Functor Lifting (2B)

Young Won Lim
7/21/18

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice.

Based on

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#the-functor-typeclass

http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Haskell in 5 steps
https://wiki.haskell.org/Haskell_in_5_steps

Lifting

Lifting is a concept which allows you

to transform a **function** into a corresponding **function**

within another (usually more general) setting.

https://wiki.haskell.org/Lifting

fmap : lifting operation

fn :: a -> b

fmap fn :: f a -> f b

fmap :: Functor f => (a -> b) -> f a -> f b

fmap :: Functor f => (a -> b) -> (f a -> f b)

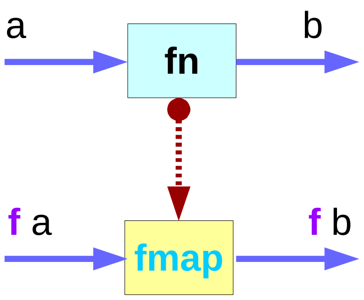
notice that fmap is a **lifting operation**

fmap transforms a function **fn :: a -> b**

between simple types **a** and **b**

into a function **fmap fn :: f a -> f b**

between pairs of these types **f a** and **f b**



function fmap
function fn
type constructor f

https://wiki.haskell.org/Lifting

Functor Pair

consider a **Pair** functor

instances not allowed for

```
type Pair a = (a, a)
```

define a new datatype by using **data Pair a**

```
data Pair a = Pair a a      deriving Show
instance Functor Pair where
    fmap fn (Pair x y) = Pair (fn x) (fn y)
```

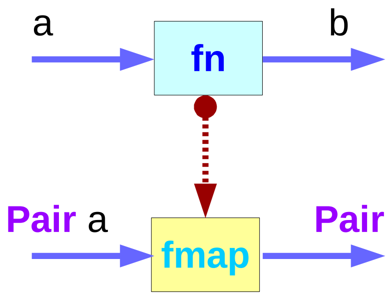
Pair : type constructor

Pair : data constructor

https://wiki.haskell.org/Lifting

Functor Lifting

```
data Pair a = Pair a a     deriving Show
instance Functor Pair where
   fmap fn (Pair x y) = Pair (fn x) (fn y)
```



a

fn

b

Pair a

fmap

Pair b

function fmap
function fn
type constructor Pair

```
lift :: (a -> b) -> Pair a -> Pair b
lift = fmap
```

A **functor** can only lift functions of exactly one variable, (a->b)
but we want to lift other functions, too: (a->b->c), (a->b->c->d),

```
lift0 :: a -> Pair a
lift0 x = Pair x x
```

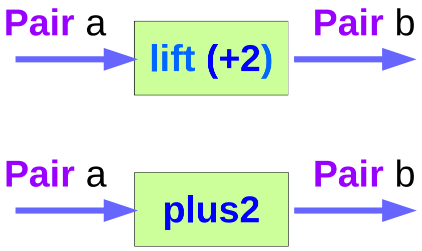
```
lift2 :: (a -> b -> r) -> (Pair a -> Pair b -> Pair r)
lift2 fn (Pair x1 x2) (Pair y1 y2) = Pair (fn x1 y1) (fn x2 y2)
```

https://wiki.haskell.org/Lifting

Functor Lifting Example

```
data Pair a = Pair a a      deriving Show
instance Functor Pair where
    fmap fn (Pair x y) = Pair (fn x) (fn y)
```

```
plus2 :: Pair Int -> Pair Int
plus2 = lift (+2)

plus2 (Pair 2 3) ---> Pair 4 5
```



```
plus :: Pair Int -> Pair Int -> Pair Int
plus = lift2 (+)

plus (Pair 1 2) (Pair 3 4) --->
Pair 4 6
```

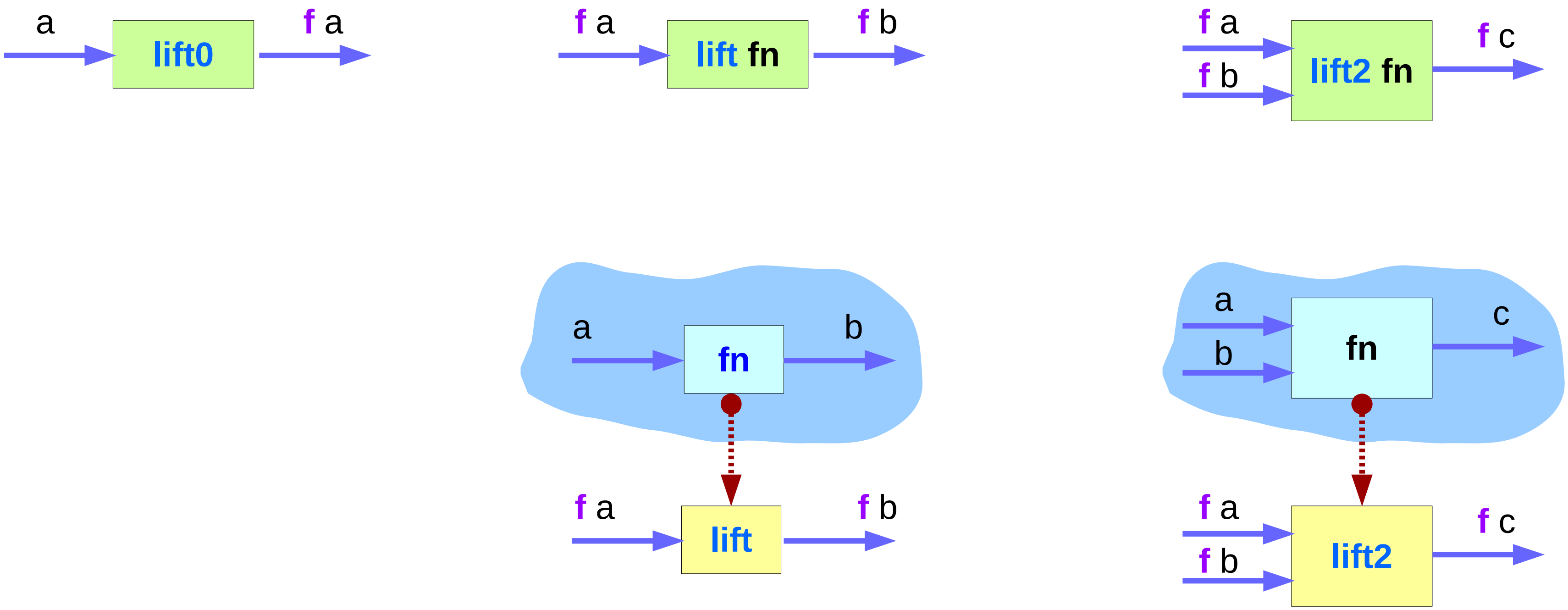
Not all functions between **Pair a** and **Pair b** can be constructed as a lifted function

\(x, _) -> (x, 0)

can't be a lifting function

https://wiki.haskell.org/Lifting

Functor Lifting



https://wiki.haskell.org/Lifting

Liftable Lifting

```
class Functor f => Liftable f where
   zipL :: f a -> f b -> f (a, b)
   zeroL :: f ()
```

```
liftL :: Liftable f => (a -> b) -> (f a -> f b)
liftL = fmap
```

```
liftL2 :: Liftable f => (a -> b -> c) -> (f a -> f b -> f c)
liftL2 fn x y = fmap (uncurry fn) $ zipL x y
```

```
fn :: a -> b -> c
x :: f a
y :: f b
```

```
liftL3 :: Liftable f => (a -> b -> c -> d) -> (f a -> f b -> f c -> f d)
liftL3 fn x y z = fmap (uncurry . uncurry $ fn) $ zipL (zipL x y) z
```

```
fn :: a -> b -> c -> d
x :: f a
y :: f b
z :: f c
```

```
liftL0 :: Liftable f => a -> f a
liftL0 x = fmap (const x) zeroL
```

https://wiki.haskell.org/Lifting

Liftable Lifting – **liftL2**

```
liftL2 :: Liftable f => (a -> b -> c) -> (f a -> f b -> f c)
liftL2 fn x y = fmap (uncurry fn) $ zipL x y
```

```
x :: f a
y :: f b
zipL :: f a -> f b -> f (a, b)
```

```
zipL x y :: zipL f a f b
zipL x y :: f (a, b)
```

```
fn :: a -> b -> c
```

```
uncurry fn :: (a, b) -> c
```

```
fmap (uncurry fn) $ zipL x y :: fmap (uncurry fn) f (a, b)
fmap (uncurry fn) $ zipL x y :: fmap ((a, b) -> c) f (a, b)
fmap (uncurry fn) $ zipL x y :: f c
```

https://wiki.haskell.org/Lifting

Liftable Lifting – **liftL3**

```
liftL3 :: Liftable f => (a -> b -> c -> d) -> (f a -> f b -> f c -> f d)
liftL3 fn x y z = fmap (uncurry . uncurry $ fn) $ zipL (zipL x y) z
```

```
x :: f a
y :: f b
zipL :: f a -> f b -> f (a, b)
```

```
z :: f c
```

```
zipL x y :: zipL f a f b
zipL x y :: f (a, b)
```

```
zipL (zipL x y) z :: zipL f (a, b) f c
zipL (zipL x y) z :: f (a, b, c)
```

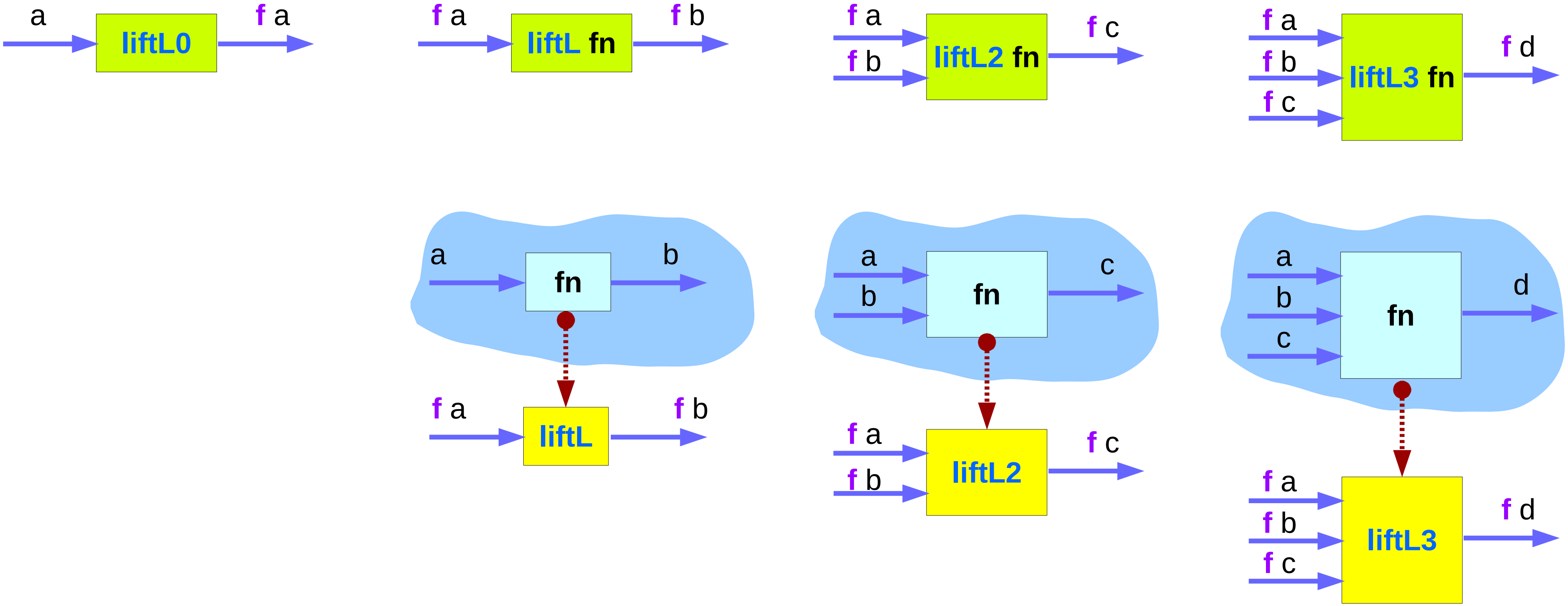
```
fn :: a -> b -> c -> d
```

```
uncurry $ fn :: (a, b) -> c -> d
uncurry . uncurry  $ fn :: (a, b, c) -> d
```

```
fmap (uncurry . uncurry $ fn) $ zipL (zipL x y) z :: fmap (uncurry . uncurry $ fn) f (a, b, c)
fmap (uncurry . uncurry $ fn) $ zipL (zipL x y) z :: fmap ((a, b, c) -> d) f (a, b, c)
fmap (uncurry . uncurry $ fn) $ zipL (zipL x y) z :: f d
```

https://wiki.haskell.org/Lifting

Liftable Lifting



https://wiki.haskell.org/Lifting

Applicative Lifting **appL**

```
class Functor f => Liftable f where
   zipL :: f a -> f b -> f (a, b)
   zeroL :: f ()
```

```
liftL2 :: Liftable f => (a -> b -> c) -> (f a -> f b -> f c)
```

```
fn :: a -> b -> c    ➡   ($) :: (a -> b) -> a -> b
x :: f a             ➡   ff :: f (a -> b)
y :: f b             ➡   fx :: f a
```

```
f $ x = f x
```

```
liftL2 ($) :: Liftable f => ((a -> b) -> a -> b) -> (f (a -> b) -> f a -> f b)
```

```
       ($) ff fx ::  ((a -> b) -> a -> b) -> f (a -> b) -> f a
liftL2 ($) ff fx :: f b
```

```
ff :: f (a -> b)
fx :: f a
```

```
appL ff fx = liftL2  ff fx
```

```
appL :: Liftable f => f (a -> b) -> f a -> f b
appL = liftL2 ($)
```

https://wiki.haskell.org/Lifting

Applicative Lifting

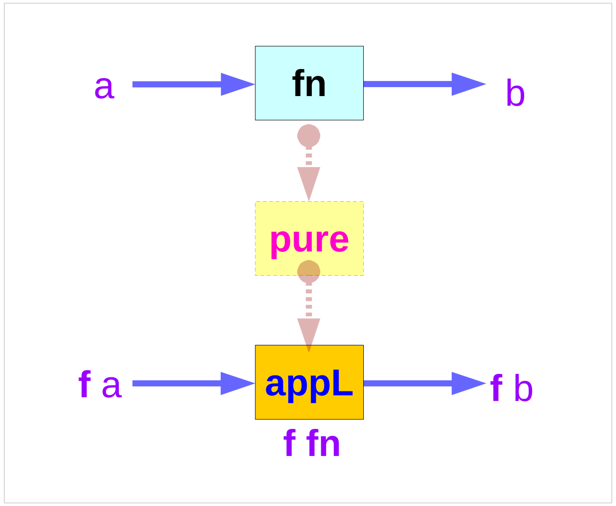
```
class Functor f => Liftable f where
    zipL :: f a -> f b -> f (a, b)
    zeroL :: f ()
```

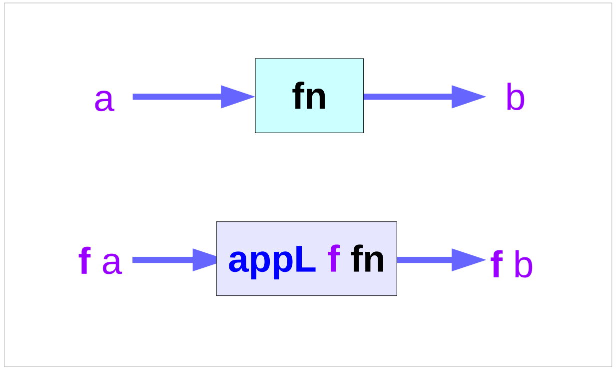
```
appL :: Liftable f => f (a -> b) -> f a -> f b
appL = liftL2 ($)
```

```
appL ff x = liftL2 ($) ff fx
```

```
($) :: (a -> b) -> a -> b
ff :: f (a -> b)                  fn :: (a -> b)
fx :: f a                         x :: a
```

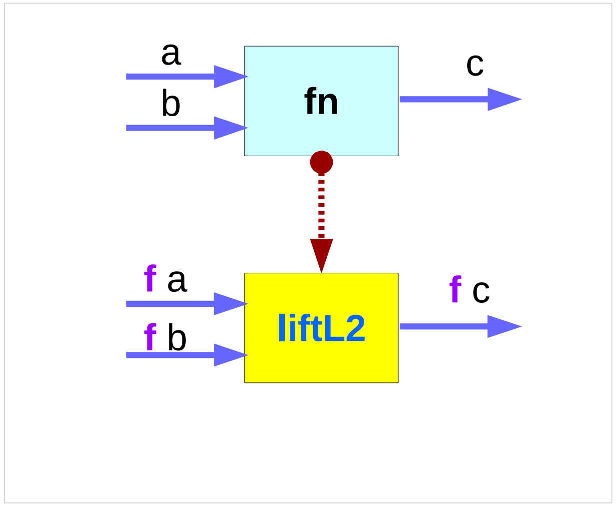
```
liftL2 :: Liftable f => ((a -> b) -> a -> b) -> f (a -> b) -> f a -> f b
```

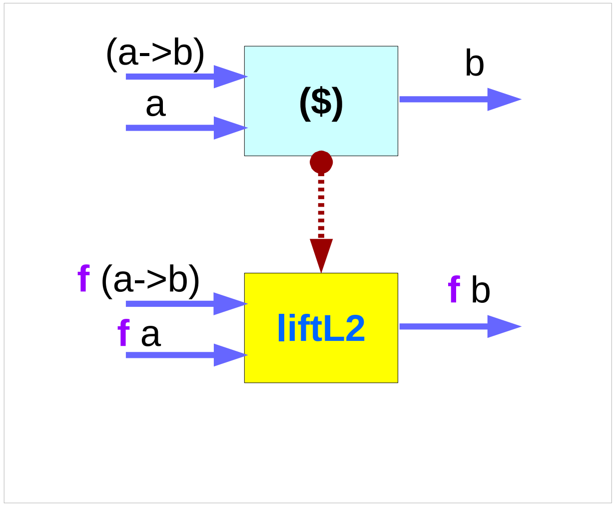




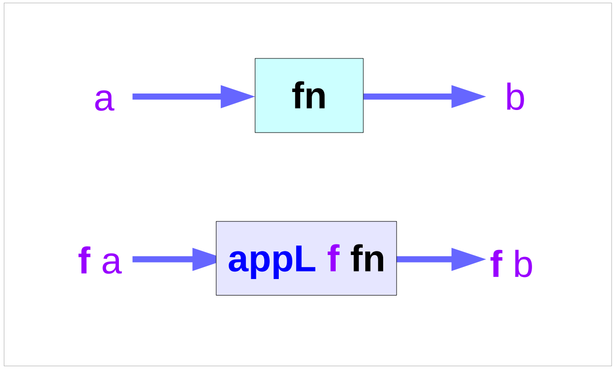
https://wiki.haskell.org/Lifting

Applicative Lifting using **liftL2**



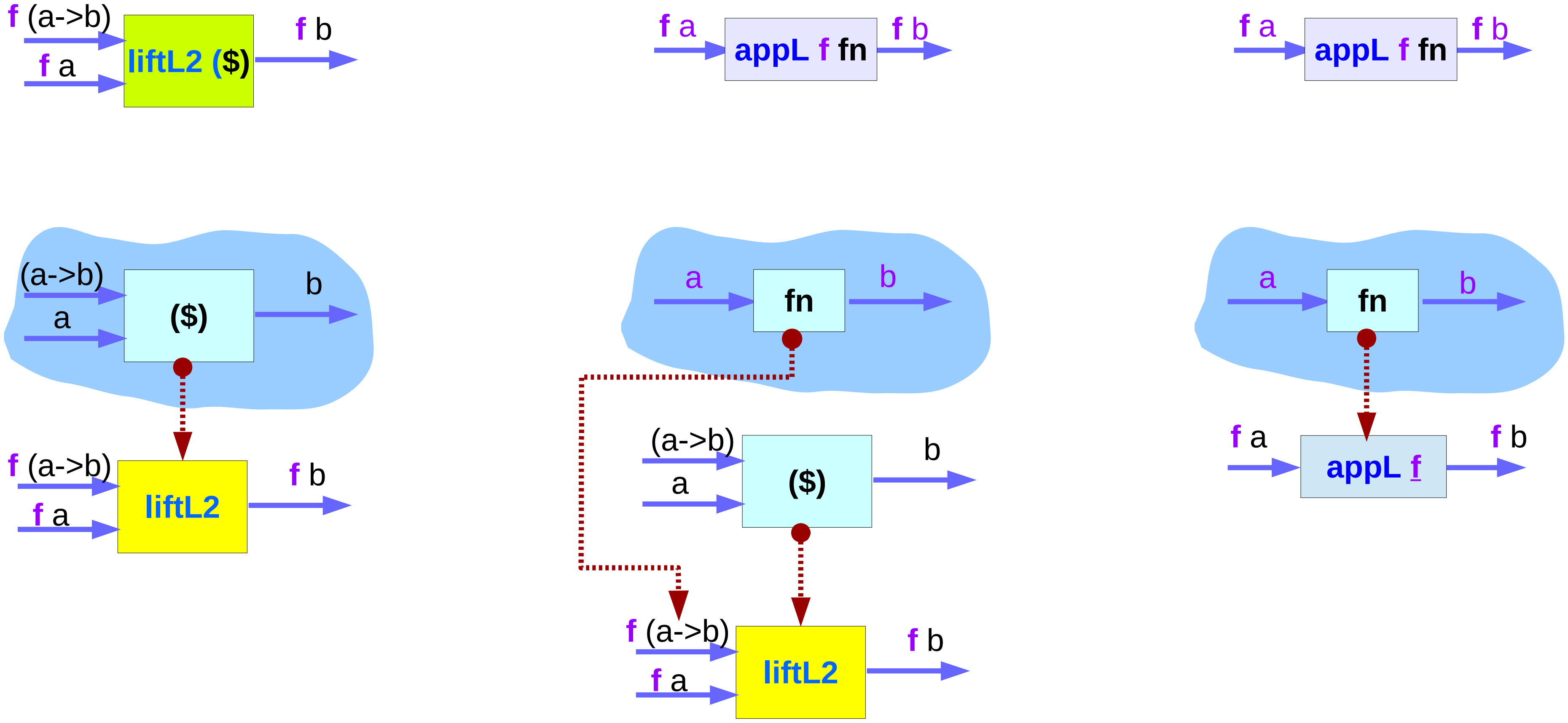


```
class Functor f => Liftable f where
    zipL :: f a -> f b -> f (a, b)
    zeroL :: f ()

liftL2 :: Liftable f => (a -> b -> c) -> (f a -> f b -> f c)
liftL2 fn x y = fmap (uncurry fn) $ zipL x y

appL :: Liftable f => f (a -> b) -> f a -> f b
appL = liftL2 ($)
```



http://learnyouahaskell.com/functors-applicative-functors-and-monoids

Applicative Lifting

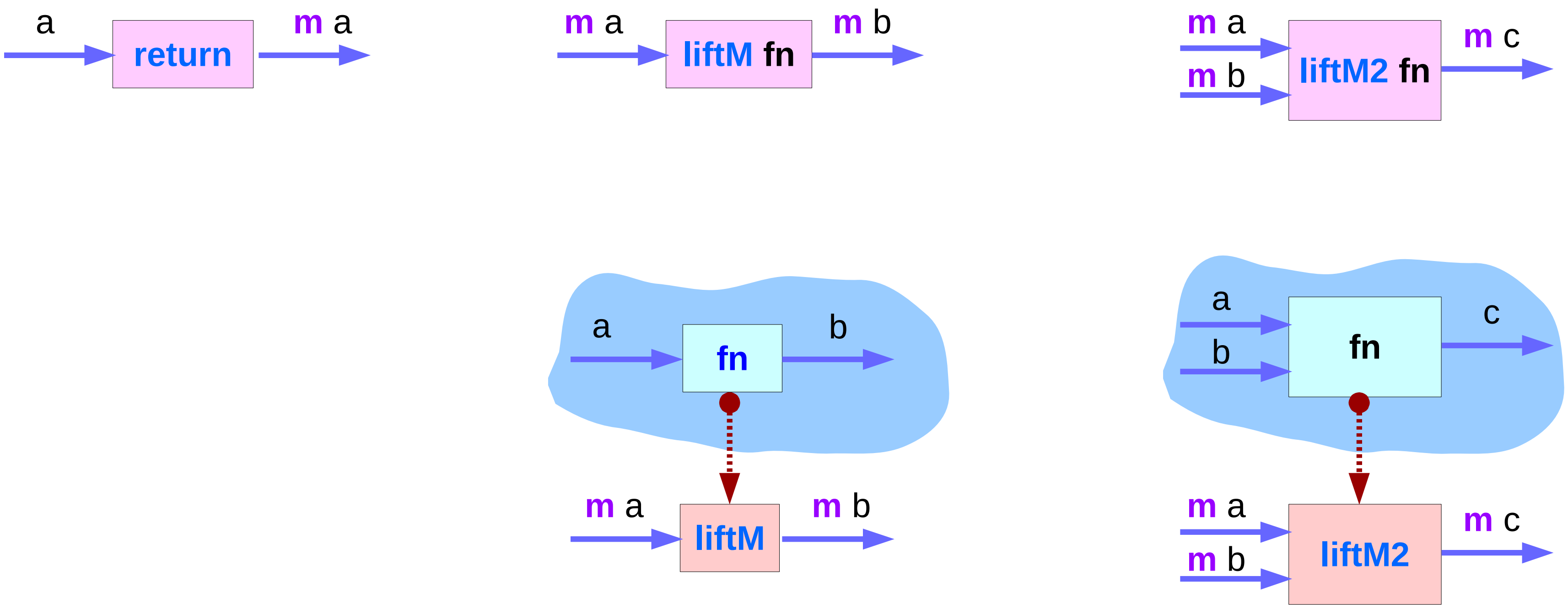


Monad Lifting

```
return   :: (Monad m) => a1 -> m a1
liftM    :: (Monad m) => (a1 -> r) -> m a1 -> m r
liftM2   :: (Monad m) => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
```

https://wiki.haskell.org/Lifting

Monad Lifting



```
return   :: (Monad m) => a -> m b
liftM    :: (Monad m) => (a -> b) -> m a -> m b
liftM2   :: (Monad m) => (a -> b -> c) -> m a -> m b -> m c
```

https://wiki.haskell.org/Lifting

Monad Lifting Examples

```
plus :: [Int] -> [Int] -> [Int]
plus = liftM2 (+)


plus [1,2,3] [3,6,9] --->
[4,7,10, 5,8,11, 6,9,12]

plus [1..] []        --->
_|_        (i.e., keeps on calculating forever)

plus [] [1..]        --->
[]
```

example the list monad (**MonadList**).
It performs a nondeterministic calculation,
returning all possible results.

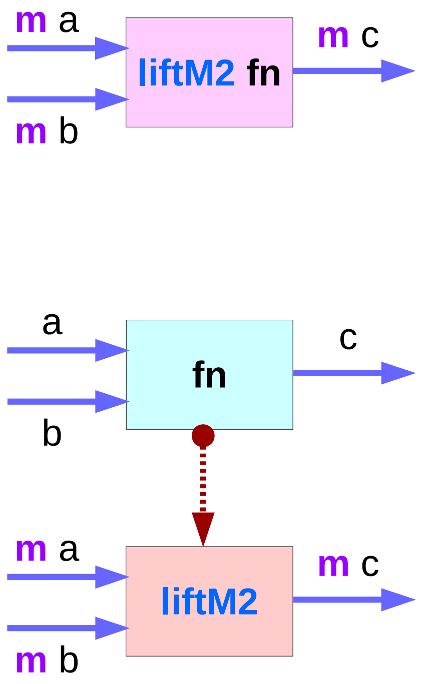
liftM2 just turns a deterministic function into a nondeterministic one:

https://wiki.haskell.org/Lifting

Using **liftM2** of Monad

Every **Monad** can be made
an **instance** of **Liftable**

```
{-# OPTIONS -fglasgow-exts #-}
{-# LANGUAGE AllowUndecidableInstances #-}
import Control.Monad

instance (Functor m, Monad m) => Liftable m where
    zipL  = liftM2 (\x y -> (x,y))
    zeroL = return ()
```



mf :: a b -> (a,b)
mf = (\x y -> (x,y))

https://wiki.haskell.org/Lifting

Instance of Liftable using **liftM2**

```
class Functor f => Liftable f where
    zipL :: f a -> f b -> f (a, b)
    zeroL :: f ()
```

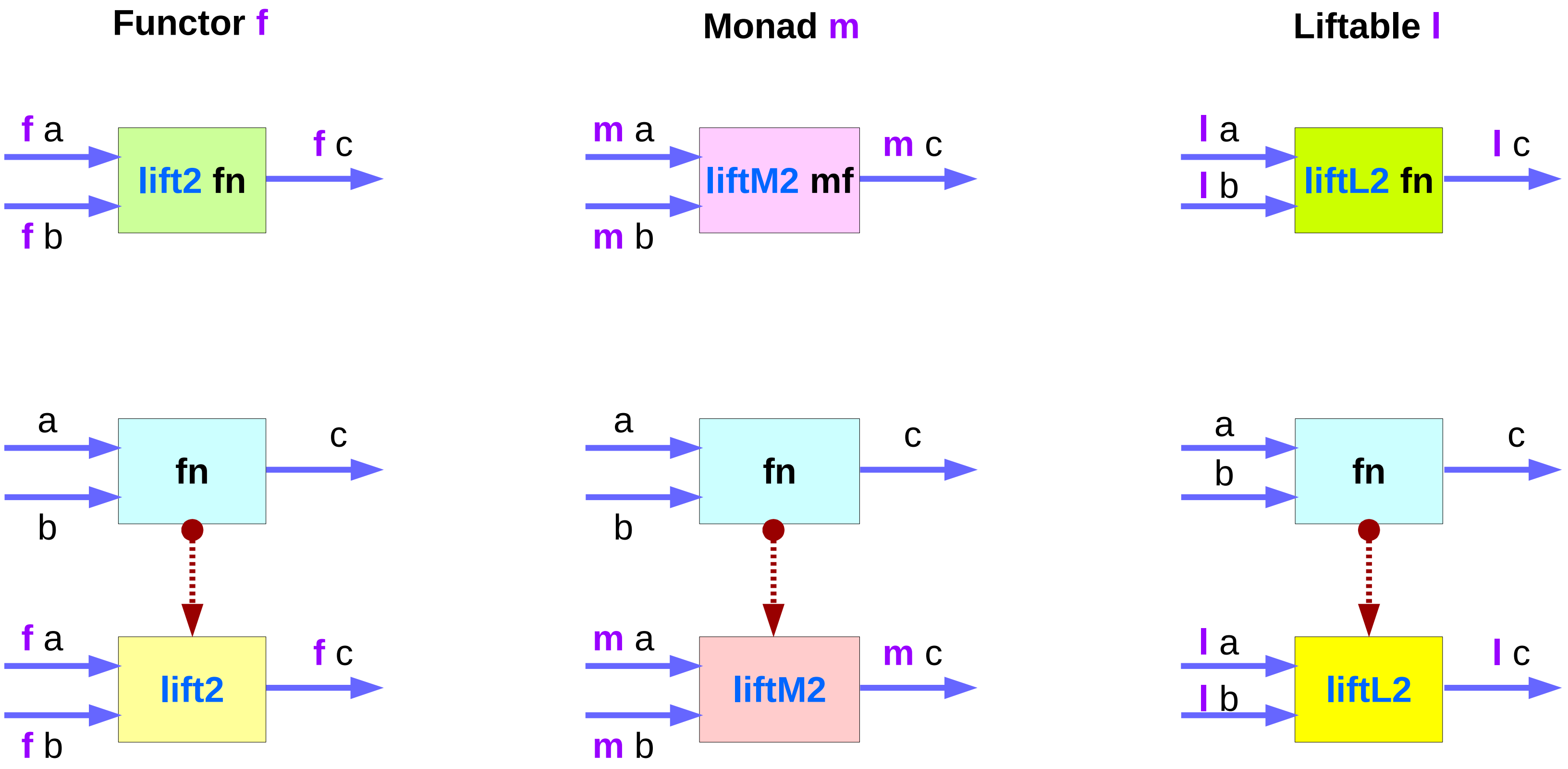
```
liftL2 :: Liftable f => (a -> b -> c) -> (f a -> f b -> f c)
liftL2 fn x y = fmap (uncurry fn) $ zipL x y
```

```
liftM2    :: (Monad m) => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
liftM2    :: (Monad m) => (a   -> b  -> c) -> m a   -> m  b  -> m c

          (\x y -> (x,y)) :: a -> b -> c                    x :: a,      y :: b,      (x,y) :: c
liftM2    (\x y -> (x,y)) :: m a -> m  b -> m c
liftM2    (\x y -> (x,y)) :: f   a -> f    b -> f   c
```

```
instance (Functor m, Monad m) => Liftable m where
    zipL  = liftM2 (\x y -> (x,y))
    zeroL = return ()
```

https://wiki.haskell.org/Lifting

Monad Lifting



https://wiki.haskell.org/Lifting

Typeclass Definitions of Functor, Applicative, and Monad

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

  (<$) :: a        -> f b -> f a
  (<$) = fmap . Const
```

```
class Functor f => Applicative f where
  pure  :: a -> f a
  infixl 4 <*>, *>, <*
  (<*>) :: f (a -> b) -> f a -> f b

  (*>) :: f a -> f b -> f b
  a1 *> a2 = (id <$ a1) <*> a2

  (<*) :: f a -> f b -> f a
  (<*) = liftA2 const
```

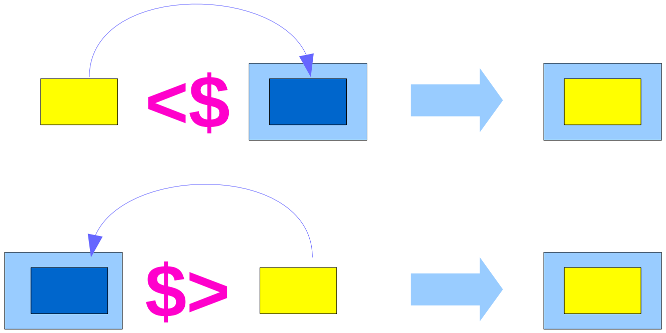
```
class Applicative m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  m >> n = m >>= \_ -> n

  fail   :: String -> m a
```

https://wiki.haskell.org/Typeclassopedia

Functor <$> related operators

Functor map <$>

```
(<$>)  :: Functor f => (a -> b) -> f a   -> f b
(<$)   :: Functor f => a         -> f b   -> f a
($>)   :: Functor f => f a       -> b     -> f b
```



replace b in f b

with a … f a

replace a in f a

with b … f b

The **<$>** operator is just a <u>synonym</u> for the **fmap** function in the Functor typeclass.

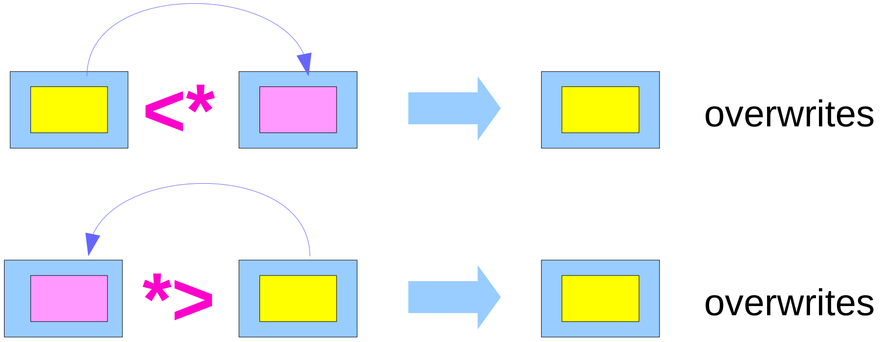
fmap generalizes **map** for **lists** to other data types : **Maybe**, **IO**, **Map**.

https://haskell-lang.org/tutorial/operators

Applicative <*> related operators

Applicative function application <*>

```
(<*>)   :: Applicative f => f (a -> b) -> f a -> f b
(*>)    :: Applicative f => f a        -> f b -> f b
(<*)    :: Applicative f => f a        -> f b -> f a
```



<*> is an operator that applies a wrapped function to a wrapped value.

<*> is a part of the **Applicative** typeclass,

<*>is very often used as follows

foo <$> bar <*> baz

faa <*> bar <*> baz

https://haskell-lang.org/tutorial/operators

Monadic binding / composition operators

```
(>>=)  :: Monad m => m a        -> (a -> m b)  -> m b
(=<<)  :: Monad m => (a -> m b) -> m a         -> m b
(>>)   :: Monad m => m a        -> m b         -> m b
(>=>)  :: Monad m => (a -> m b) -> (b -> m c)  -> (a -> m c)
(<=<)  :: Monad m => (b -> m c) -> (a -> m b)  -> (a -> m c)
```

https://haskell-lang.org/tutorial/operators

Monad Transformers

Using several monads at once
a function could use both **I/O** and **Maybe exception handling**

While a type like **IO (Maybe a)** would work just fine,
it would force us to **do pattern matching**
within **IO do-blocks** to extract values,
something that the **Maybe** monad was meant to spare us from.

monad transformers:
special types that allow us to roll two **monads**
into a single one that shares the behavior of both.

https://wiki.haskell.org/Lifting

Monad Transformer MaybeT

define a **monad transformer** that gives the **IO monad**
some characteristics of the **Maybe monad**;
we will call it **MaybeT**

monad transformers have a "**T**" appended
to the name of the **monad** whose characteristics they provide.

https://wiki.haskell.org/Lifting

Multi-level Monad Lifting

Lifting becomes especially interesting
when there are more levels you can lift between.

Control.Monad.Trans defines a class

```
class MonadTrans t where
    lift :: Monad m => m a -> t m a
```

-- lifts a value from the inner **monad m**
-- to the transformed **monad t m**
-- could be called **lift0**

https://wiki.haskell.org/Lifting

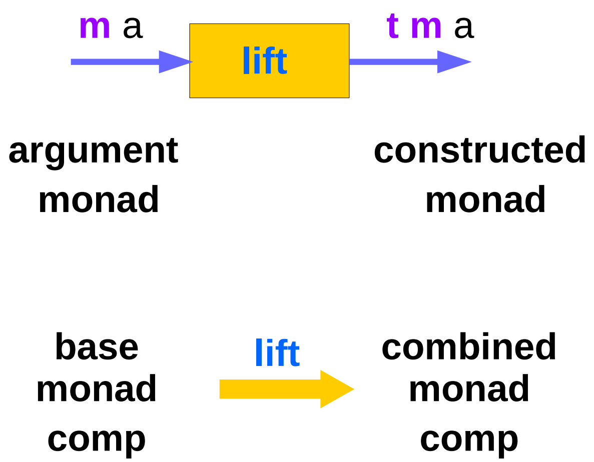
MonadTrans Lifting

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

the class of **monad transformers**

minimal complete definition : **lift** method

this method lifts a **computation**
from the **argument monad** **m a**
to the **constructed monad** **t m a**

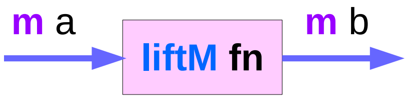


http://hackage.haskell.org/package/transformers-0.5.5.0/docs/Control-Monad-Trans-Class.html

liftM : lifting a function

liftM converts a plain **function** into one that acts within **m**.
By **lifting**, we refer to bringing something into something else
— in this case, a **function** into a **monad**.

liftM allows us to apply a plain **function** to a **monadic value**
without needing do-blocks or other such tricks:



bind notation

```
monadicValue >>=
  \x -> return (f x)
```

do notation

```
do x <- monadicValue
   return (f x)
```

liftM operation

```
liftM f monadicValue
```

https://en.wikibooks.org/wiki/Haskell/Monad_transformers

lift : lifting monad computations

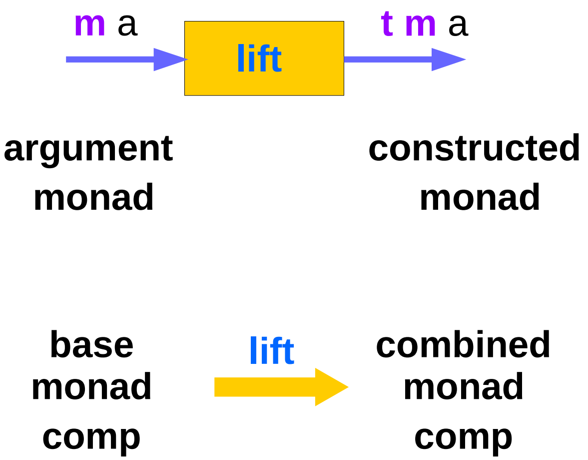
The **lift** function of the **MonadTrans** class
plays an analogous role of **liftM**
when working with **monad transformers**.

It brings (promotes) **base monad computations**
to the **combined monad computations**.

lift enables us to easily insert **base monad computations**
as part of a larger computation in the **combined monad**.

lift is the single method of the **MonadTrans** class,
found in **Control.Monad.Trans.Class**.

All **monad transformers** are **instances** of **MonadTrans**,
and so **lift** is available for them all.



https://en.wikibooks.org/wiki/Haskell/Monad_transformers

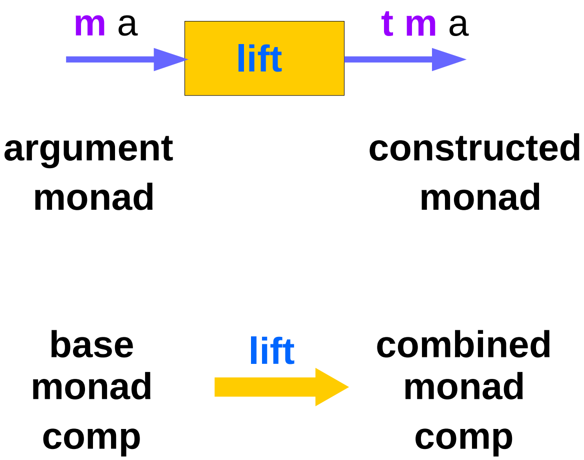
MonadTrans Lifting Laws

```
class MonadTrans t where
   lift :: Monad m => m a -> t m a
```

the class of **monad transformers**.

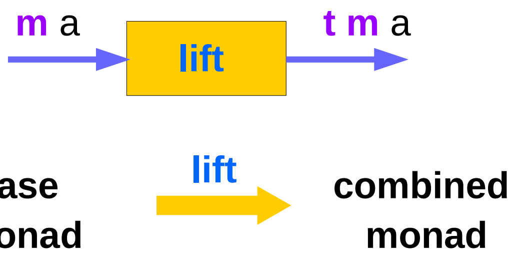
instances should satisfy the following **laws**,
which state that **lift** is a **monad transformation**:

```
lift . return = return

lift (m >>= f) = lift m >>= (lift . f)
```



http://hackage.haskell.org/package/transformers-0.5.5.0/docs/Control-Monad-Trans-Class.html

MonadIO Lifting



```
class (Monad m) => MonadIO m where
   liftIO :: IO a -> m a
```

```
class MonadTrans t where
   lift :: Monad m => m a -> t m a
```

There is a variant of lift specific to **IO** operations, called liftIO,

the single method of the **MonadIO** class in **Control.Monad.IO.Class**.

https://en.wikibooks.org/wiki/Haskell/Monad_transformers

MonadIO Lifting

```
class (Monad m) => MonadIO m where
   liftIO :: IO a -> m a
```

```
class MonadTrans t where
   lift :: Monad m => m a -> t m a
```

when multiple **transformers** are stacked
into a single **combined monad**.

In such cases, **IO** is always the innermost **monad**,
and so we typically need more than one **lift**
to bring **IO** values to the top of the stack.

liftIO is defined for the instances in a way
that allows us to bring an **IO** value from any depth
while writing the function a single time.

https://en.wikibooks.org/wiki/Haskell/Monad_transformers

MonadTrans Instance Example

```
instance MonadTrans MaybeT where
   lift m = MaybeT (liftM Just m)
```

```
class MonadTrans t where
   lift :: Monad m => m a -> t m a
```

Implementing the **MaybeT transformer**:

We begin with a **monadic value**
of the **base monad (Just m)**.
With **liftM** (**fmap** would have worked just as fine),
we slip the precursor monad (through the **Just** constructor) underneath,
so that we go from **m a** to **m (Maybe a)**).

Finally, we wrap things up with the **MaybeT constructor**.
Note that the **liftM** here works in the **base monad (m a)**,
just like the **do-block** wrapped by **MaybeT**
in the implementation of **(>>=)**
we saw early on was in the **base monad**.

https://en.wikibooks.org/wiki/Haskell/Monad_transformers

Arrow Lifting

Until now, we have only considered
lifting from functions to other functions.
John Hughes' arrows
are a generalization of computation
that aren't functions anymore.

An **arrow a b c** stands for a **computation**
which transforms **values** of **type b**
to **values** of **type c**.

The basic primitive **arr**, aka **pure**,

arr :: (Arrow a) => (b -> c) -> a b c

https://wiki.haskell.org/Lifting

References

[1] ftp://ftp.geoinfo.tuwien.ac.at/navratil/HaskellTutorial.pdf
[2] https://www.umiacs.umd.edu/~hal/docs/daume02yaht.pdf