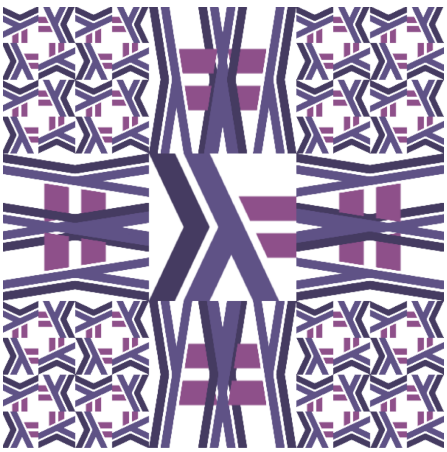
Algebra-Driven Design (Sandy Maguire)

Example: Tiling manager



Basic blocks: Tile: square

$cw : Tile \to Tile$

$ccw : Tile \to Tile$

$cw \cdot cw \cdot cw \cdot cw = id$

$ccw \cdot ccw \cdot ccw \cdot ccw = id$

$cw \cdot ccw = id$

$ccw \cdot cw = id$

Ex: Express ccw as a composition of cw

$ccw = cw \cdot cw \cdot cw$

Proof: $ccw\ t = ccw(cw \cdot cw \cdot cw \cdot cw\ \$\ t) =$

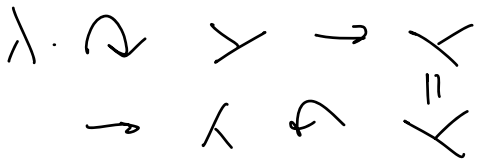
$= ccw \cdot cw\ \$\ (cw \cdot cw \cdot cw\ \$\ t) =$

$= cw \cdot cw \cdot cw\ \$\ t$

$+$ $\qquad\qquad$ $flipH\ t$

$flipH \cdot flipH = id$

$flipH \cdot cw = ccw \cdot flipH$

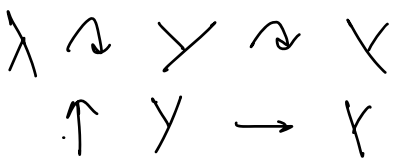


+ flipV ≠

flipV · flipV = id

flipV = ccw · flipH · cw

flipV · flipH = cw · cw



Ex. derive the property from others

$$flipV \cdot flipH = \underline{ccw} \cdot flipH \cdot cw \cdot flipH =$$
$$cw \cdot cw \cdot \underline{cw \cdot flipH} \cdot cw \cdot flipH =$$
$$cw \cdot cw \cdot flipH \cdot \underline{ccw \cdot cw} \cdot flipH =$$
$$cw \cdot cw \cdot \underline{flipH \cdot flipH} =$$
$$cw \cdot cw$$

---

t_1

t_2

beside t_1 t_2

beside t_1 (beside t_1 t_1)

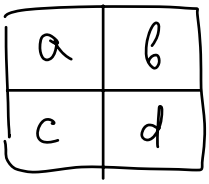
Ex: find a property which links beside & flipH

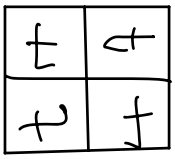
flipH (beside t_1 t_2) = beside (flipH t_2) (flipH t_1)

Ex: Recreate this tile:

above t_1 t_2 =
cw (beside (ccw -1) (ccw t_2))

quad a b c d =
above (beside a b) (beside c d)





swirl t = quad t (cw t) (ccw t) (cw (cw t))

---

behind:

	r	g	b	α
color	1	0.8	0	1

	r	g	b	α
color	0	0.67	0.87	0.5

Ex: come up with laws regarding color

ccw / cw (color r g b α) = color$_1$

beside (color$_1$) (color$_1$) = color$_1$

behind t (color r g b 1) = color r g b 1

empty = color r g b 0 | behind t empty = t

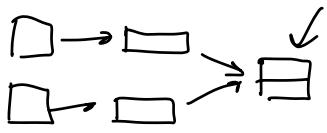
Observations

rasterize :: Int (width) → Int (height) → Tile → [[Color]] (pixels)

∀ w, h:

rasterize w h t_1 = rasterize w h t_2 ⇒ $t_1 = t_2$

```
rasterize w h (flipV t) = reverse (rasterize w h t)
rasterize w h (flipH t) = fmap reverse (rasterize w h t)
```



```
rasterize w h (above t1 t2) =
    rasterize w (h `div` 2) t1 <>
        rasterize w (h - (h `div` 2)) t2

rasterize w h (beside t1 t2) =
    transpose $
      transpose (rasterize (w `div` 2) h t1) <>
       transpose (rasterize (w - (w `div` 2) h t2)

rasterize w h (cw t) =
  fmap reverse (transpose (rasterize h w t))
```

Ex: · define rasterize w h (ccw t)
· define rasterize w h (color r g b a)

```
rasterize w h (color r g b a) =
replicate h (replicate w (rgba r g b a))
```

Generalization

```haskell
data Tile a
instance Functor Tile

rasterize :: Int -> Int -> Tile a -> [[a]]

cw     :: Tile a -> Tile a
ccw    :: Tile a -> Tile a
beside :: Tile a -> Tile a -> Tile a
above  :: Tile a -> Tile a -> Tile a
flipH  :: Tile a -> Tile a
flipV  :: Tile a -> Tile a
quad   :: Tile a -> Tile a -> Tile a -> Tile a -> Tile a
swirl  :: Tile a -> Tile a

color  :: Double -> Double -> Double -> Double -> Tile Color
empty  :: Tile Color
behind :: Tile Color -> Tile Color -> Tile Color
```

```haskell
rasterize :: Tile a -> (Int -> Int -> [[a]])
```

```haskell
f
    :: (Int -> Int -> [[a -> b]])
    -> (Int -> Int -> [[a]])
    -> (Int -> Int -> [[b]])
```

<*>

↓

Make it work with ZipLists

```haskell
rasterize'
    :: Int -> Int -> Tile a -> Compose ZipList ZipList a
```

Law: "rasterize/ap"

```haskell
∀ (w :: Int) (h :: Int) (t1 :: Tile (a -> b))
      (t2 :: Tile a).
  rasterize' w h (t1 <*> t2) =
    rasterize' w h t1 <*> rasterize' w h t2
```

beside (pure id) (pure invert) <*> t_2

```
empty  :: Monoid a => Tile a
behind :: Monoid a => Tile a -> Tile a -> Tile a
```

Law: "behind"

```
∀ (t1 :: Tile a) (t2 :: Tile a).
  behind t1 t2 = liftA2 (<>) t2 t1
```

↑

color blending

Good algebra:

- compositional: build everything from smaller blocks
- task-relevant: directly useful to humans
- parsimonious: each combinator does 1 thing, does it well
- combinators are orthogonal to each other
- closed
- complete
- generalized

Implementation: Initial encoding

```haskell
data Tile a
instance Functor Tile
instance Applicative Tile

rasterize :: Int -> Int -> Tile a -> [[a]]

cw    :: Tile a -> Tile a
ccw   :: Tile a -> Tile a
flipH :: Tile a -> Tile a
flipV :: Tile a -> Tile a

quad  :: Tile a -> Tile a -> Tile a -> Tile a -> Tile a
swirl :: Tile a -> Tile a

beside :: Tile a -> Tile a -> Tile a
above  :: Tile a -> Tile a -> Tile a

empty  :: Monoid a => Tile a
behind :: Monoid a => Tile a -> Tile a -> Tile a
```

```haskell
data Tile a where
  Cw    :: Tile a -> Tile a
  Ccw   :: Tile a -> Tile a
  FlipH :: Tile a -> Tile a
  FlipV :: Tile a -> Tile a
  Quad  :: Tile a -> Tile a -> Tile a -> Tile a -> Tile a
  Swirl :: Tile a -> Tile a
  Beside :: Tile a -> Tile a -> Tile a
  Above  :: Tile a -> Tile a -> Tile a
  Empty  :: Monoid a => Tile a
  Behind :: Monoid a => Tile a -> Tile a -> Tile a
  Fmap :: (a -> b) -> Tile a -> Tile b  ❶
  Pure :: a -> Tile a  ❷
  Ap   :: Tile (a -> b) -> Tile a -> Tile b  ❸
```

combinators implementation

```haskell
cw :: Tile a -> Tile a
cw = Cw

ccw :: Tile a -> Tile a
ccw = Ccw

flipH :: Tile a -> Tile a
flipH = FlipH
```

```haskell
flipH :: Tile a -> Tile a
flipH (FlipH t) = t
flipH t = FlipH t
```

```haskell
cw :: Tile a -> Tile a
cw (Cw (Cw (Cw t))) = t
cw t = Cw t
```

the way to make properties of algebra to hold by construction

```haskell
data Tile a where
  Cw     :: Tile a -> Tile a
  FlipH  :: Tile a -> Tile a
  Above  :: Tile a -> Tile a -> Tile a
  Pure   :: a -> Tile a
  Ap     :: Tile (a -> b) -> Tile a -> Tile b
```

← better initial encoding: only primitive blocks

```haskell
above :: Tile a -> Tile a -> Tile a
above = Above

instance Applicative Tile where
  pure = Pure
  (<*>) = Ap


instance Functor Tile where
  fmap f t = pure f <*> t
```

```haskell
empty :: Monoid a => Tile a
empty = pure mempty

behind :: Monoid a => Tile a -> Tile a -> Tile a
behind = flip (liftA2 (<>))
```

Rasterize:

```haskell
rasterize w h (Pure a) = replicate h $ replicate w a
```

```haskell
rasterize w h (Ap f a) =
  coerce (rasterize' w h f <*> rasterize' w h a)
```

```haskell
rasterize w h (FlipH t) = fmap reverse $ rasterize w h t
```

```haskell
rasterize w h (Above t1 t2) =
    rasterize w (div h 2) t1 <>
    rasterize w (h - div h 2) t2
```

you can then encode properties as property based tests (QuickCheck)
And (?) generate new properties with QuickSpec

Efficient Implementation

— alternative def of Tile

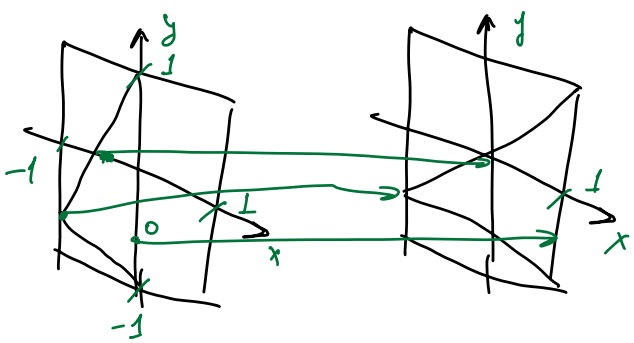
```haskell
type Tile a = Point -> a

type Point = (Double, Double)
```

```haskell
pure :: a -> (Point -> a)
pure a = \_ -> a   . . . . .
```

```haskell
(<*>)
    :: (Point -> (a -> b))
    -> (Point -> a)
    -> (Point -> b)
ff <*> fa = \p -> (ff p) (fa p)
```

```haskell
sample :: Point -> Tile a -> a
```



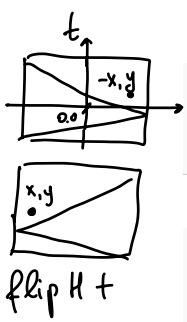

```haskell
sample (x, y) (flipH t) =
  sample (negate x, y) t
```

```haskell
sample (x, y) (flipV t) =
  sample (x, negate y) t
```

```haskell
sample (x, y) (cw t) =
  sample (y, negate x) t
```

```haskell
sample (x, y) (ccw t) =
  sample (negate y, x) t
```

```haskell
sample (x, y) (pure a) = pure a
```




```haskell
x < 0 =>
  sample (x, y) (beside t1 t2) =
    sample ((x + 0.5) * 2, y) t1
```

```haskell
x >= 0 =>
  sample (x, y) (beside t1 t2) =
    sample ((x - 0.5) * 2, y) t2
```

```haskell
sample (x, y) (tf <*> ta) =
  sample (x, y) tf <*> sample (x, y) ta
```