Natural Transformations as Business Logics: An Operational Intuition.

Cass Alexandru

2025-01-10

Motivation

- “Sorting with Bialgebras and Distributive Laws” (HJHWM, 2012)
- NLFPD ’24: I gave a talk abt bialgebraic semantics
- There: Post-hoc analysis of recursion behavior of sorting algs
- Contrived introductory examples: `length`, `replicate`
- This talk: Bottom-up operational intuition for distr. laws as business logic
- Revisit and uncontrive `length`, `replicate` examples
- Haskell examples again using the `recursion-schemes` library
- Less Category Theory, more TikZ animations

Structure

Base Functors of Recursive Datatypes

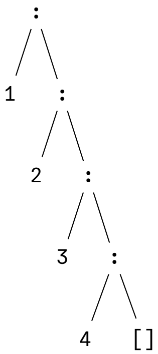
- Recursive datatypes have a shape given by a *base functor* F
- E.g. Natural numbers: $(1 + -)$. Lists of element type A: $(1 + A \times -)$.
- Recursive datatype is given by fixpoint of composition of base functor F with itself
- Least fixpoint: Inductive datatype. Greatest: coinductive – not nec. well founded

Induction

- Can *eliminate* (map out of) a recursive datatype
- Algebra: Compositionally interpret *syntax* to a domain by giving it *semantics*, giving each constructor an *interpretation*.
- Roll up the datatype from the base cases (Bottom-up).

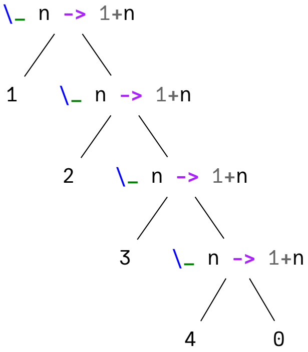
Functions Replace Constructors

List

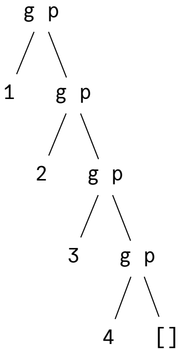


Traversals

`length`



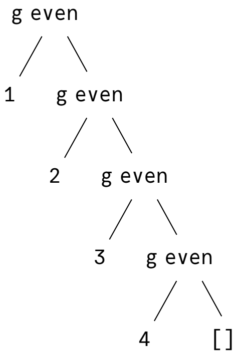
`filter p`




```
g p x xs =
  (if p x then [x] else [])
  ++ xs
```

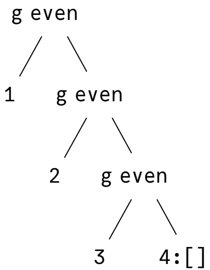
Example Evaluation of filter even

```
g even x xs =
  (if even x then [x] else []) ++ xs
```



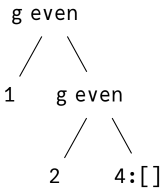
Example Evaluation of `filter even`

```
g even x xs =
  (if even x then [x] else []) ++ xs
```



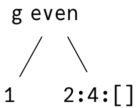
Example Evaluation of `filter even`

```
g even x xs =
  (if even x then [x] else []) ++ xs
```



Example Evaluation of `filter even`

```
g even x xs =
  (if even x then [x] else []) ++ xs
```



Example Evaluation of filter even

```
g even x xs =
  (if even x then [x] else []) ++ xs
```

```
2:4:[]
```

Coinduction

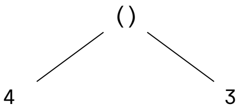
- Map *into* a recursive datatype
- Coalgebra: From a seed, create one level of the datatype, with new seeds at recursive positions
- Iteratively apply until base cases are reached
- NB: Base cases may not be reached! $\rightarrow$ non well founded trees

Example: Growing a Fibonacci Tree

5

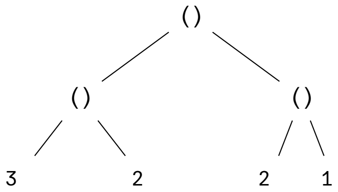
```
fib :: Nat -> TreeF () Nat
fib = \case
  0 -> NodeF () []
  1 -> NodeF () []
  n -> NodeF () [n-1,n-2]
```

Example: Growing a Fibonacci Tree



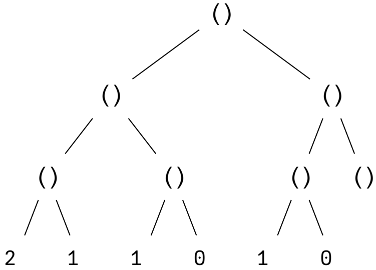

```
fib :: Nat -> TreeF () Nat
fib = \case
  0 -> NodeF () []
  1 -> NodeF () []
  n -> NodeF () [n-1,n-2]
```

Example: Growing a Fibonacci Tree



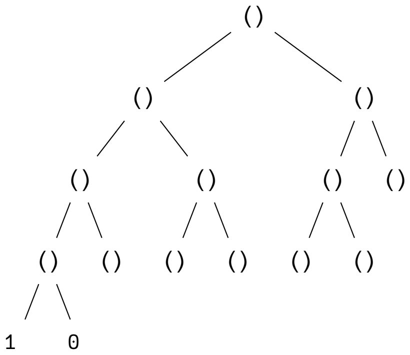

```
fib :: Nat -> TreeF () Nat
fib = \case
  0 -> NodeF () []
  1 -> NodeF () []
  n -> NodeF () [n-1,n-2]
```

Example: Growing a Fibonacci Tree



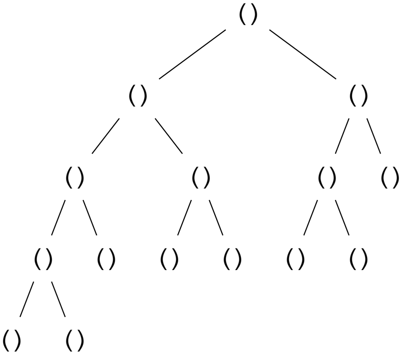

```
fib :: Nat -> TreeF () Nat
fib = \case
  0 -> NodeF () []
  1 -> NodeF () []
  n -> NodeF () [n-1,n-2]
```

Example: Growing a Fibonacci Tree




```
fib :: Nat -> TreeF () Nat
fib = \case
  0 -> NodeF () []
  1 -> NodeF () []
  n -> NodeF () [n-1,n-2]
```

Example: Growing a Fibonacci Tree



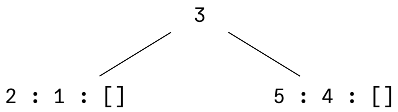

```haskell
fib :: Nat -> TreeF () Nat
fib = \case
  0 -> NodeF () []
  1 -> NodeF () []
  n -> NodeF () [n-1,n-2]
```

Example: Growing a BST with partition

```
2 : 5 : 4 : 1 : 3 : []
```

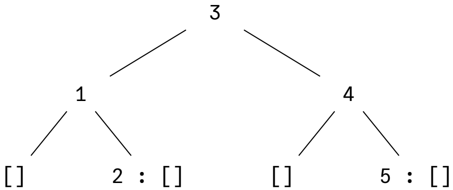
```haskell
partition :: (Ord a) =>
  [a] -> (TreeF a) [a]
```

Example: Growing a BST with `partition`



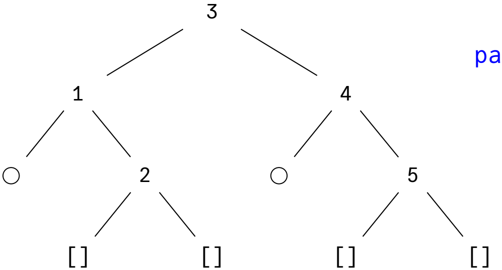

```haskell
partition :: (Ord a) =>
  [a] -> (TreeF a) [a]
```

Example: Growing a BST with partition



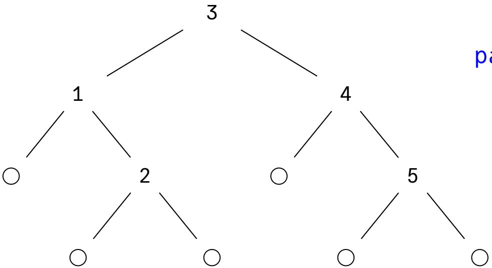

```haskell
partition :: (Ord a) =>
  [a] -> (TreeF a) [a]
```

Example: Growing a BST with partition




```
partition :: (Ord a) =>
  [a] -> (TreeF a) [a]
```

Example: Growing a BST with partition




```
partition :: (Ord a) =>
  [a] -> (TreeF a) [a]
```

Structure

Maps Between Recursive Datatypes

- `Rec F -> Rec G`
- Algebraically: `F (Rec G) -> Rec G`
- Coalgebraically: `Rec F -> G (Rec F)`
- A secret third option?

Natural Transformations

- $\delta \colon F \Rightarrow G$

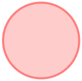
Natural Transformations

- $\delta\colon F \Rightarrow G$
- $$\begin{array}{ccc} FX & \xrightarrow{\delta_X} & GX \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle Gf} \\ FY & \xrightarrow{\delta_Y} & GY \end{array}$$

Natural Transformations

- $\delta \colon F \Rightarrow G$
- $$\begin{array}{ccc} FX & \xrightarrow{\delta_X} & GX \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle Gf} \\ FY & \xrightarrow{\delta_Y} & GY \end{array}$$
- ```
  type f :=> g = (Functor f, Functor g) =>
    forall a.  f a -> g a
  ```
  ```

Natural Transformations

- $\delta\colon F \Rightarrow G$
- $$\begin{array}{ccc} FX & \xrightarrow{\delta_X} & GX \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle Gf} \\ FY & \xrightarrow{\delta_Y} & GY \end{array}$$
- ```haskell
  type f :=> g = (Functor f, Functor g) =>
    forall a. f a -> g a
  ```
- 
  ```

Natural Transformations

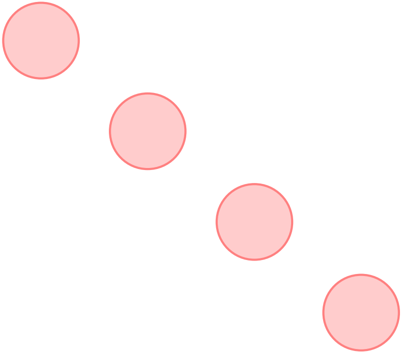
- $\delta \colon F \Rightarrow G$
- $$\begin{array}{ccc} FX & \xrightarrow{\delta_X} & GX \\ \downarrow{\scriptstyle Ff} & & \downarrow{\scriptstyle Gf} \\ FY & \xrightarrow{\delta_Y} & GY \end{array}$$
-
```haskell
type f :=> g = (Functor f, Functor g) =>
  forall a. f a -> g a
```
-

length & replicate

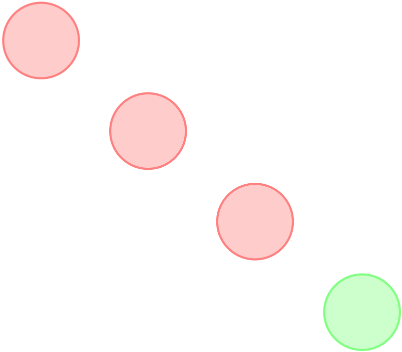
```haskell
forget :: ListF a :=> NatF
forget = \case
  Nil -> Zero
  Cons _ x -> Suc x
```

```haskell
deco :: a -> NatF :=> (ListF a)
deco e = \case
  Zero -> Nil
  Suc x -> Cons e x
```

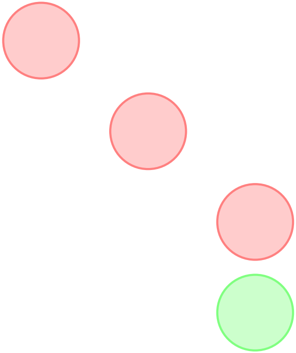
Bottom-up



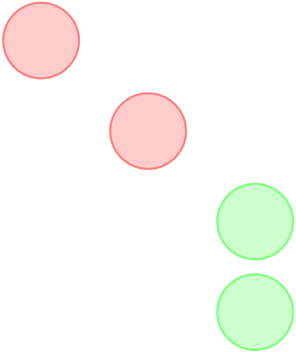
Bottom-up



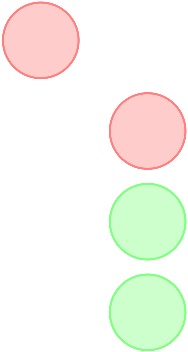
Bottom-up



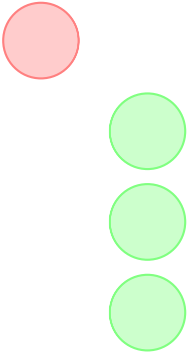
Bottom-up



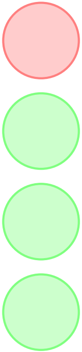
Bottom-up



Bottom-up

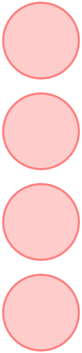


Bottom-up

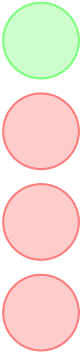


Bottom-up

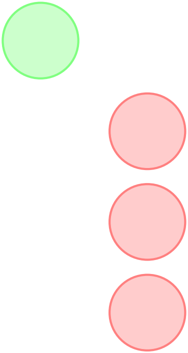
Top-down



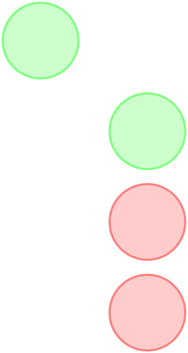
Top-down



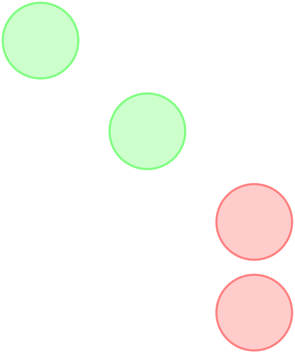
Top-down



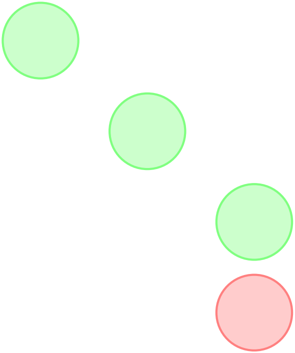
Top-down



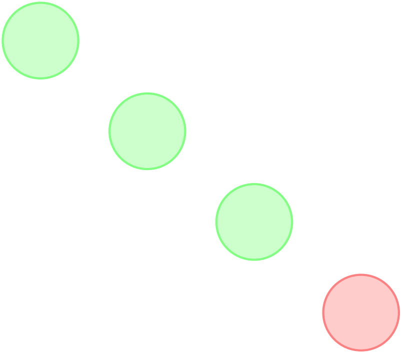
Top-down



Top-down



Top-down



Top-down

Natural Transformation Semantics

```
natSem :: forall μf νg . (Recursive μf, Corecursive νg) =>
  (Base μf :=> Base νg) -> μf -> νg
natSem δ = fold @μf alg where
   alg :: (Base μf) νg -> νg
   alg = embed @νg . δ

coNatSem :: forall μf νg . (Recursive μf, Corecursive νg) =>
  (Base μf :=> Base νg) -> μf -> νg
coNatSem δ = unfold @νg coalg where
  coalg :: μf -> (Base νg) μf
  coalg = δ @μf . project
```

- NB: For `natSem`, we used `embed` from the `recursion-schemes` library
- Corresponds to the initial algebra ⇒ `νg` isn't actually codata
- For `coNatSem` we used `unfold`. But (`δ @μf . project`) is a *recursive coalgebra* (transposition proposition in (Eppendahl, 2000))
- Both of these semantics go between carriers of *initial* algebras, so data, not codata

Structure

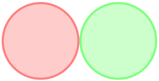
Distributive Laws

- $FG \Rightarrow GF$

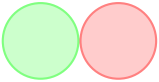
Distributive Laws

- $FG \Rightarrow GF$
-
```
type DistrLaw = (Functor f, Functor g) =>
  forall a.  f (g a) -> g (f a)
```

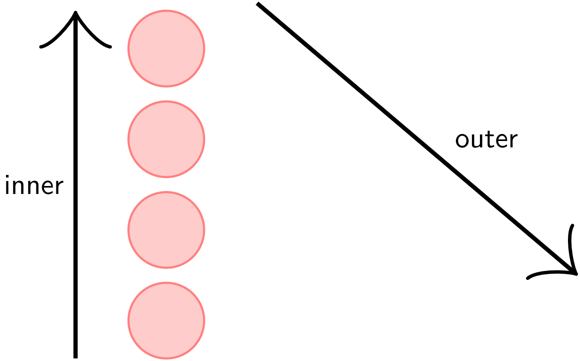
Distributive Laws

- $FG \Rightarrow GF$
- ```haskell
  type DistrLaw = (Functor f, Functor g) =>
    forall a.  f (g a) -> g (f a)
  ```
- Alternatively:
  ```haskell
  type DistrLaw' = f `Compose` g :=> g `Compose` f
  ```
  ```

Distributive Laws

- $FG \Rightarrow GF$
- ```
  type DistrLaw = (Functor f, Functor g) =>
    forall a.  f (g a) -> g (f a)
  ```
- Alternatively:
  ```
  type DistrLaw' = f `Compose` g :=> g `Compose` f
  ```
- 
  ```

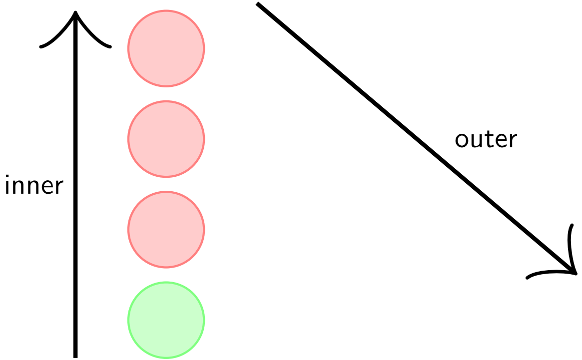
Distributive Laws

- $FG \Rightarrow GF$
-
```haskell
type DistrLaw = (Functor f, Functor g) =>
  forall a.  f (g a) -> g (f a)
```
- Alternatively:
```haskell
type DistrLaw' = f `Compose` g :=> g `Compose` f
```
- 

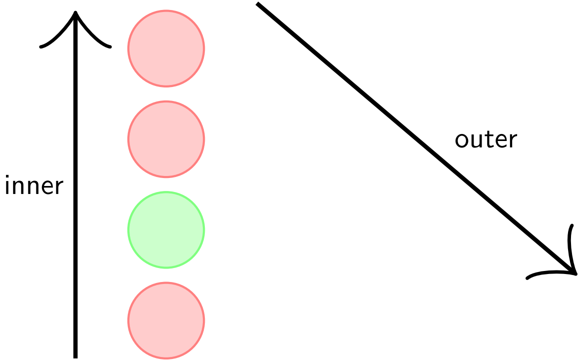
Top-Down 2



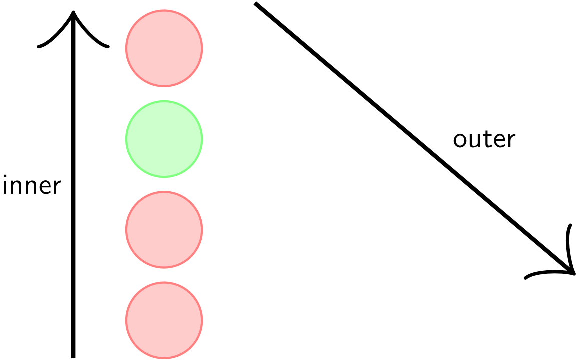
Top-Down 2



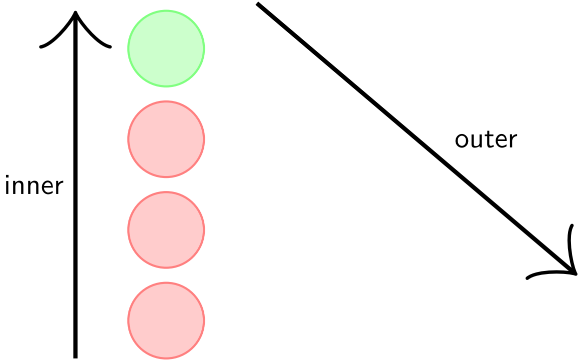
Top-Down 2



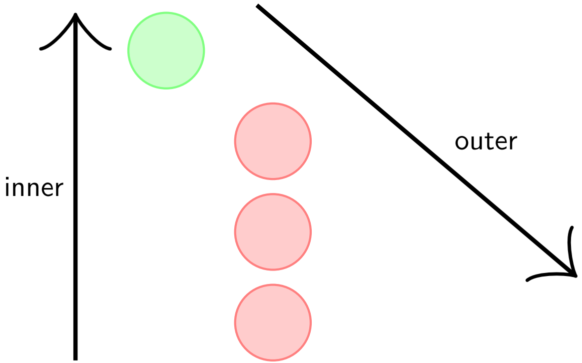
Top-Down 2



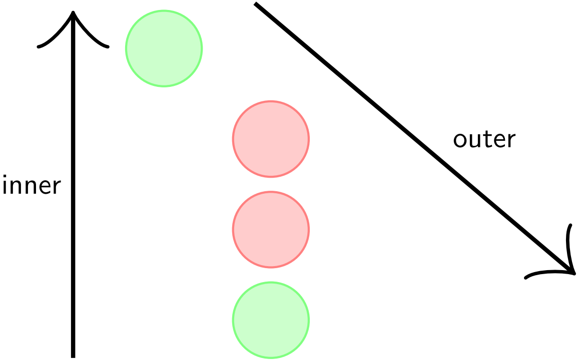
Top-Down 2



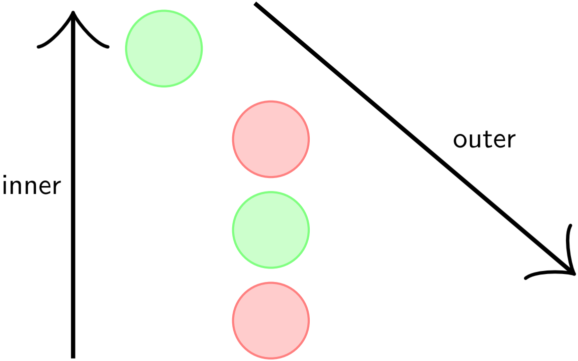
Top-Down 2



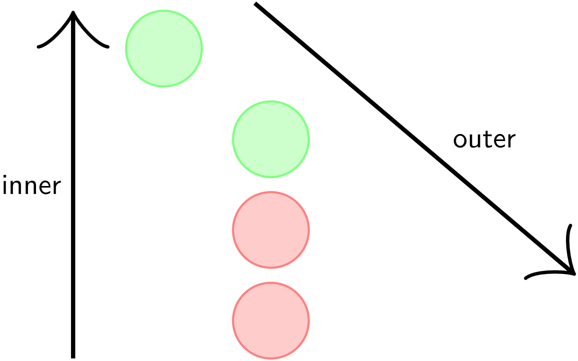
Top-Down 2



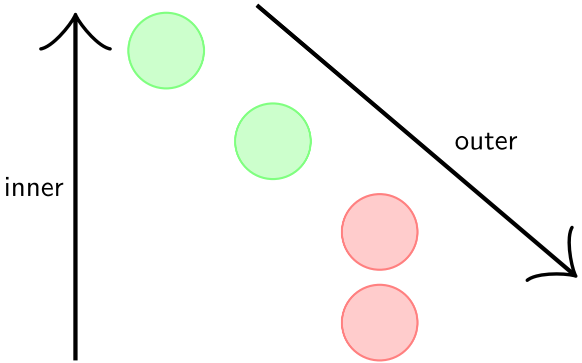
Top-Down 2



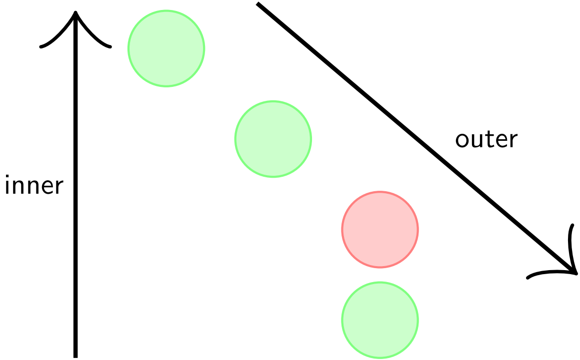
Top-Down 2



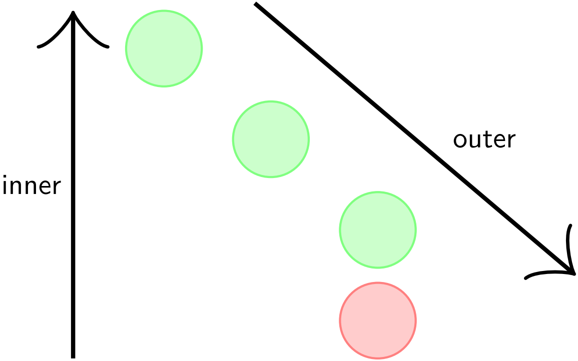
Top-Down 2



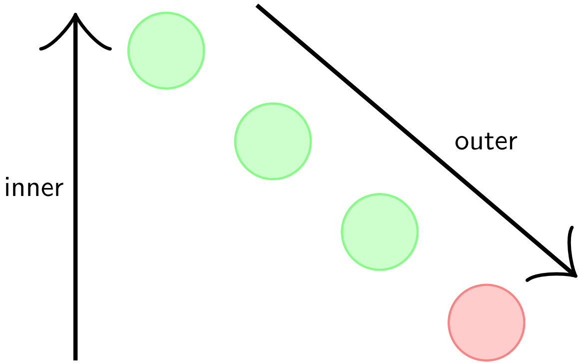
Top-Down 2



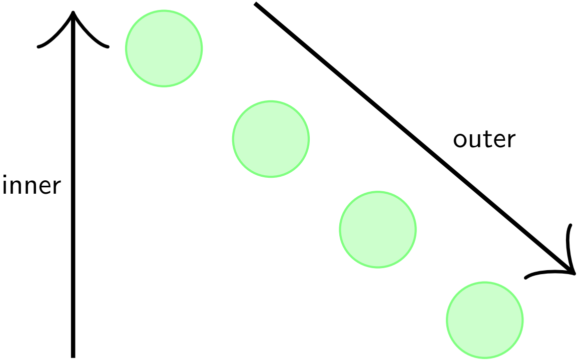
Top-Down 2



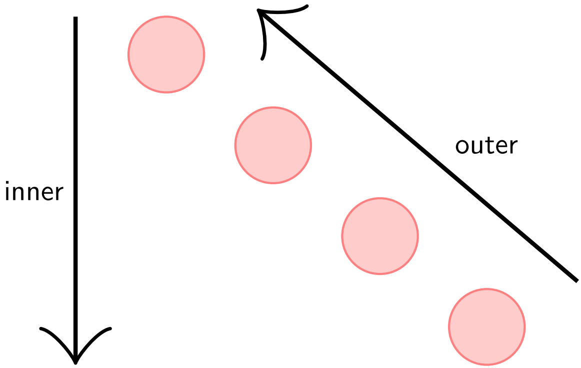
Top-Down 2



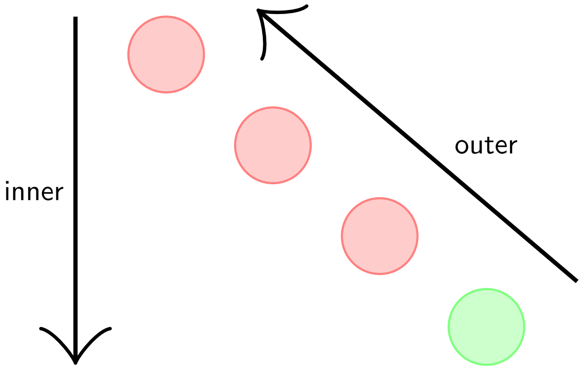
Top-Down 2



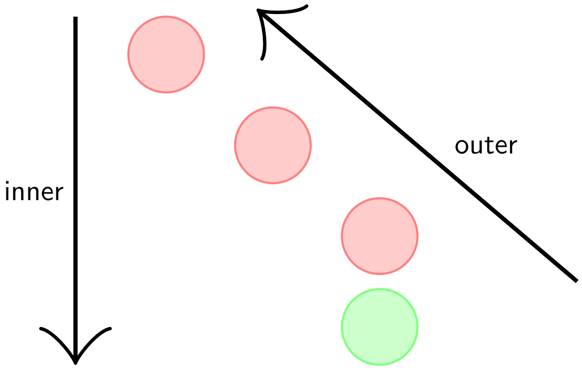
Bottom-up 2



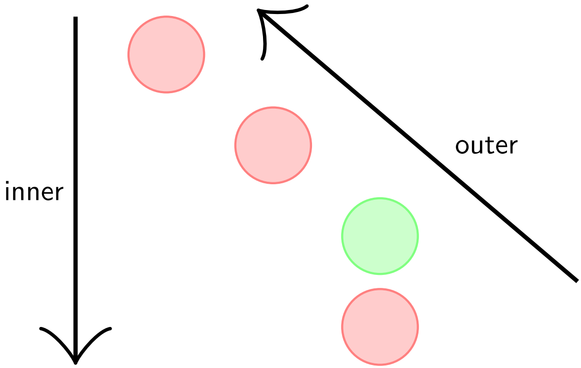
Bottom-up 2



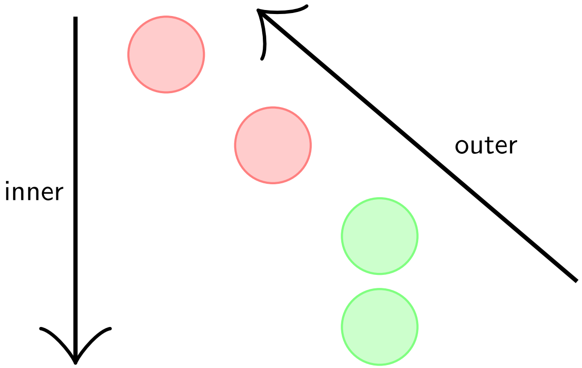
Bottom-up 2



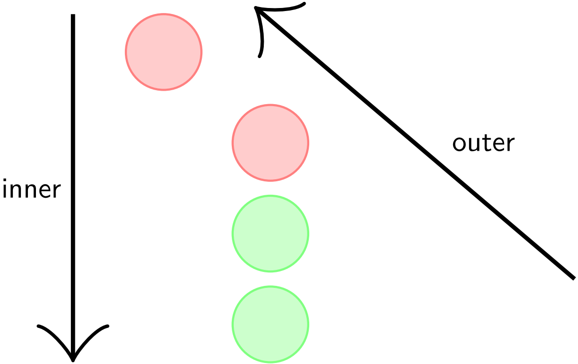
Bottom-up 2



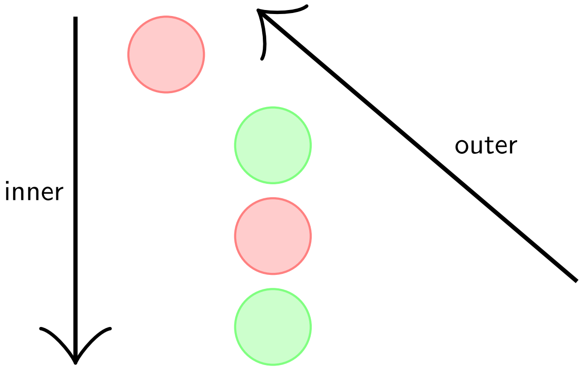
Bottom-up 2



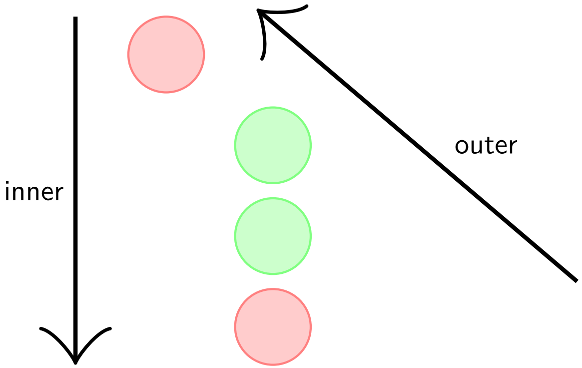
Bottom-up 2



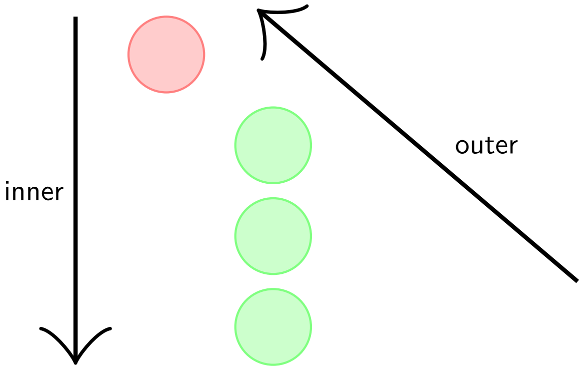
Bottom-up 2



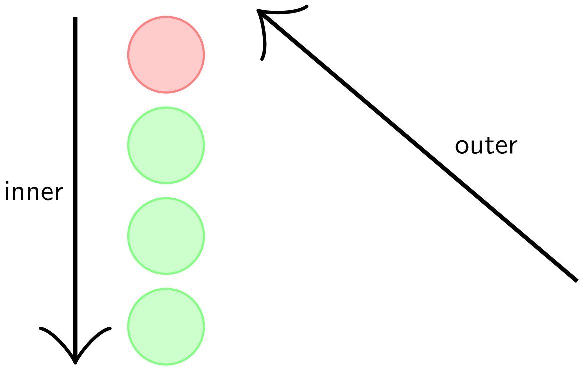
Bottom-up 2



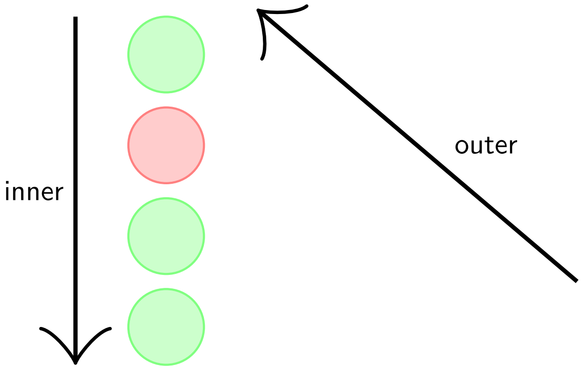
Bottom-up 2



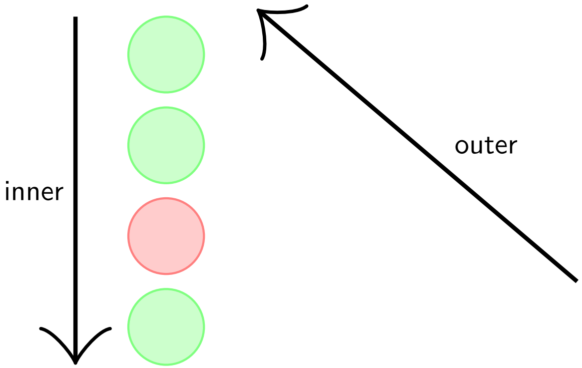
Bottom-up 2



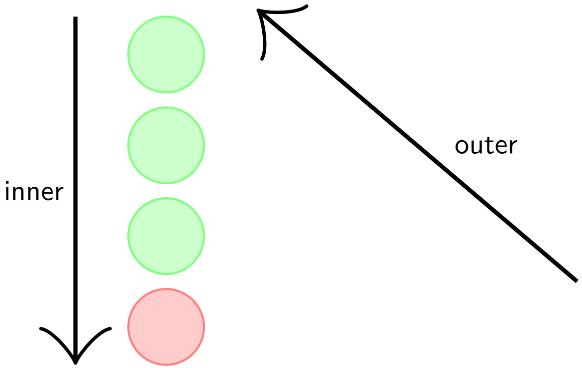
Bottom-up 2



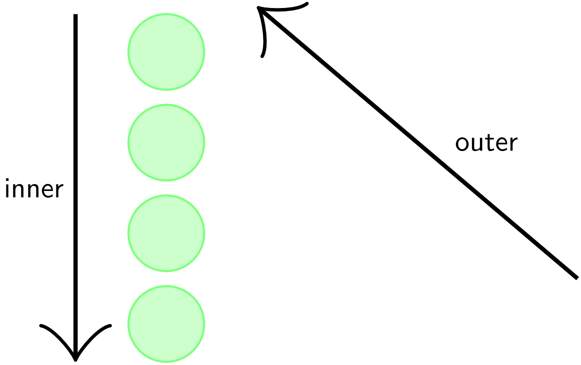
Bottom-up 2



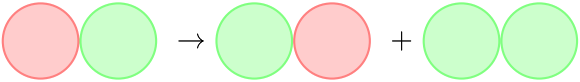
Bottom-up 2



Bottom-up 2



Insights

- Early termination avails only in the nested coalgebraic step
- 
- Lazy evaluation: Variant with outer unfold is always *incremental*, outer fold is (in general) *monolithic*

Example: Insertion- / Selection Sort

```
σ :: (Ord a) => L a (O a r) -> Either (O a (L a r)) (O a (O a r))
σ = \case
  Nil             -> Nil
  a `Cons` Nil    -> a `OCons` Nil
  a `Cons` (b `OCons` r)
    | a <= b      -> Right $ a `OCons` b `OCons` r
    | otherwise   -> Left  $ b `OCons` a `Cons` r
```

Applications

- Recent paper "Intrinsically Correct Sorting in Cubical Agda" (Alexandru and Choudhury and Rot and van der Weide, CPP '25)
- more in-depth on verified bialgebraic sorting algorithms
- encoding invariants in base functors led to the coincidence of partial and total correctness I talked about at NLFPD '24
- makes the input/output base functors meaningfully different (otherwise just aliased)
- correctness of BL in form of distr. law yields correctness of entire algorithm (both variants)

Structure

The Cata is a Lie

- $$\begin{array}{ccc} F\mu F & \xrightarrow{F?} & FX \\ \Big\downarrow \text{in} & & \Big\downarrow a \\ \mu F & \xrightarrow{?} & X \end{array}$$

The Cata is a Lie

- $$\begin{array}{ccc} F\mu F & \overset{F?}{\dashrightarrow} & FX \\ {\scriptstyle \mathrm{in}^{-1}}\big\uparrow\big\downarrow{\scriptstyle \mathrm{in}} & & \big\downarrow{\scriptstyle a} \\ \mu F & \overset{?}{\dashrightarrow} & X \end{array}$$

The Cata is a Lie

- $$\begin{array}{ccc} F\mu F & \overset{F?}{\dashrightarrow} & FX \\ {\scriptstyle \mathrm{in}^{-1}}\uparrow\downarrow{\scriptstyle \mathrm{in}} & & \downarrow{\scriptstyle a} \\ \mu F & \overset{?}{\dashrightarrow} & X \end{array}$$
- $? := a \circ F? \circ \mathrm{in}^{-1}$

The Cata is a Lie

- $$\begin{array}{ccc} F\mu F & \overset{F?}{\dashrightarrow} & FX \\ {\scriptstyle \mathrm{in}^{-1}}\big\uparrow\big\downarrow{\scriptstyle \mathrm{in}} & & \big\downarrow{\scriptstyle a} \\ \mu F & \overset{?}{\dashrightarrow} & X \end{array}$$
- $? := a \circ F? \circ \mathrm{in}^{-1}$
- Coalgebra-to-algebra morphism from recursive coalgebra in^{-1}

The Cata is a Lie

- $$\begin{array}{ccc} F\mu F & \xrightarrow{F?} & FX \\ \mathrm{in}^{-1}\big\uparrow\big\downarrow\mathrm{in} & & \big\downarrow a \\ \mu F & \xrightarrow{?} & X \end{array}$$
- $? := a \circ F? \circ \mathrm{in}^{-1}$
- Coalgebra-to-algebra morphism from recursive coalgebra in^{-1}
- “We believe that, as long as structured recursion is concerned, recursive coalgebras are a more basic concept than initial algebras” – (Capretta and Uustalu and Vene, 2004)

Future Work

- No proof assistant will recognize unfolds of rec. coalgs as terminating

Future Work

- No proof assistant will recognize unfolds of rec. coalgs as terminating
- This holds for both basic rec. coalgs s.a. the inverse of an initial algebra, as well as those constructed from modular parts as in (Capretta and Uustalu and Vene, 2004), (Hinze and Wu and Gibbons, 2015)

Future Work

- No proof assistant will recognize unfolds of rec. coalgs as terminating
- This holds for both basic rec. coalgs s.a. the inverse of an initial algebra, as well as those constructed from modular parts as in (Capretta and Uustalu and Vene, 2004), (Hinze and Wu and Gibbons, 2015)
- Termination checking in general is non-compositional and syntactic – I'm hoping to look into if rec. coalgs can change that

Future Work

- No proof assistant will recognize unfolds of rec. coalgs as terminating
- This holds for both basic rec. coalgs s.a. the inverse of an initial algebra, as well as those constructed from modular parts as in (Capretta and Uustalu and Vene, 2004), (Hinze and Wu and Gibbons, 2015)
- Termination checking in general is non-compositional and syntactic – I'm hoping to look into if rec. coalgs can change that
- More future work: More algorithms with BL in form of nat. trans. – simple, distr. law, or: distr. law with coherence conditions (comonad over a functor, etc.) (Turi and Plotkin, '97)

- `@cxandru@types.pl`
- `cxandru.ee`