Generic Dot Products

Sean Seefried

Dot products

Take two vectors of length n

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

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

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

Works for lists of different lengths because

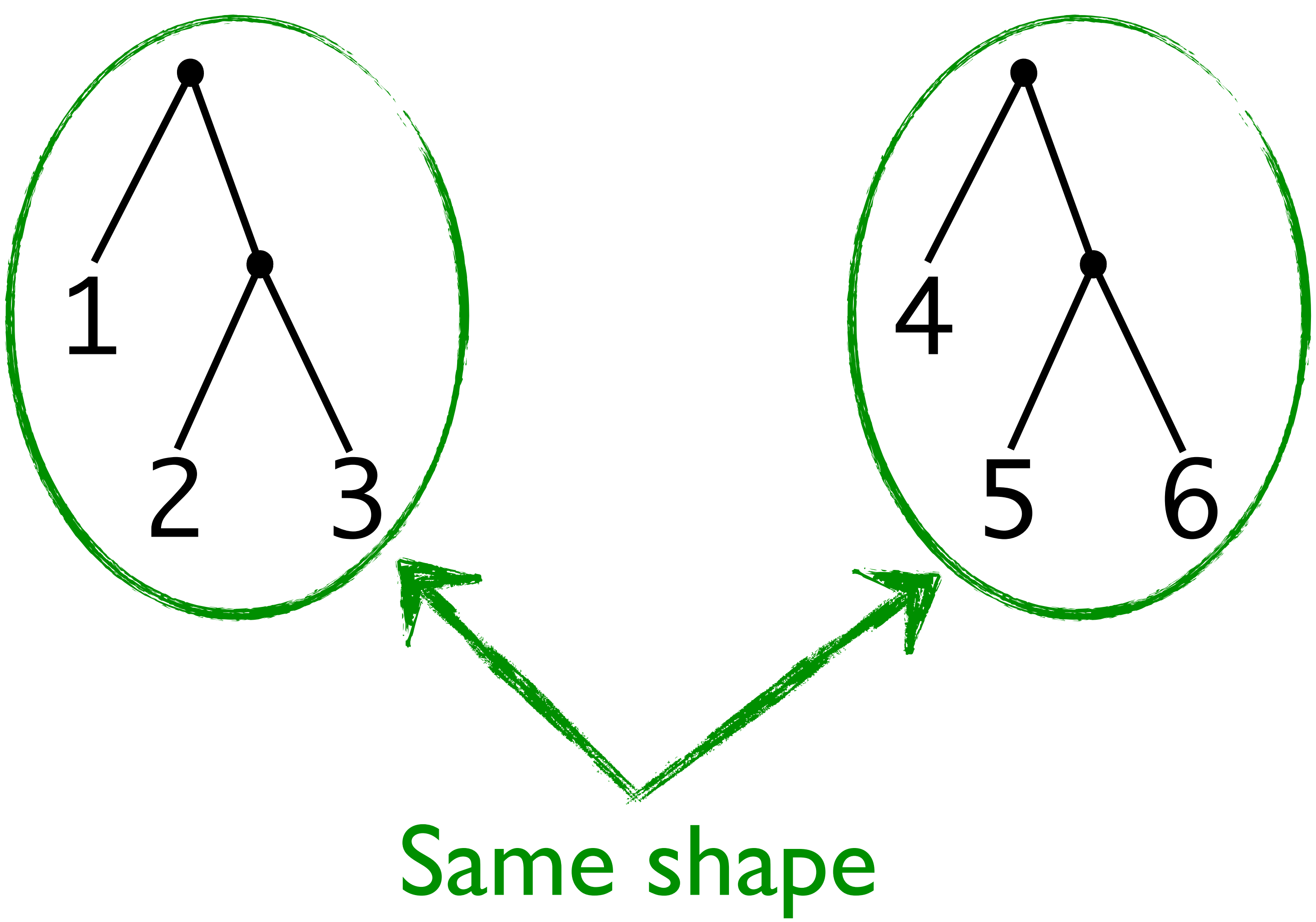
```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f [] _ = []
zipWith f _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

What about trees?

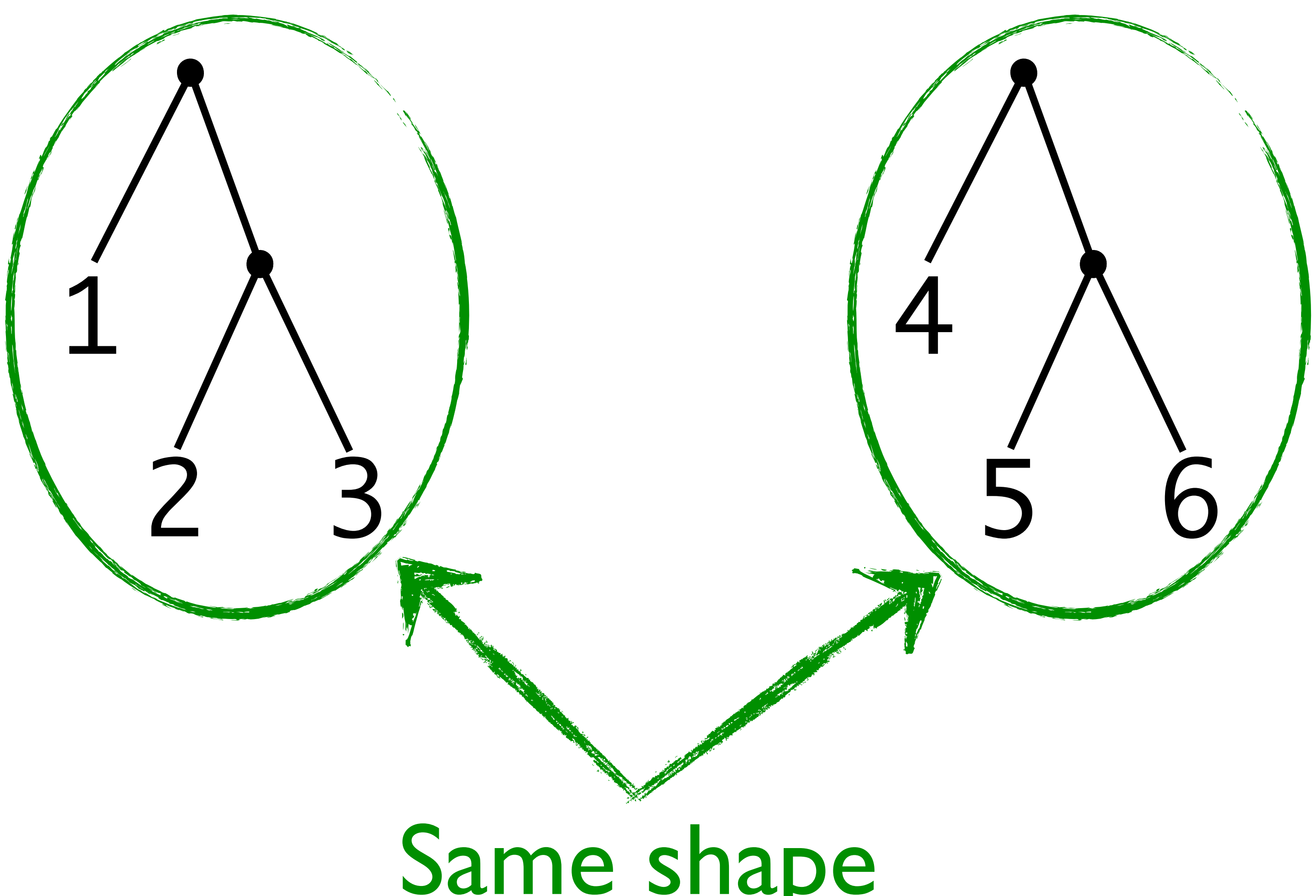
- What does dot product even mean on trees?
- What conditions need to hold?

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

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



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



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

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

Vec

```
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
```

Total functions

```
headVec :: Vec (S n) a -> a
headVec (Cons x _) = x

tailVec :: Vec (S n) a -> Vec n a
tailVec (Cons _ xs) = xs
```

You just can't write the Nil case!

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 this

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

...but there is no way to define this as anything other than `undefined` (⊥)

Trees with shapes

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

Example

```
> :t Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
(Num t) => Tree ((), ((), ())) t
```

zipWithT

```
zipWithT :: (a -> b -> c) -> Tree sh a -> Tree sh b -> Tree sh c
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')
```

Again, type checker won't complain about other cases but they will never be executed

Dot product on trees

```
foldlT :: (a -> b -> a) -> a -> Tree sh b -> a
foldlT f z (Leaf a)     = f z a
foldlT f z (Branch s t) = foldlT f (foldlT f z s) t

dotT :: Num a => Tree sh a -> Tree sh a -> a
dotT t1 t2 = foldlT (+) 0 (zipWithT (*) t1 t2)
```

Let’s generalise!

Generalising

`zipWith` is actually `liftA2`

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

Specialised to lists

```
liftA2 :: (a -> b -> c) -> [a] -> [b] -> [c]
```

What is Applicative?

- Brain-child of Conor McBride and Ross Paterson.
- `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.

Applicative

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

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

`liftA2 (*)` on lists

```
pure (*)     <*>     [1,2,3]     <*>  [4,5,6]
```

liftA2 (*) on lists

```
[(*),(*),(*)]   <*>    [1,2,3]    <*>  [4,5,6]
```

`liftA2 (*)` on lists

```
[(1*),(2*),(3*)]   <*>   [4,5,6]
```

`liftA2 (*)` on lists

```
[4,10,18]
```

Okay, I lied

```
> liftA2 (*) [1,2,3] [4,5,6]
[4,5,6,8,10,12,12,15,18]
```

Applicative on [] is actually list monad

```
> do { x <- [1,2,3]; y <- [4,5,6]; return (x * y) }
[4,5,6,8,10,12,12,15,18]
```

```
> [ x * y | x <- [1,2,3], y <- [4,5,6] ]
[4,5,6,8,10,12,12,15,18]
```

ZipList

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

```
> liftA2 (*) (ZipList [1,2,3]) (ZipList [4,5,6]
ZipList [4,10,18]
```

ZipList is unsatisfying

```
instance Applicative ZipList where
  pure x = ZipList (repeat x)
  ZipList fs <*> ZipList xs = ZipList (zipWith id fs xs)
```

`pure` returns an infinite list

What we had before was really...

```
[(*),(*),(*),...]  <*>     [1,2,3]     <*>   [4,5,6]
```

Wouldn't it be nice if it was just the right length?

Applicative on Vec

You need *two* instances. One for each constructor.

```
instance Applicative (Vec Z) where
  pure _ = Nil
  Nil <*> Nil                                 = Nil

instance Applicative (Vec n) => Applicative (Vec (S n)) where
  pure a = a `Cons` pure a
  (fa `Cons` fas) <*> (a `Cons` as) = fa a `Cons` (fas <*> as)
```

Second instance is like *inductive case* in structural induction proof. Builds up infinite family of Applicative instances for all shapes.

pure produces a vector of exactly the right length!

```
pure 1 :: Vec (S (S (S Z))) Int
<[1,1,1]|3>
```

Better still, the type checker complains when you try to put patterns in the wrong instances!

```
instance Applicative (Vec Z) where
  pure _ = Nil
  Nil <*> Nil                        = Nil
  (fa `Cons` fas) <*> (a `Cons` as) = fa a `Cons` (fas <*> as)
```

```
Couldn't match expected type `Z' against inferred type `S n'
      Expected type: Vec Z b
      Inferred type: Vec (S n) b
    In the expression: fa a `Cons` (fas <*> as)
```

Drum roll please...

Generic dot product

```
dot :: (Num a, Foldable f, Applicative f) => f a -> f a -> a
dot x y = foldl (+) 0 (liftA2 (*) x y)
```

`foldl` is from `Foldable` type class not Prelude!

```
class Foldable t where
  foldl :: (a -> b -> a) -> a -> t b -> a
```

For a data structure T

IF you can define Foldable and Applicative instances

THEN you have dot product!

Applicative on Trees

```
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)
```

Applicative on shapeless Trees. Yuck.

```
instance Applicative Tree where
  pure a                             = ??? -- Leaf or Branch?

  Leaf fa         <*> Leaf a         = Leaf (fa a)
  (Branch fs ft) <*> (Branch s t) = Branch (fs <*> s) (ft <*> t)
  (Leaf fa)       <*> (Branch s t) = ???
  Branch fs ft    <*> Leaf a         = ???
```

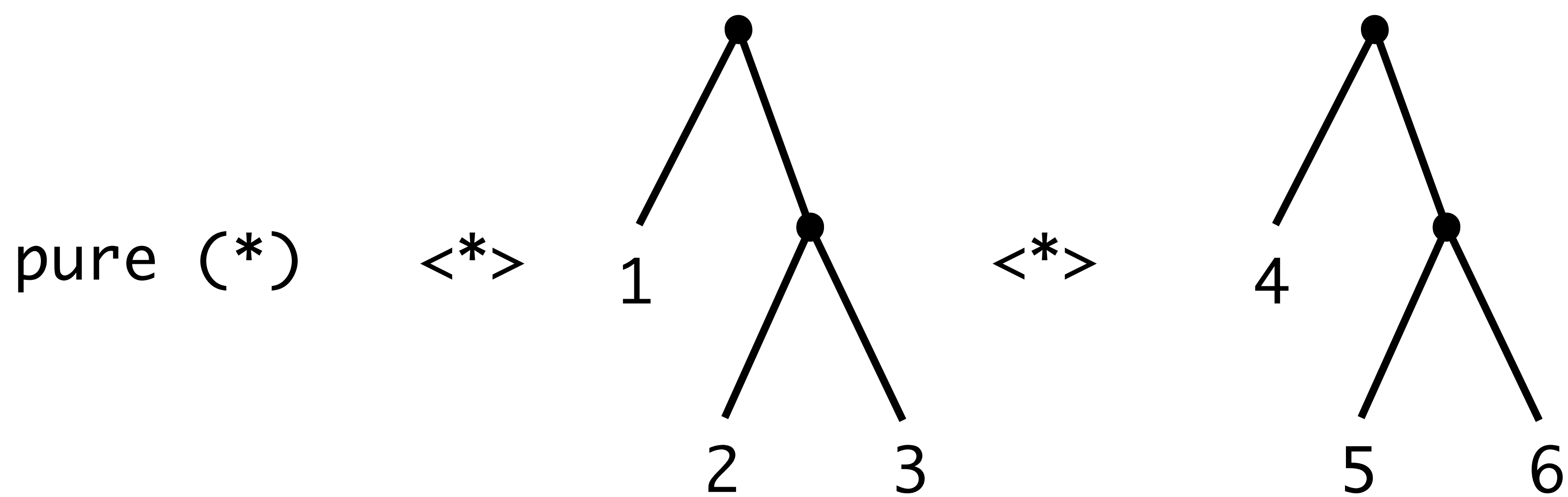
It’s beautiful that insisting on same shape leads to the elegant instance!

“The reward is that the resulting designs are simple and general, and sometimes have the feel of profound *inevitability* —as something beautiful we have discovered, rather than something functional we have crafted. A gift from the gods.”

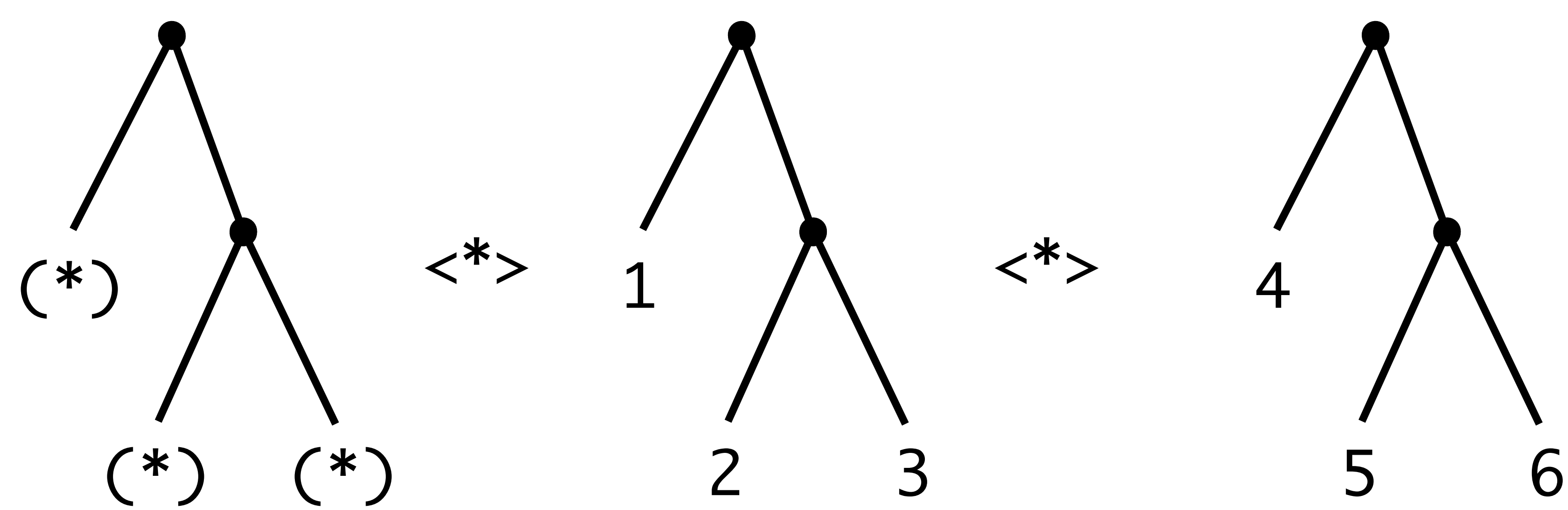
— Conal Elliott in *Denotational design with type class morphisms*

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

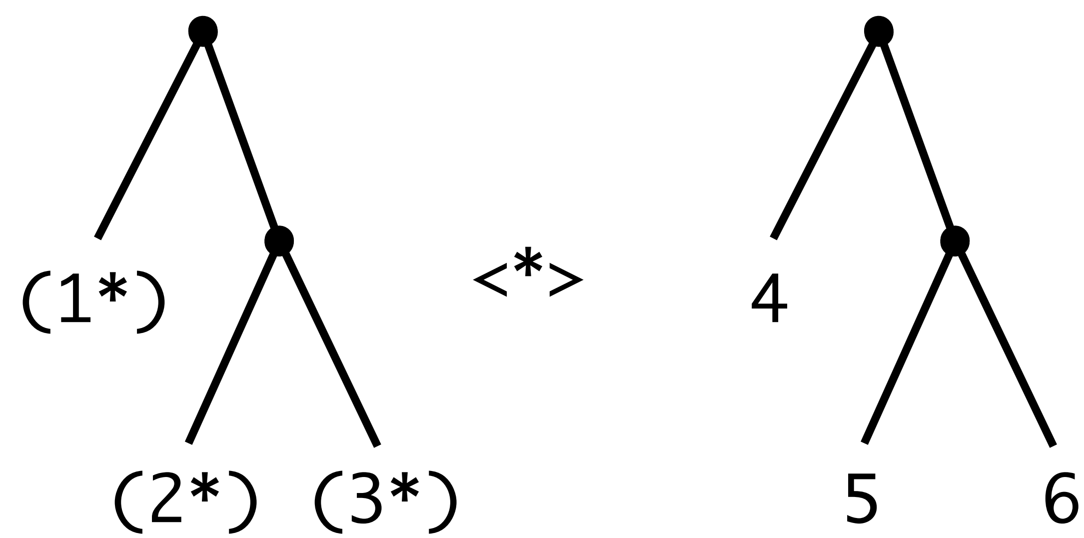
`liftA2 (*)` on Trees



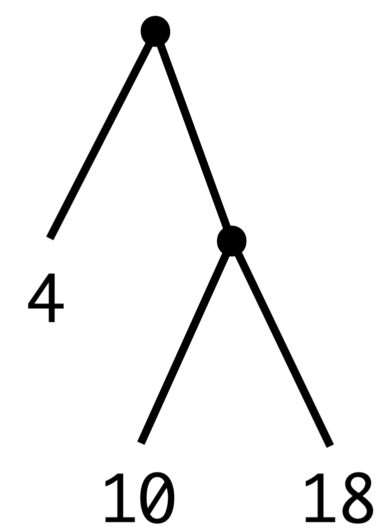
`liftA2 (*)` on Trees



`liftA2 (*)` on Trees



`liftA2 (*)` on Trees



```
> pure 1 :: Tree ((), ((),())) Int
Branch (Leaf 1) (Branch (Leaf 1) (Leaf 1))

> liftA2 (*) t1 t2
Branch (Leaf 4) (Branch (Leaf 10) (Leaf 18))

> let t1 = Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3))
> let t2 = Branch (Leaf 4) (Branch (Leaf 5) (Leaf 6))
> dot t1 t2
32
```

In the next episode...

Generic matrix
multiplication

Teaser

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

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

Why am I doing this?

- Generalise matrix multiplication to data structures other than lists or arrays
- Develop a generic implementation using
  - Reusable algebraic machinery.
  - i.e. Functor, Applicative, Foldable, Traversable
- Derive *work efficient* parallel algorithm.

lambdalog.seanseefried.com

http://lambdalog.seanseefried.com/posts/2011-06-27-generic-dot-products.html