↙ Type (Constructor) Function :: Type → Type → Type → Type
(or so-called polymorphic Type)

$$\text{Type Transform } f \underbrace{a\ b}_{} = (a \to b) \to (f\ a \to f\ b)$$

← user specify

$(f, a, b) \to$ Type

Can still remain polymorphic by partial application:

transformList :: forall a, b. Transform List a b ← Type Function :: $* \to * \to *$

("factory pattern")
↙ Higher-order Type Function :: Type → Type Function

$$\text{Type Transform } f = \underbrace{\text{forall } a, b.}_{} \ \underbrace{(a \to b)}_{} \to (f\ a \to f\ b)$$

$f \to ((a, b) \to \text{Type})$ inferred by compiler from the given function's type.

this definition utilizes <u>currying of Type Functions</u>.

transformList :: Transform List

↖ the abstraction level is restricted by the Type Function i.e. Transform

user is not allowed to evaluate $(* \to * \to *)$ by substituting Types.

Kind System (to organize Types and Type Functions)

- 'Type', the kind of Types
- '$k_1 \to k_2$', Arrow kinds (the kind of Type Functions)
  (->) :: kind -> kind -> kind (a kind (constructor) Function)
- '#k', Row kinds (a record of Types of kind 'k')
  unordered map
  # :: kind -> kind
- User-defined kinds

example : Effect :: kind (befor 0.12)

- foreign import kind Effect
- foreign import data Eff :: # Effect -> Type -> Type
- foreign import data Console :: Effect
- foreign import data Random :: Effect
- ~~~~~~~~ HTTP :: Effect

need to move to 'Type' kind to operate with the Type System.

kind = { Type, Arrow, Row, Effect }

Effect = { Console, Random, HTTP, ... }

after 0.12   Effect :: Type -> Type   ~ IO in Haskell

```
class Monad m <= MonadEffect m where
    liftEffect :: forall a. Effect a -> m a
instance MonadEffect Effect where
    liftEffect = identity (:: Effect a -> Effect a)
```