Generic Matrix Multiplication

Sean Seefried

This talk is about generalising matrix multiplication to other data structures

Matrix multiplication

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

Dot products

Take two vectors of length n

$u = (u_1, \ldots, u_n)$ $v = (v_1, \ldots, v_n)$

Dot product is

$$u \cdot v = u_1v_1 + \ldots + u_nv_n$$

or

$$u \cdot v = \sum_{i=1}^{n} u_iv_i$$

Dot product for lists

```
dot :: Num a => [a] -> [a] -> a
dot xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

Dot product for lists

```
dot :: Num a => [a] -> [a] -> a
dot xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

Works for lists of different lengths because

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Vectors

```haskell
data Z
data S n

infixr 5 `Cons`
data Vec n a where
  Nil  :: Vec Z a
  Cons :: a -> Vec n a -> Vec (S n) a
```

zipWithV

```
zipWithV :: (a -> b -> c) -> Vec n a -> Vec n b -> Vec n c
zipWithV f Nil         Nil         = Nil
zipWithV f (Cons x xs) (Cons y ys) = f x y
                                     `Cons`
                                     zipWithV f xs ys
```

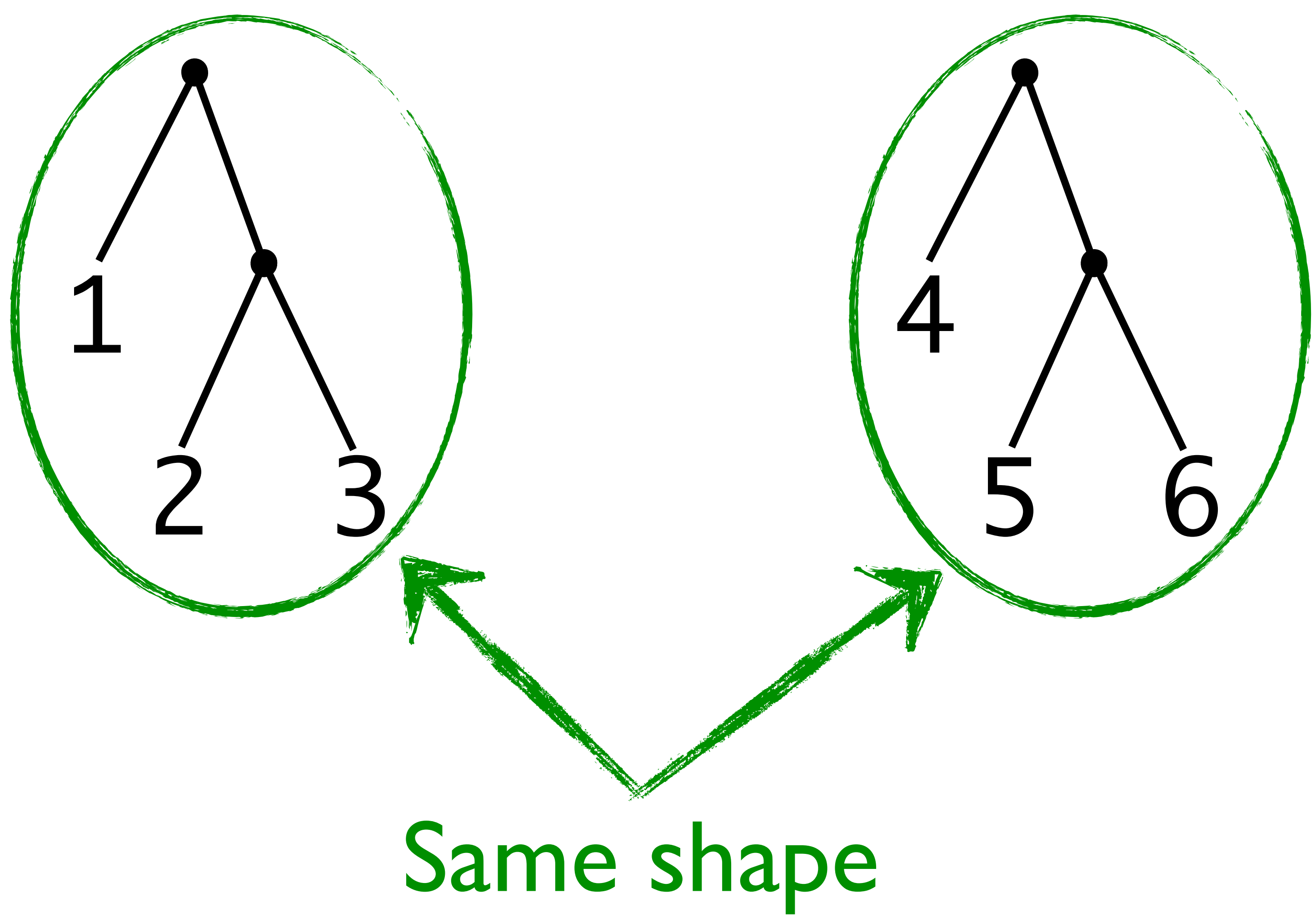
Annoying thing: GHC won't disallow writing this

```
zipWithV f (Cons x xs) Nil = {- ? -} undefined
```

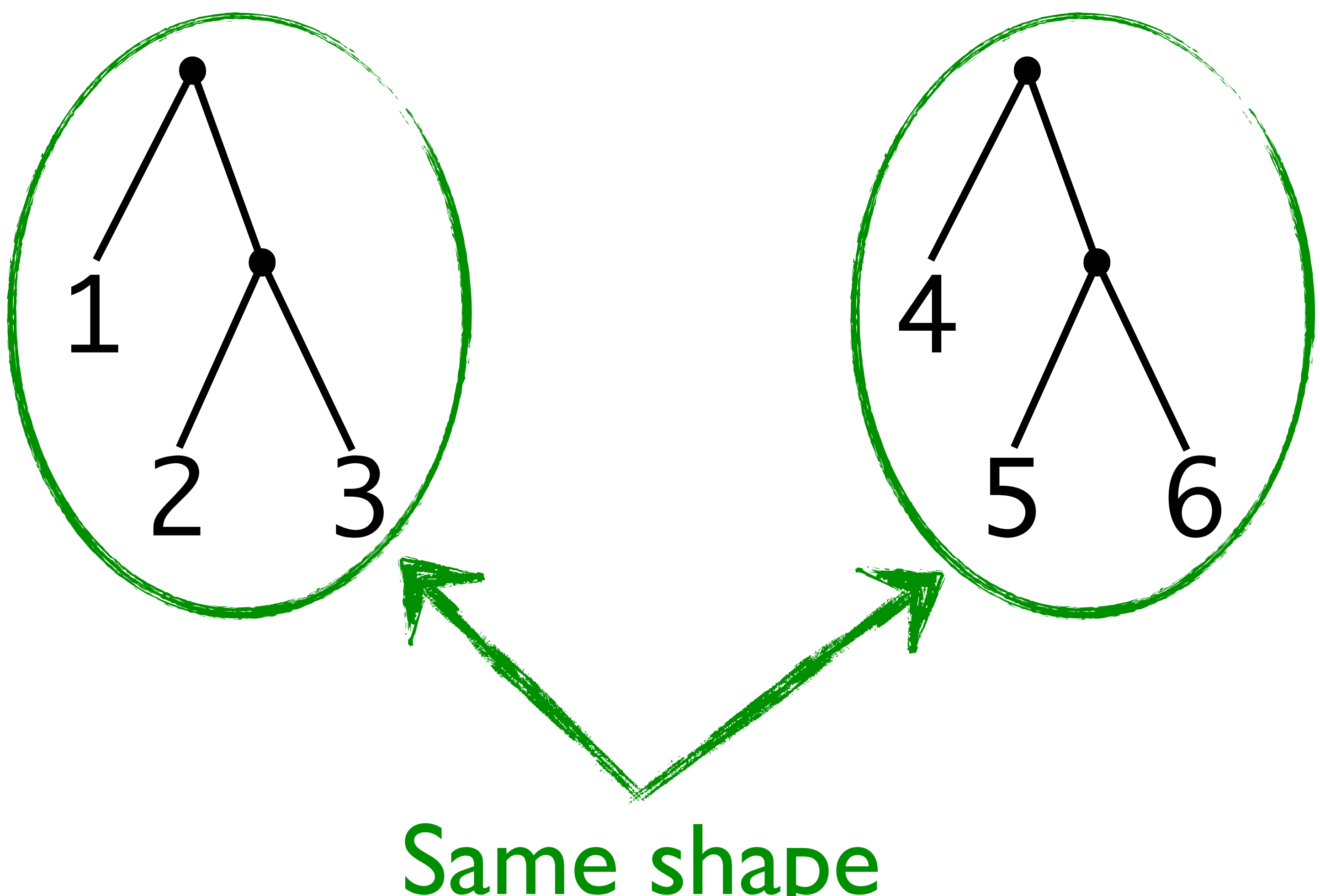
...but this will never be executed at run time

What about other data types?

What about other data types?



What about other data types?



1*4 + 2*5 + 3*6 = 32

zipWith for Trees

zipWith for Trees

```
zipWithT f (Leaf a) (Leaf b)          = Leaf (f a b)
```

zipWith for Trees

```
zipWithT f (Leaf a) (Leaf b)           = Leaf (f a b)
zipWithT f (Branch s t) (Branch s' t') = Branch (zipWithT f s s')
                                                (zipWithT f t t')
```

zipWith for Trees

```
zipWithT f (Leaf a) (Leaf b)              = Leaf (f a b)
zipWithT f (Branch s t) (Branch s' t') = Branch (zipWithT f s s')
                                                (zipWithT f t t')

zipWithT f (Leaf a) (Branch s' t')     = {- ? -} undefined
```

zipWith for Trees

```
zipWithT f (Leaf a) (Leaf b)                   = Leaf (f a b)
zipWithT f (Branch s t) (Branch s' t') = Branch (zipWithT f s s')
                                                       (zipWithT f t t')

zipWithT f (Leaf a) (Branch s' t')        = {- ? -} undefined
zipWithT f (Branch s t) (Leaf b)          = {- ? -} undefined
```

Shapes

- Use *type indexed types*. Encode *shape* of data structure with GADTs
- Type system ensures that only values of same shape can be `zipWith`ed together

Trees with shapes

```
data Tree sh a where
  Leaf   :: a -> Tree () a
  Branch :: Tree m a -> Tree n a -> Tree (m,n) a
```

```
Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
                                    :: Tree ((), ((),())) Int
```

Generalise with Applicative class

- zipWith is actually liftA2 on ZipList instance of Applicative class.
- `Applicative` *lifts* a value into a fragment of a larger domain
- `Applicative` allows you to apply values from this domain to each other.
- All Monads are Applicatives. Not all Applicatives are Monads.

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

Generalising

`zipWith` is actually `liftA2`

```
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
```

Specialised to trees

```
liftA2 :: (a -> b -> c) -> Tree sh a -> Tree sh b -> Tree sh c
```

```
liftA2 (*):: Num a => Tree sh a -> Tree sh a -> Tree sh a
```

General result

For a data structure T

IF you can define Foldable and Applicative instances

THEN you have dot product!

Applicative on Trees with shape

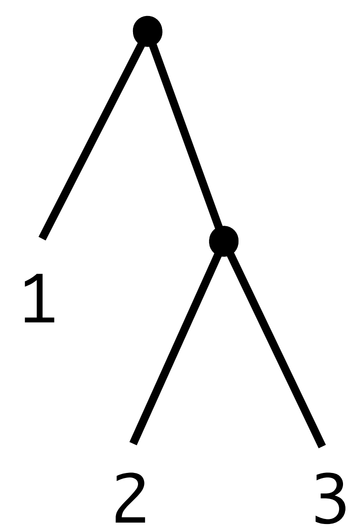
```
instance Applicative (Tree ()) where
  pure a                             = Leaf a
  Leaf fa <*> Leaf a                 = Leaf (fa a)

instance (Applicative (Tree m), Applicative (Tree n)) =>
          Applicative (Tree (m,n)) where
  pure a                             = Branch (pure a) (pure a)
  (Branch fs ft) <*> (Branch s t) = Branch (fs <*> s) (ft <*> t)
```

Let's see `liftA2 (*)` on Trees

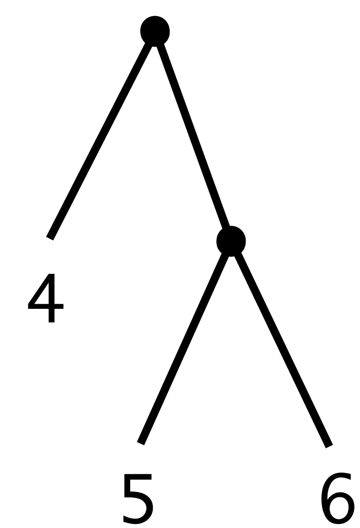
liftA2 (*) on Trees

```
liftA2 f a b = pure f <*> a <*> b
```

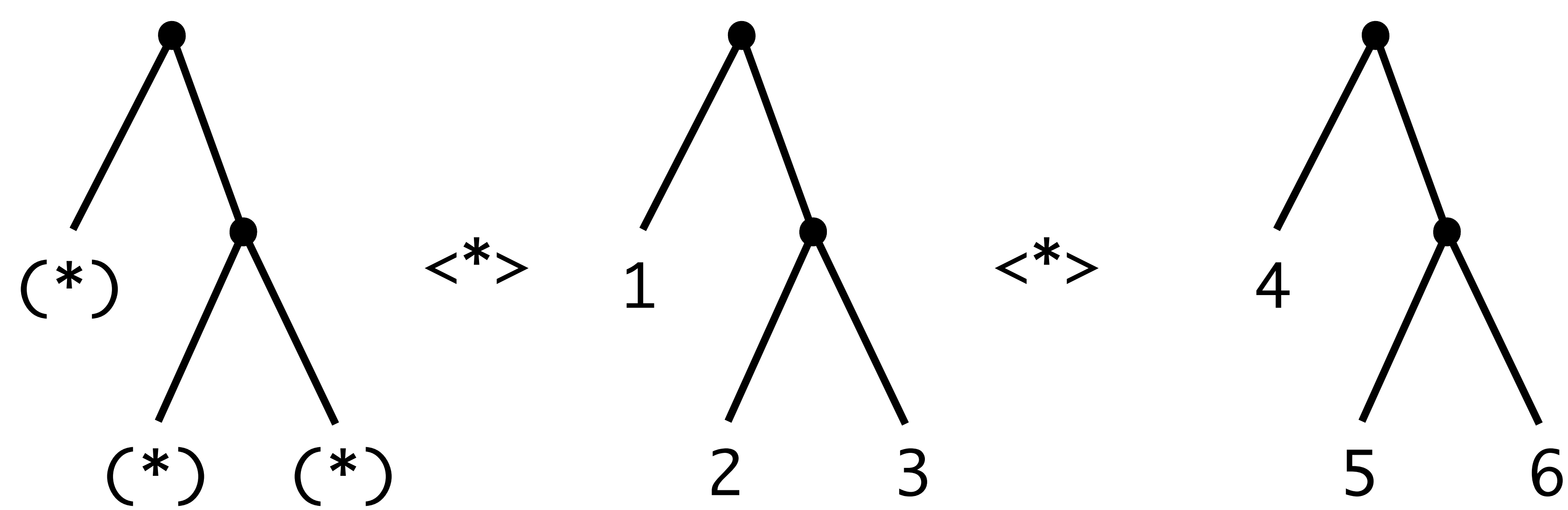
pure (*) <*>



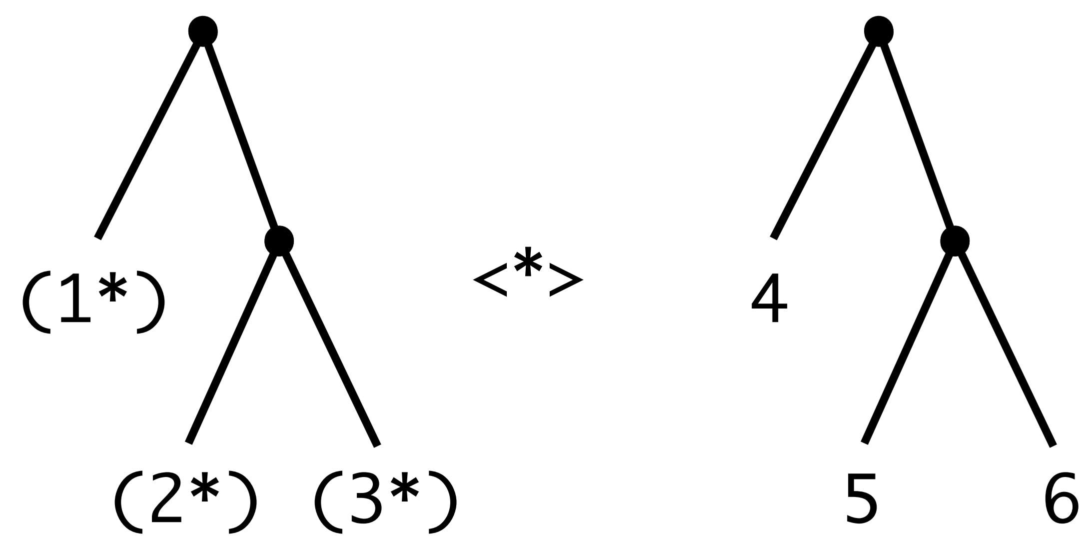
<*>



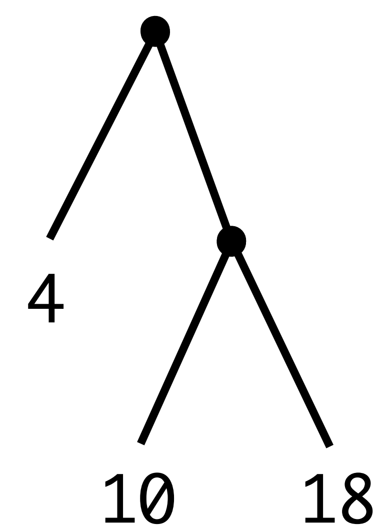
`liftA2 (*)` on Trees



`liftA2 (*)` on Trees

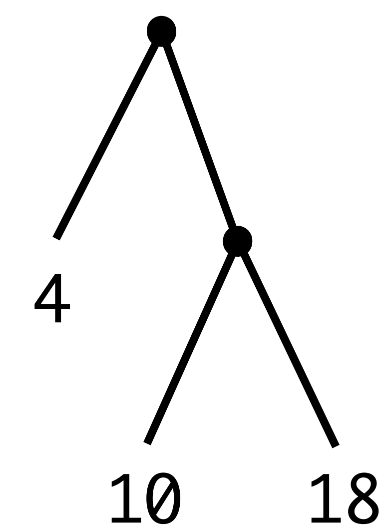


`liftA2 (*)` on Trees



`liftA2 (*)` on Trees

Now just `fold (+)` over this to get dot product



`liftA2 (*)` on Trees

Now just `fold (+)` over this to get dot product

32

Foldable on Trees with shape

```
class Foldable t where
  fold    :: Monoid m => t m -> m
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

```
instance Foldable (Tree sh) where
  -- foldMap :: Monoid m => (a -> m) -> Tree sh a -> m
  foldMap f (Leaf a)     = f a
  foldMap f (Branch s t) = foldMap f s `mappend` foldMap f t

  -- fold :: Monoid m => Tree sh m -> m
```

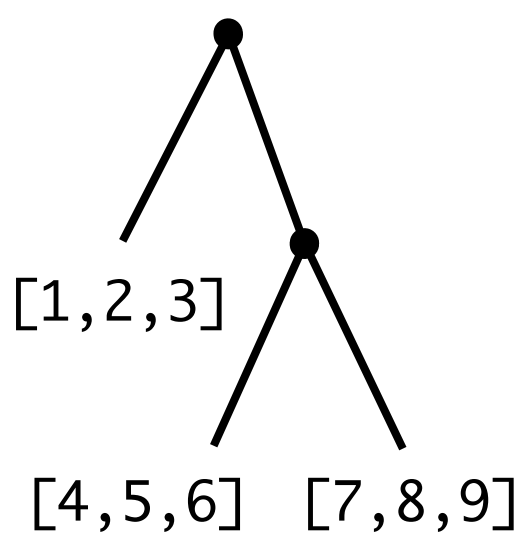
Generic dot product

```haskell
-- Defined in Data.Monoid module
newtype Sum a = Sum { getSum :: a }
        deriving (Eq, Ord, Read, Show, Bounded)

instance Num a => Monoid (Sum a) where
        mempty = Sum 0
        Sum x `mappend` Sum y = Sum (x + y)
```

```haskell
dot :: (Num a, Foldable f, Applicative f) => f a -> f a -> a
dot x y = foldSum $ liftA2 (*) x y
  where foldSum = getSum . fold . fmap Sum
```

What is a matrix?

A collection of collections




```
:: Tree ((), ((), ())) (Vec (S (S (S Z))) Integer)
```

Generalising dimensions

For regular matrices *dimensions* of input matrices determine dimensions of output matrix

$$m{\times}n \times n{\times}p = m{\times}p$$

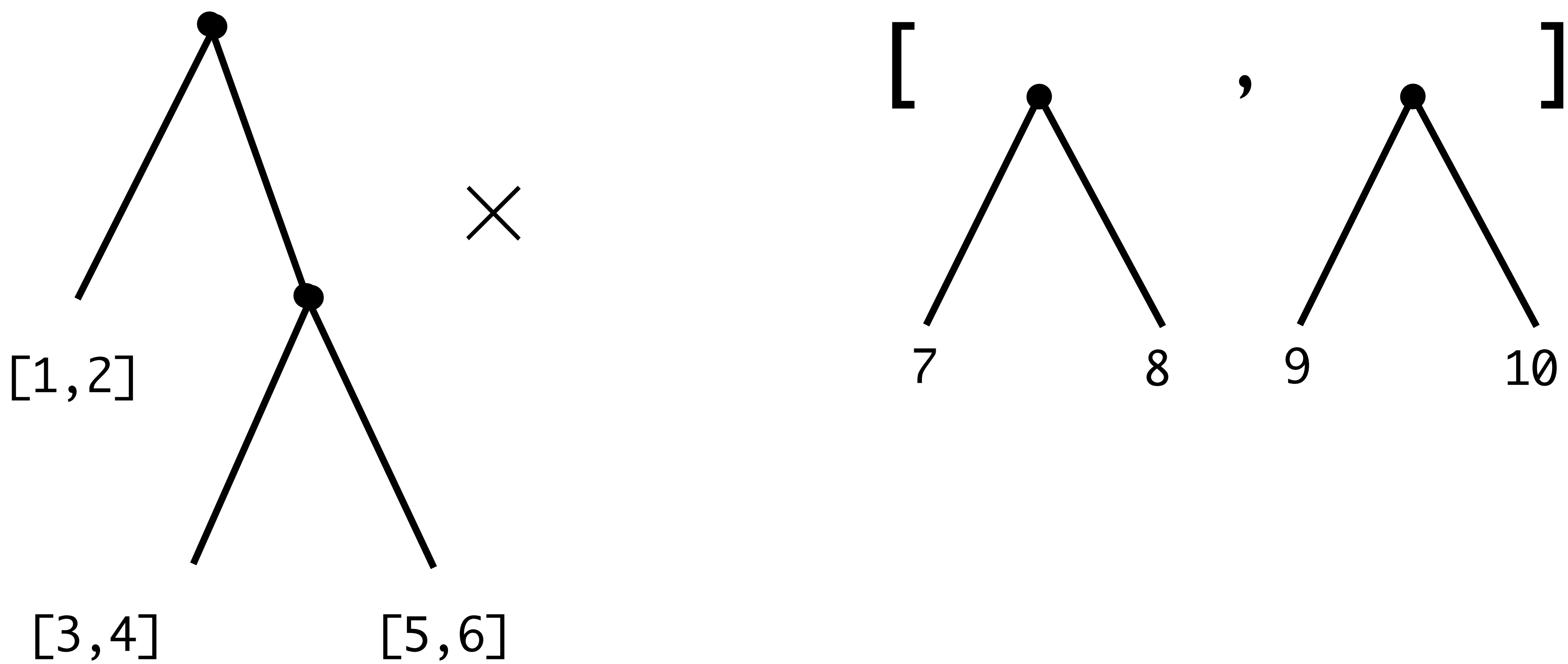
For generic matrices *type and shape* of input matrices determine *type and shape* of output matrix

$$Tree\ s{\times}Vec\ n \times Vec\ n{\times}Tree\ t = Tree\ s{\times}Tree\ t$$

Recap of matrix multiplication

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$$

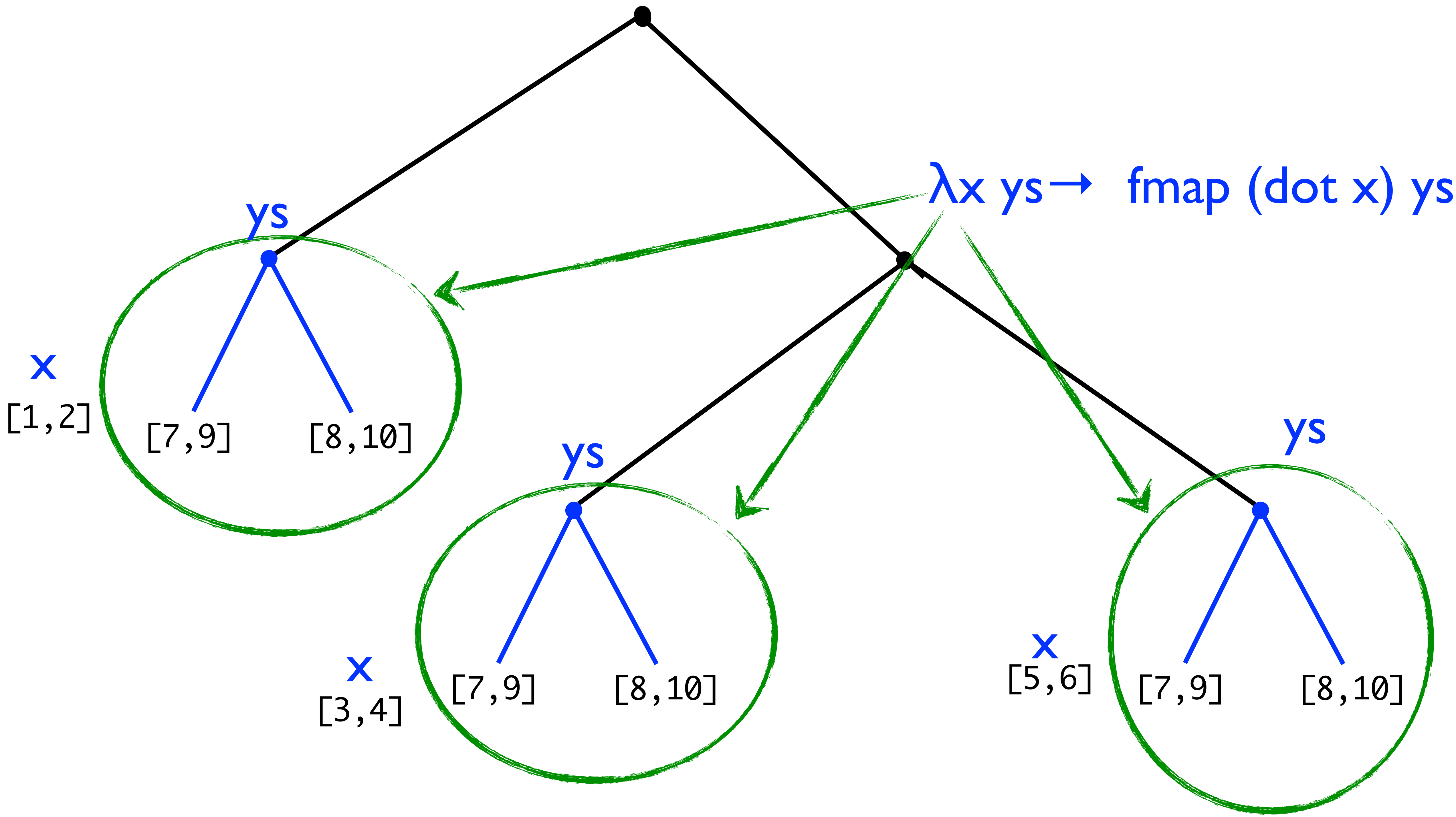
How it's done



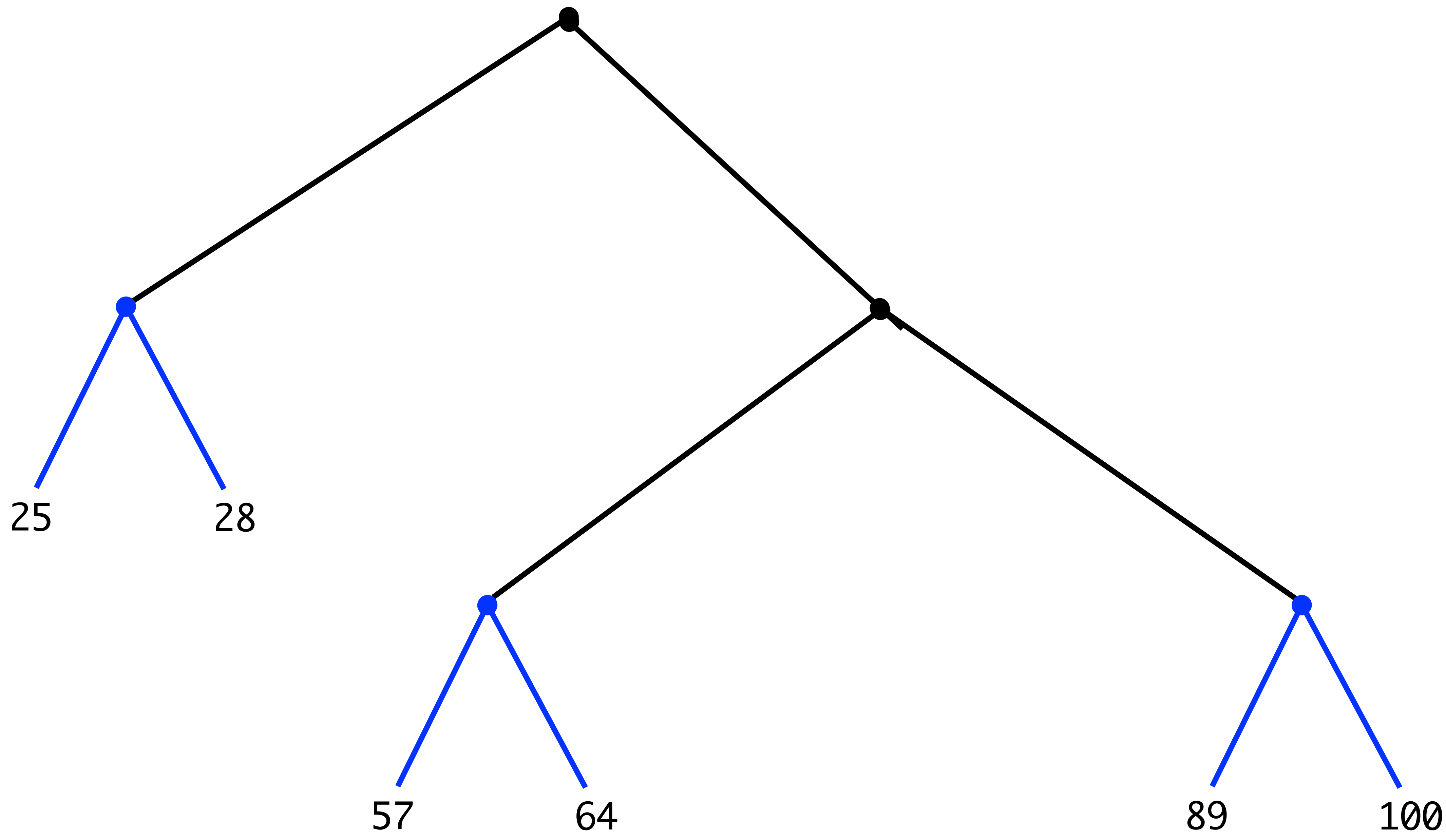
Results should be tree of trees

How it's done

How it's done



How it's done



```haskell
dot :: (Num a, Foldable f, Applicative f) => f a -> f a -> a
dot x y = foldSum $ liftA2 (*) x y
  where foldSum = getSum . fold . fmap Sum
```

```haskell
transpose :: (Traversable f1, Applicative f2)
          => f1 (f2 a) -> f2 (f1 a)
transpose = sequenceA

mmult :: (Num a, Applicative f1, Applicative f2, Applicative f3,
          Traversable f1, Traversable f2)
       => f1 (f2 a) -> f2 (f3 a) -> f1 (f3 a)
mmult m1 m2 = fmap (flip (fmap . dot) (transpose m2)) m1
```

DEMO