Software System Design and Implementation

Functors, Applicative and Monads

Gabriele Keller

The University of New South Wales
School of Computer Science and Engineering
Sydney, Australia

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

Data constructors revisited

```
data Point = Point Float Float
```

```
Point :: Float -> Float -> Point

Point 0.0 1.25 :: Point
```

Data constructors revisited

```
data Shape
   = Circle  Point Float
   | Path    [Point]
```

```
Circle :: Point   -> Float -> Shape
Path   :: [Point]          -> Shape
```

Data constructors revisited

```
data Tree a
   = Leaf
   | Node a (Tree a) (Tree a)
```

```
Leaf ::                                Tree a
Node :: a -> (Tree a) -> (Tree a) -> (Tree a)
```

Data constructors revisited

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

```
Left  :: a -> Either a b
Right :: b -> Either a b
```

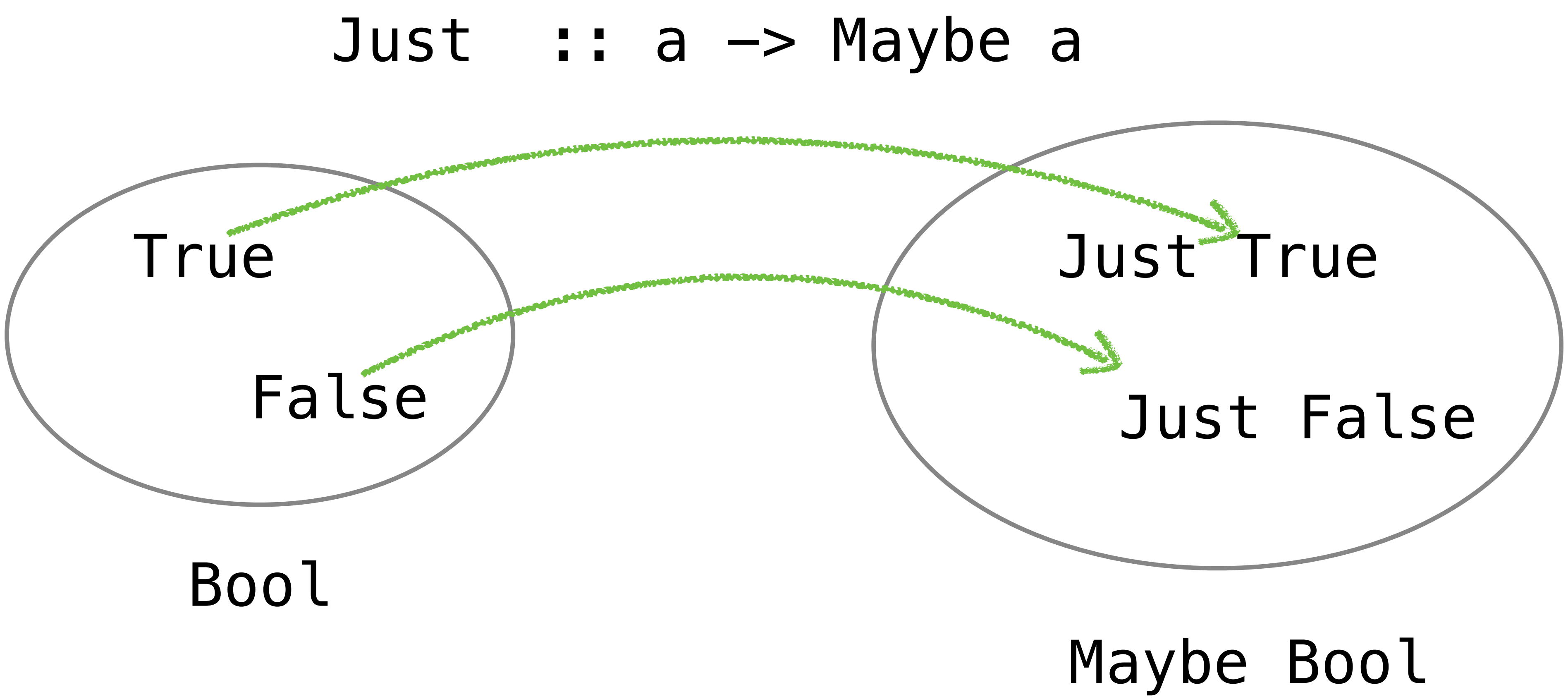
Data constructors revisited

```
data Maybe a
   = Nothing
   | Just a
```

```
Nothing ::      Maybe a
Just    :: a -> Maybe a
```

Type Constructors

- **Data constructors** map values to values:

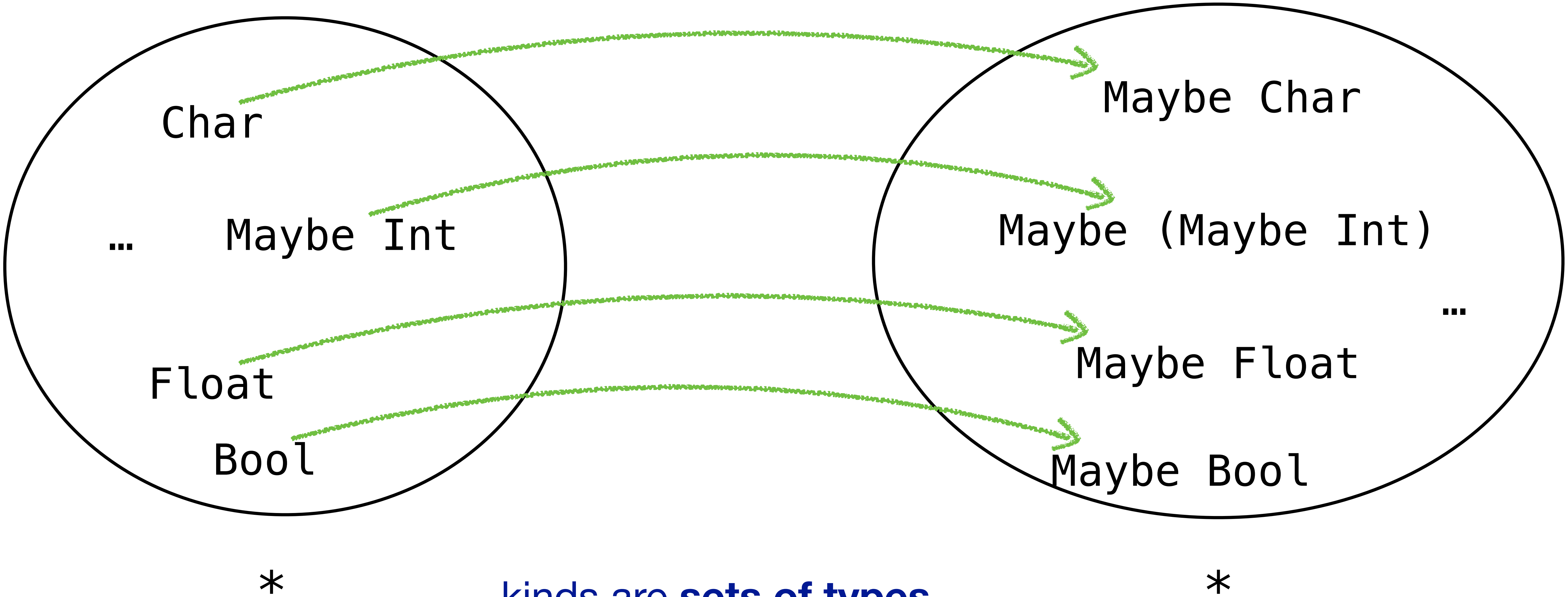


types a **sets of values**

Type Constructors

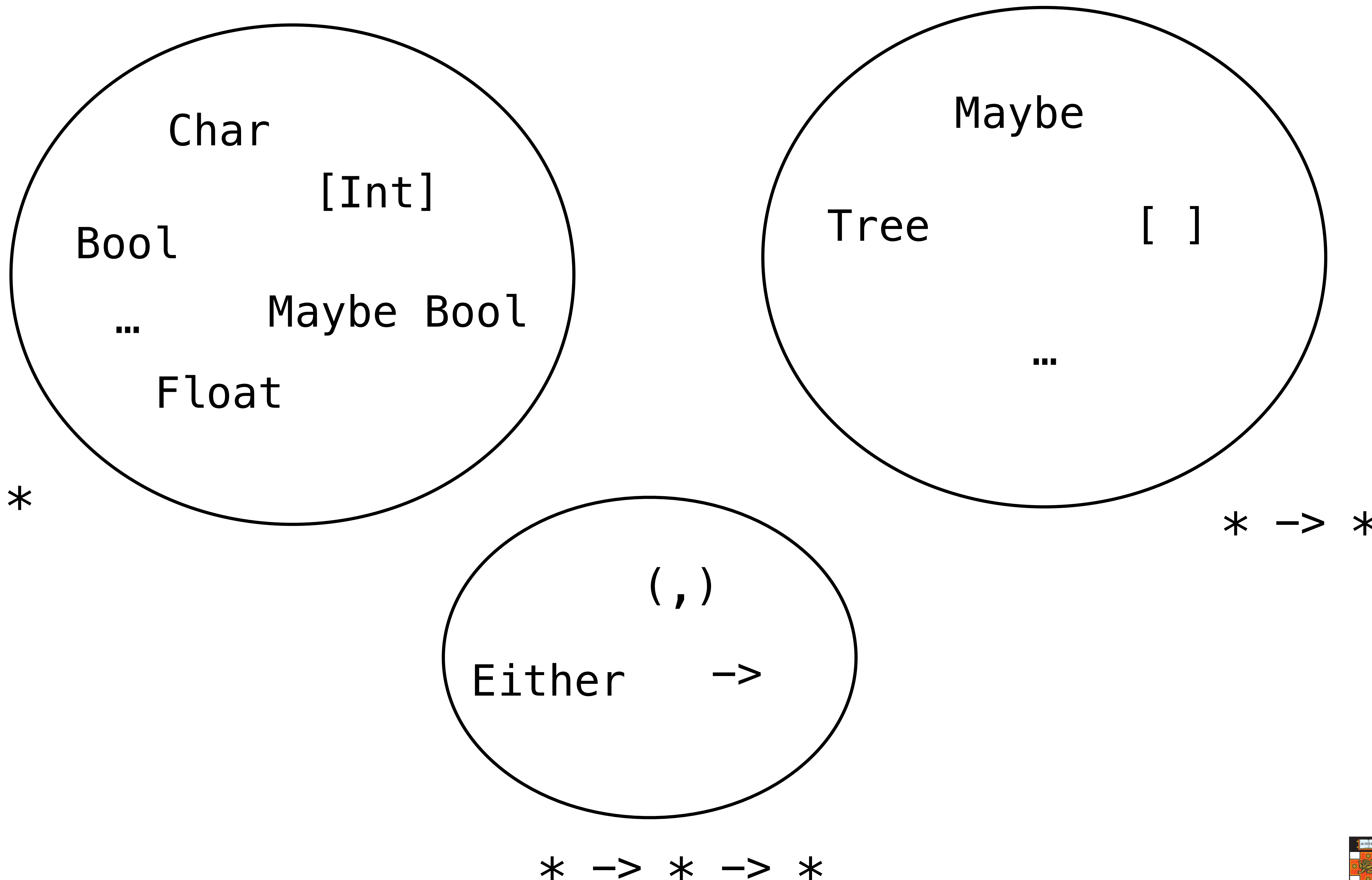
- **Type constructor** map types to type:

`Maybe :: * -> *`



kinds are **sets of types**

Kinds



Generalising map

- map on lists:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs)  = f x : map f xs
```

- map for other unary type constructors:

```
treeMap :: (a -> b) -> Tree a -> Tree b
treeMap f Leaf = Leaf
treeMap f (Node x leftSubtree rightSubtree)
  = Node f x (treeMap f leftSubtree)
             (treeMap f rightSubtree)
```

Generalising map

- map on the Maybe type:

```
maybeMap :: (a -> b) -> Maybe a -> Maybe b
maybeMap f Nothing  = Nothing
maybeMap f (Just x) = Just (f x)
```

```
fmap :: (a -> b) ->  f a -> f b
```

Functors

- We have seen how type classes can be used to group types according to the operations supported on their values:

a :: *

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

instance Eq Bool where
  (==) True   True  = True
  (==) False  False = True
  (==) _      _     = False
  (/=) b1     b2    = not (b1 == b2)
```

Functors

- We can also use type classes to group type constructors:

f :: * -> *

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


instance Functor Tree where
   fmap f Leaf   =  Leaf
   fmap f (Node a t1 t2)
      = Node (f a) (fmap f t1) (fmap f t2)
```

What properties should map have?

- Should leave the structure intact:

```
      fmap id xs  == xs

fmap (f . g) xs  == ((fmap f) . (fmap g)) xs
```

- These properties are not enforced by the compiler
  - it's the programmers responsibility to ensure
  - these are quickcheckable properties, but proofs are often straight forward
  - these abstractions are very useful to understand code

Applicative

- Applicative are functors with two additional operations:

```
class Functor f => Applicative f where

   pure  :: a -> f a
  (<*>)  :: f (a -> b) -> f a -> f b
```

Applicative

- Properties

```
              pure id <*> v == v

pure (.) <*> u <*> v <*> w == u <*> (v <*> w)

          pure f <*> pure x == pure (f x)

             u <*> pure y == pure ($ y) <*> u
```

Monads

- Monads

```
class Applicative m => Monad m where

  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Monads

- Properties

```
     return a >>= k  ==  k a

       m >>= return  ==  m

m >>= (\x -> k x >>= h)  ==  (m >>= k) >>= h
```

Monads

- Do-notation:

```
incMaybe :: Num a => Maybe a -> Maybe a
incMaybe (Just x) = Just (x +1)
incMaybe _        = Nothing


incM mx
  = mx >>=  \x ->
    return (x + 1)

addM mx my
  = mx >>= \x ->
    my >>= \y ->
    return (x + y)

addM mx my = do
  x <- mx
  y <- my
  return (x + y)
```