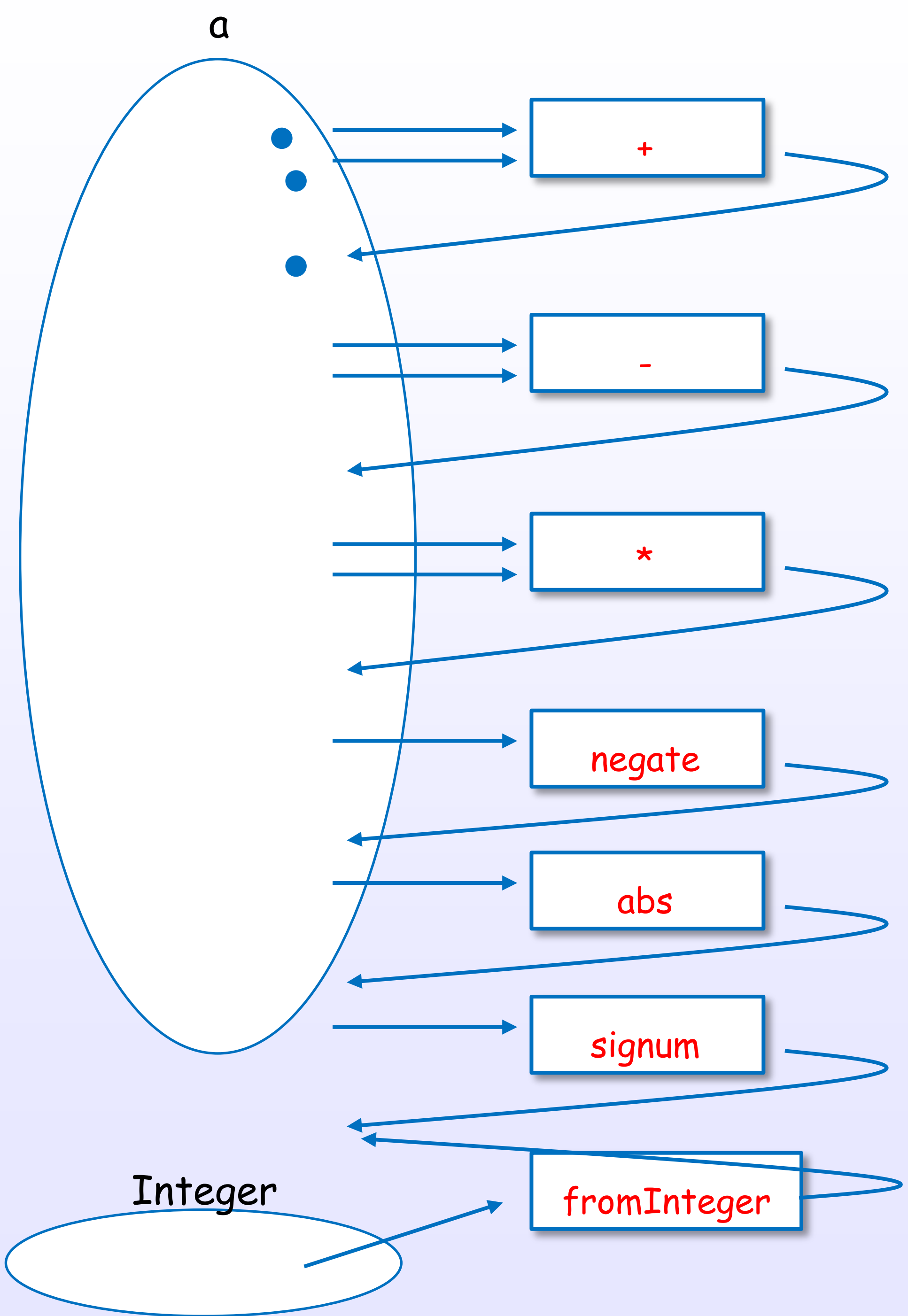
Ord class



```
class Eq a => Ord a where
   compare :: a -> a -> Ordering
   (<) :: a -> a -> Bool
   (<=) :: a -> a -> Bool
   (>) :: a -> a -> Bool
   (>=) :: a -> a -> Bool
   max :: a -> a -> a
   min :: a -> a -> a
```

References : @@@

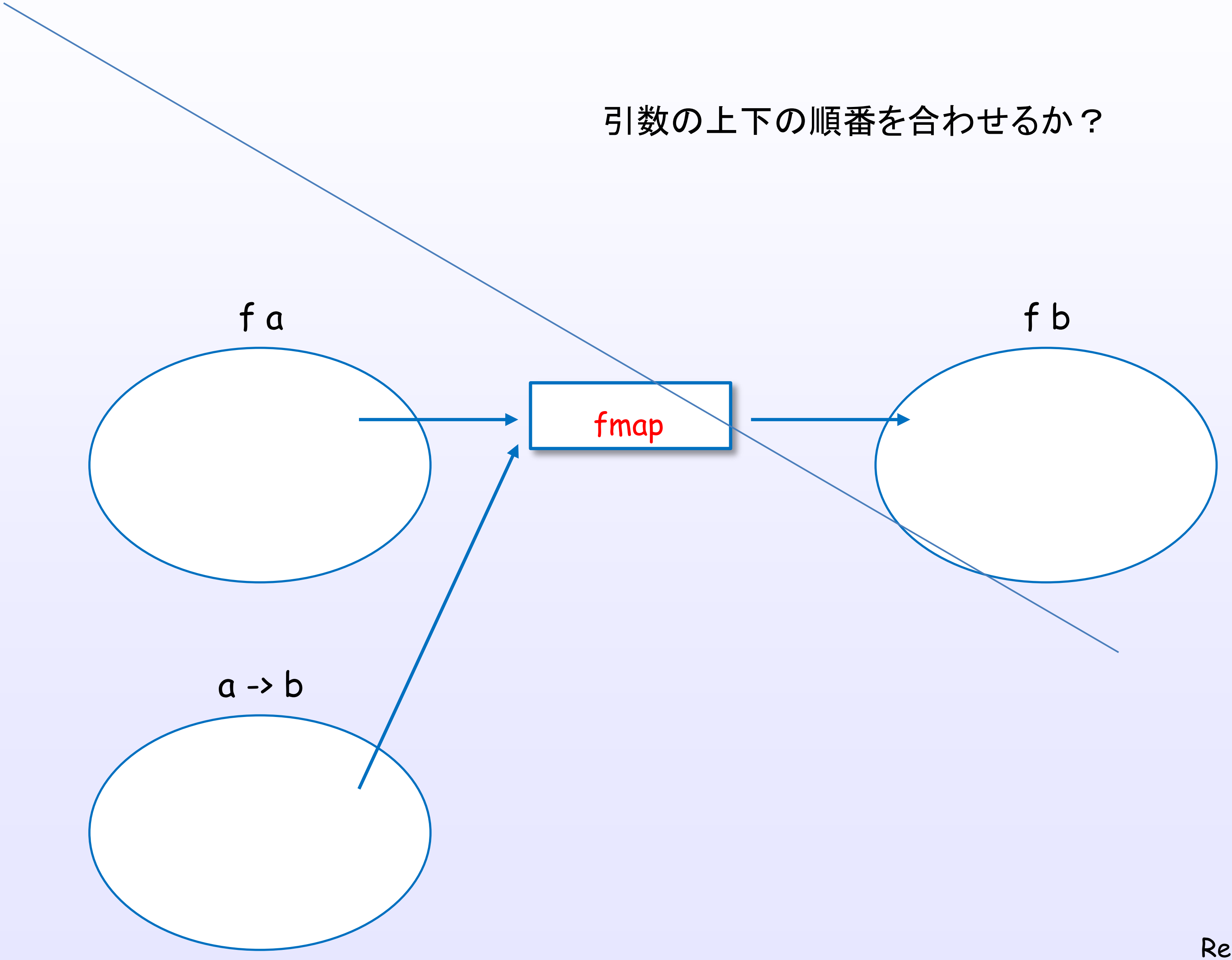
Num class



```
class  Num a  where
    (+), (-), (*)        :: a -> a -> a
    negate               :: a -> a
    abs                  :: a -> a
    signum               :: a -> a
    fromInteger          :: Integer ->
```

References : @@@

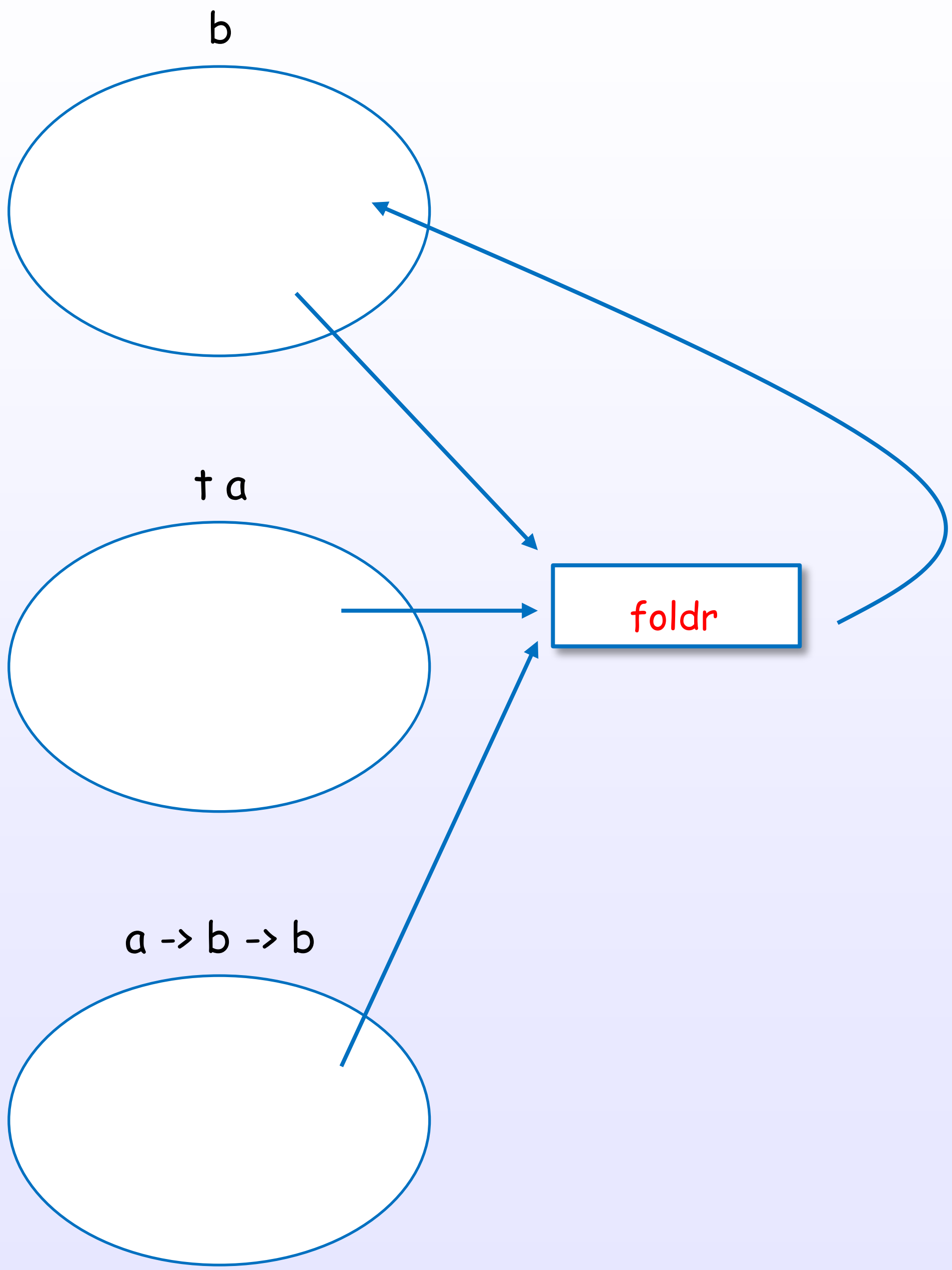
Functor class



引数の上下の順番を合わせるか？

References : @@@

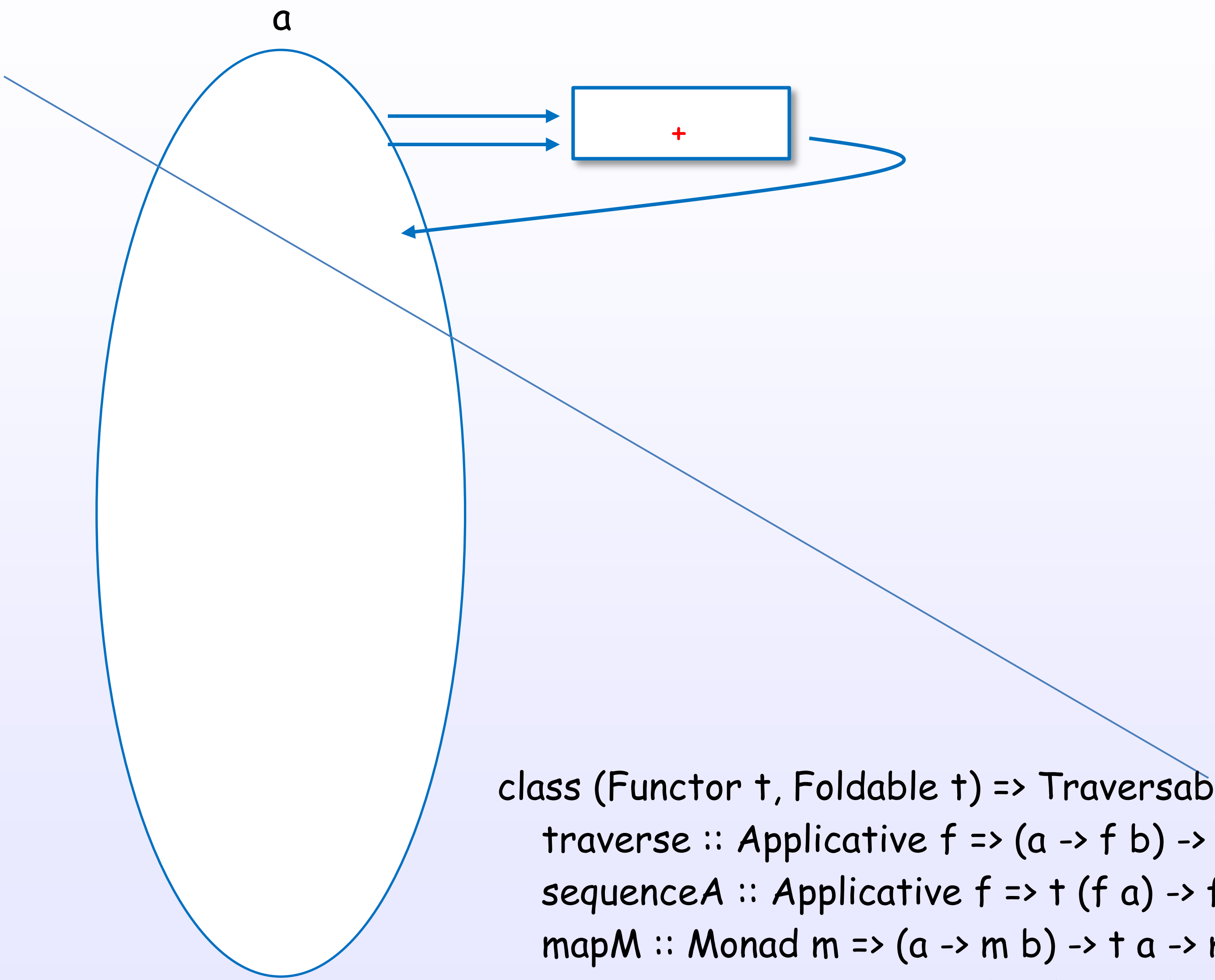
Foldable class



```
class Foldable t where
    fold :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr' :: (a -> b -> b) -> b -> t a -> b
    foldl :: (b -> a -> b) -> b -> t a -> b
    foldl' :: (b -> a -> b) -> b -> t a -> b
    foldr1 :: (a -> a -> a) -> t a -> a
    foldl1 :: (a -> a -> a) -> t a -> a
    toList :: t a -> [a]
    null :: t a -> Bool
    length :: t a -> Int
    elem :: Eq a => a -> t a -> Bool
    maximum :: forall a . Ord a => t a -> a
    minimum :: forall a . Ord a => t a -> a
    sum :: Num a => t a -> a
    product :: Num a => t a -> a
```

References : @@@

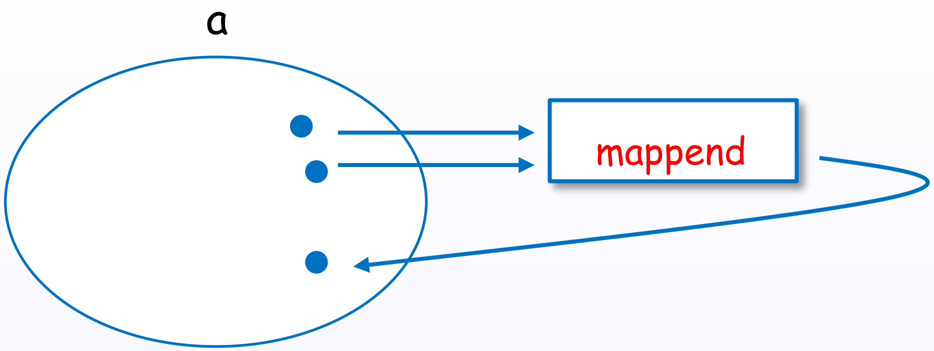
Traversable class

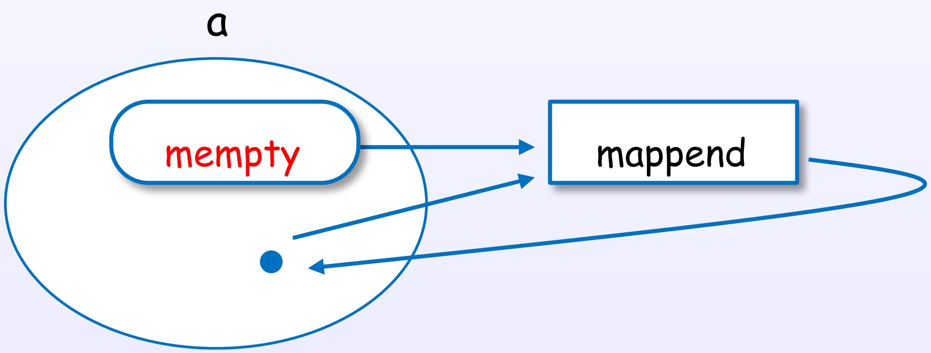


```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    sequenceA :: Applicative f => t (f a) -> f (t a)
    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    sequence :: Monad m => t (m a) -> m (t a)
```

References : @@@

Monoid class



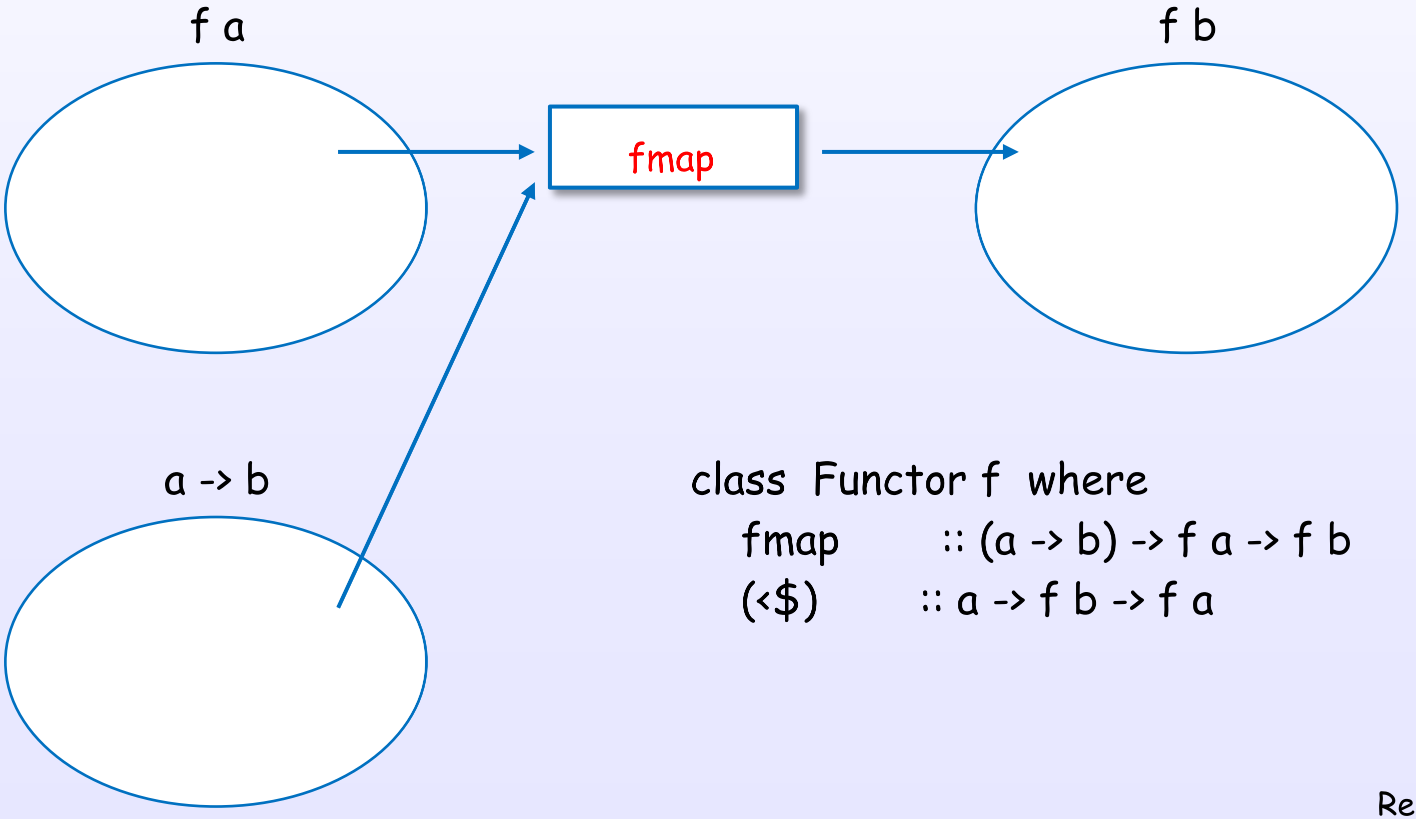


```
class Monoid a where
        mempty  :: a
        mappend :: a -> a -> a
        mconcat :: [a] -> a
```

References : @@@

Functor class

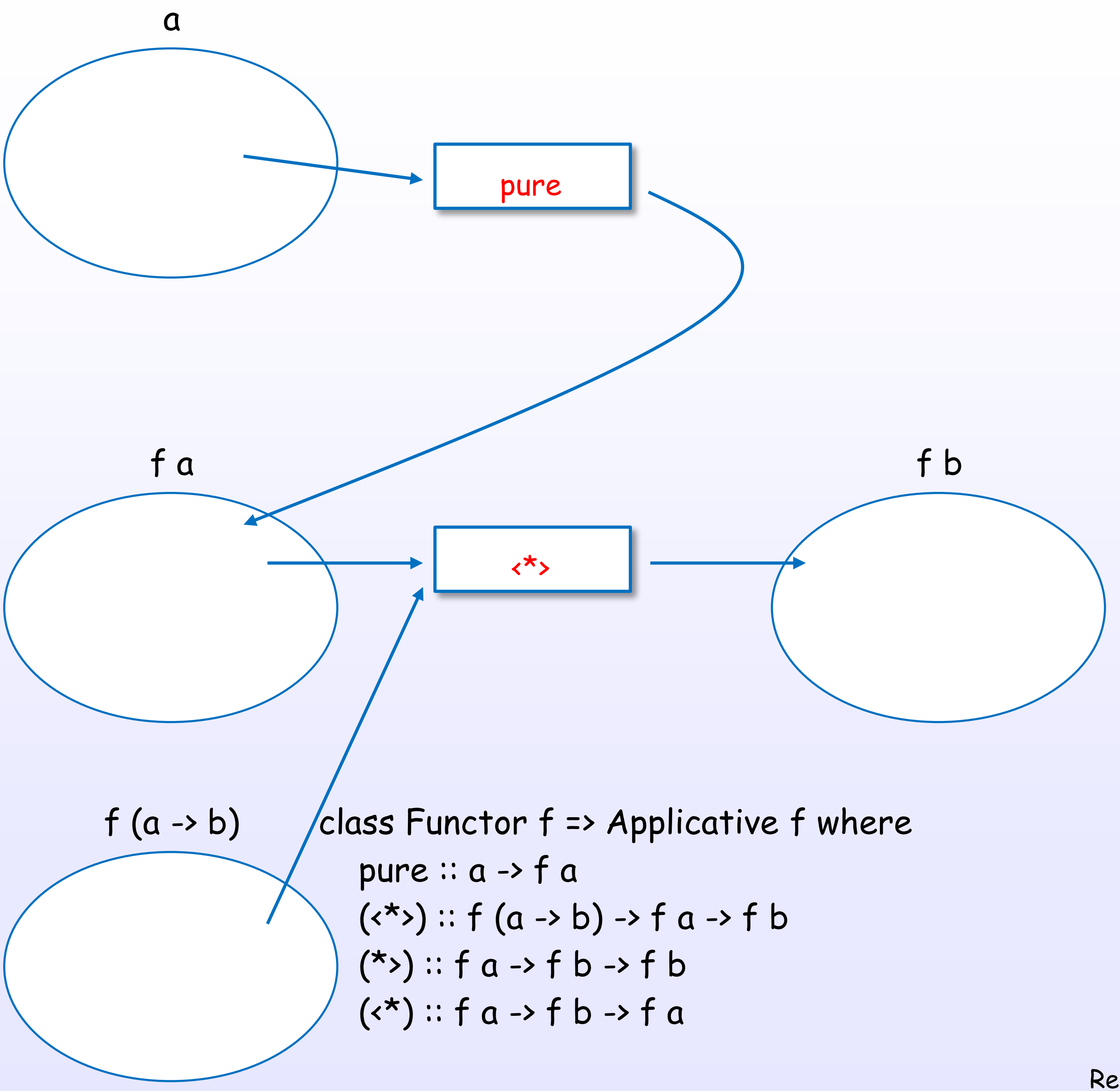
引数の上下の順番を合わせるか？



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

References : @@@

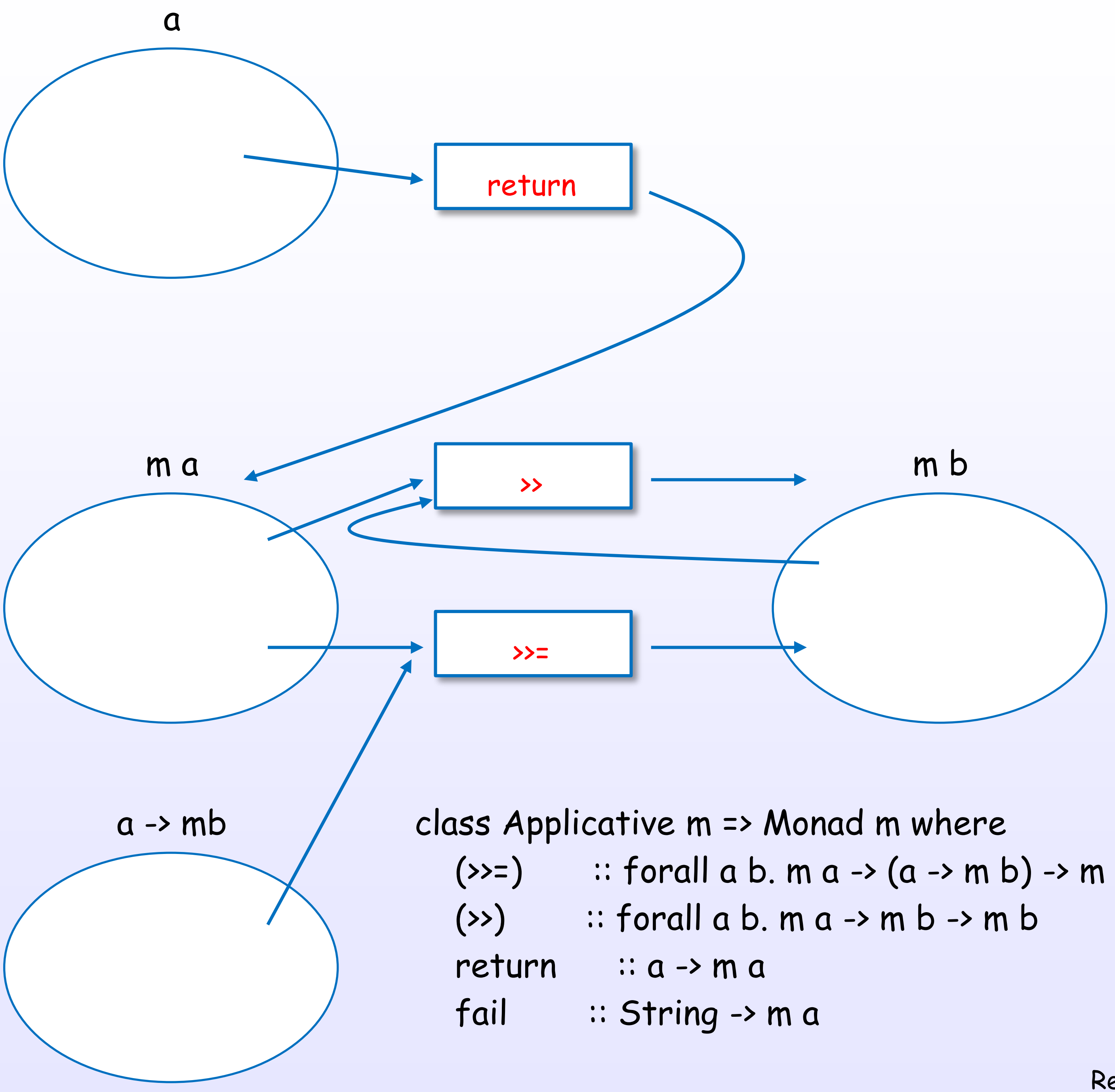
Applicative class



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

References : @@@

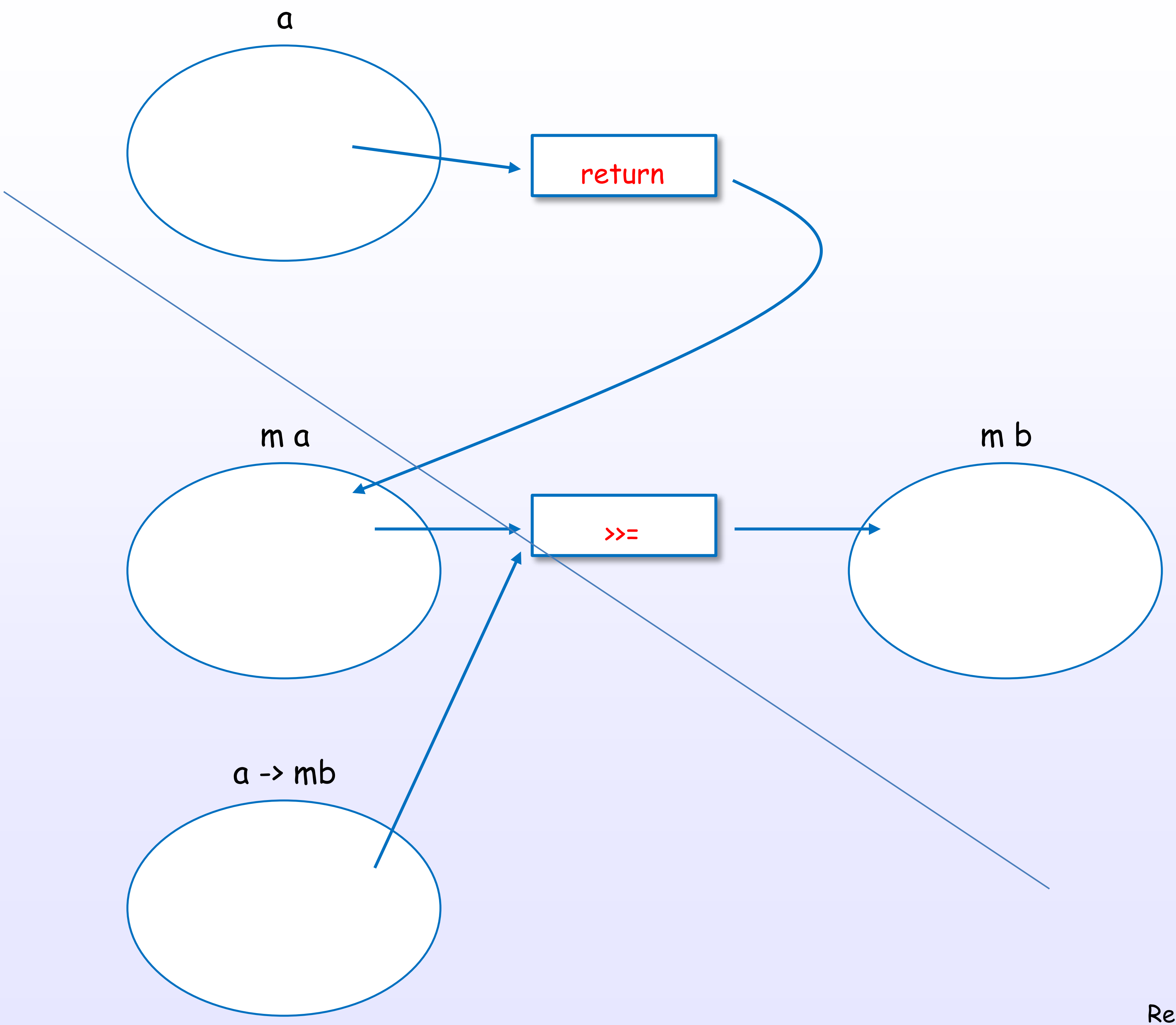
Monad class



```
class Applicative m => Monad m where
   (>>=)      :: forall a b. m a -> (a -> m b) -> m b
   (>>)       :: forall a b. m a -> m b -> m b
   return     :: a -> m a
   fail       :: String -> m a
```

References : @@@

Monad class



References : @@@

References

[B1] Learn You a Haskell for Great Good!
http://learnyouahaskell.com/

[B2] Thinking Functionally with Haskell (IFPH 3rd edition)
http://www.cs.ox.ac.uk/publications/books/functional/

[B3] Programming in Haskell
https://www.cs.nott.ac.uk/~gmh/book.html

[B4] Types and Programming Languages (TAPL)
https://mitpress.mit.edu/books/types-and-programming-languages

References

[D1] CIS 194: Introduction to Haskell
http://www.seas.upenn.edu/~cis194/lectures.html

[D2] Type Systems
http://dev.stephendiehl.com/fun/004_type_systems.html

[D3] Typeclassopedia
http://www.cs.tufts.edu/comp/150FP/archive/brent-yorgey/tc.pdf
https://wiki.haskell.org/Typeclassopedia

References

[S1] Hoogle
https://www.haskell.org/hoogle

References

[H1] Haskell 2010 Language Report
https://www.haskell.org/definition/haskell2010.pdf

[H2] The Glorious Glasgow Haskell Compilation System (GHC user's guide)
https://downloads.haskell.org/~ghc/latest/docs/users_guide.pdf