When the Types Align: A coincidence of total and partial correctness with a slice of cubical Agda

Cass Alexandru

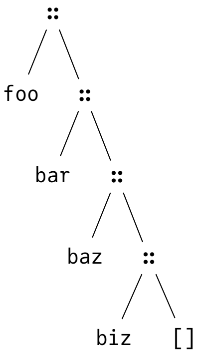
2024-01-05

Motivation

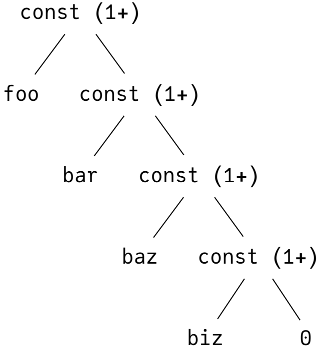
- “Sorting with Bialgebras and Distributive Laws” (HJHWM, 2012)
- Intrinsically correct version using cubical agda
- Haskell examples using the `recursion-schemes` library

Folds

a list:



a fold (`length`):




```
length = foldr 0 (const (1+))
```

Unfolds

- `unfoldr :: (b -> Maybe (a, b)) -> b -> [a]`
- ```
  replicate :: a -> Nat -> [a]
  replicate e = unfoldr produce where
    produce :: Nat -> Just (a, Nat)
    produce = \case Zero -> Nothing; n@(Suc m) -> Just (e, m)
  ```
  ```

Unfolds

- `unfoldr :: (b -> Maybe (a, b)) -> b -> [a]`
- ```
  replicate :: a -> Nat -> [a]
  replicate e = unfoldr produce where
    produce :: Nat -> Just (a, Nat)
    produce = \case Zero -> Nothing; n@(Suc m) -> Just (e, m)
  ```
- ```
  repeat :: a -> [a]
  repeat e = unfoldr produce where
    produce :: () -> Maybe (a,())
    produce = const (Just (e,()))
  ```

Unfolds

- `unfoldr :: (b -> Maybe (a, b)) -> b -> [a]`
- ```
  replicate :: a -> Nat -> [a]
  replicate e = unfoldr produce where
    produce :: Nat -> Just (a, Nat)
    produce = \case Zero -> Nothing; n@(Suc m) -> Just (e, m)
  ```
- ```
  repeat :: a -> [a]
  repeat e = unfoldr produce where
    produce :: () -> Maybe (a,())
    produce = const (Just (e,()))
  ```
- ```
  foldNat :: b -> (b -> b) -> Nat -> b
  replicate' e = foldNat [] (e ::)
  ```
  ```

Base Functor

```haskell
type family Base t :: * -> *

data ListF a r = Nil | Cons a r
type instance Base [a] = ListF a

data NatF r = Zero | Suc r
type instance Base Nat = NatF
```

Algebraic Semantics

Categorical semantics of `folds`

Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor $X : \mathcal{C}$, $\varphi : FX \to X$. Then $FX \xrightarrow{\varphi} X$ (or (X, φ)) is an *F-Algebra*, and X its *carrier*.

Algebra:

$$\begin{array}{c} FX \\ \big\downarrow{\scriptstyle \varphi} \\ X \end{array}$$

Algebra-Hom: $(X, \varphi) \xrightarrow{f} (Y, \psi)$

$$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ \big\downarrow{\scriptstyle \varphi} & & \big\downarrow{\scriptstyle \psi} \\ X & \xrightarrow{f} & Y \end{array}$$

Initial Algebra: $(\mu F, \mathsf{in})$ s.t. $\forall (X, \psi)$.

$$\begin{array}{ccc} F\mu F & \overset{F(\!|\psi|\!)}{\dashrightarrow} & FX \\ \big\downarrow{\scriptstyle \mathsf{in}} & & \big\downarrow{\scriptstyle \psi} \\ \mu F & \overset{(\!|\psi|\!)}{\dashrightarrow} & X \end{array}$$

Coalgebraic Semantics

Categorical semantics of `unfolds`

Let $B : \mathcal{C} \to \mathcal{C}$ be an endofunctor $X : \mathcal{C}$, $\varphi : X \to BX$. Then $X \xrightarrow{\varphi} BX$ (or (X, φ)) is a *B-Coalgebra*, and X its *carrier*.

Coalgebra:

$$\begin{array}{c} BX \\ \uparrow{\scriptstyle \varphi} \\ X \end{array}$$

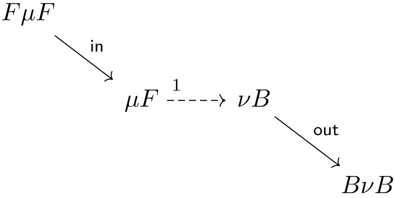
Coalgebra-Hom:
$(X, \varphi) \xrightarrow{f} (Y, \psi)$

$$\begin{array}{ccc} BX & \xrightarrow{Bf} & BY \\ \uparrow{\scriptstyle \varphi} & & \uparrow{\scriptstyle \psi} \\ X & \xrightarrow{f} & Y \end{array}$$

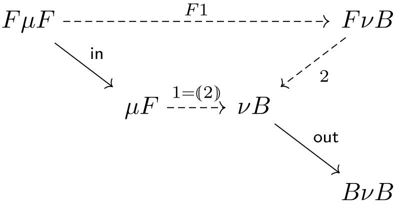
Final Coalgebra:
$(\nu B, \mathsf{out})$ s.t. $\forall (X, \psi)$.

$$\begin{array}{ccc} B\nu B & \xleftarrow{B[\![\psi]\!]} & BX \\ \uparrow{\scriptstyle \mathsf{out}} & & \uparrow{\scriptstyle \psi} \\ \nu B & \xleftarrow{[\![\psi]\!]} & X \end{array}$$

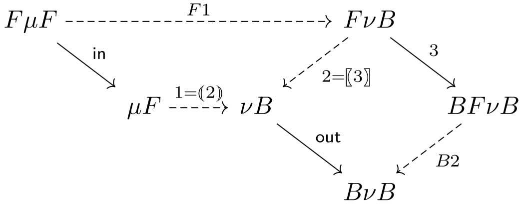
Bialgebraic semantics



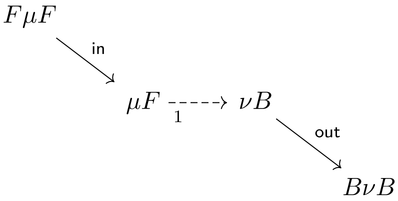
Bialgebraic semantics



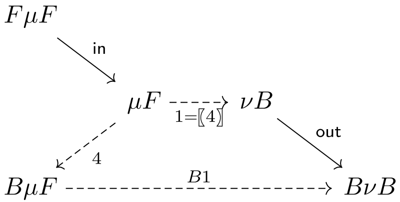
Bialgebraic semantics



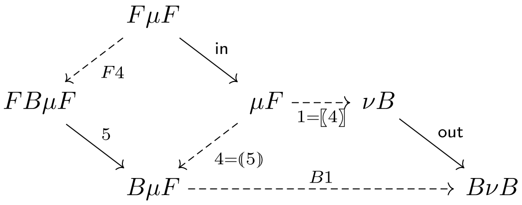
Bialgebraic semantics



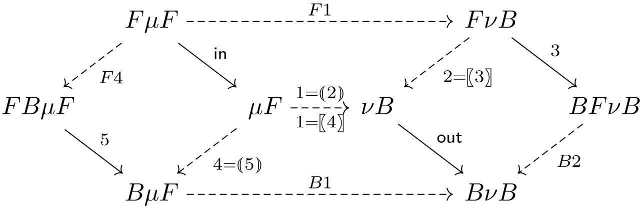
Bialgebraic semantics



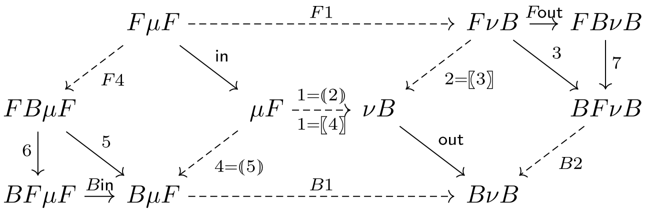
Bialgebraic semantics



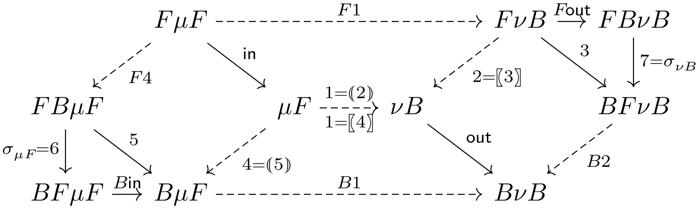
Bialgebraic semantics



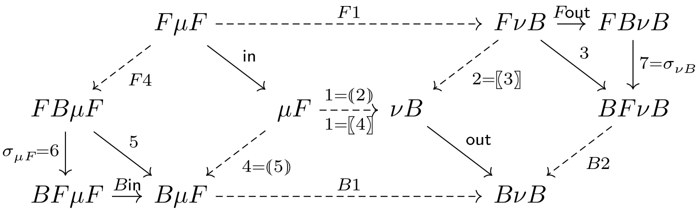
Bialgebraic semantics



Bialgebraic semantics



Bialgebraic semantics



$$\llparenthesis \llbracket \sigma_{\nu B} \circ F\mathsf{out} \rrbracket \rrparenthesis = \llbracket \llparenthesis B\mathsf{in} \circ \sigma_{\mu F} \rrparenthesis \rrbracket$$

Bialgebraic semantics: Takeaway

- Map between recursive types w/ base functors `F`, `B`
- Provide business logic `:: forall r. F (B r) -> B (F r)`
- Receive extensionally identical maps defined as fold/unfold `:: Mu F -> Nu B`

replicate, bialgebraically

$$(\![[\![\sigma_{\nu B} \circ F\mathsf{out}]\!]]\!) = [\![(\![B\mathsf{in} \circ \sigma_{\mu F}]\!)]\!]$$

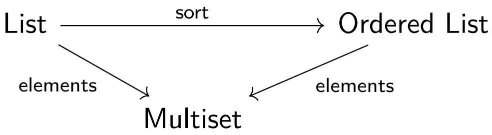
```
swap :: a -> NatF (ListF a r) -> ListF a (NatF r)
swap e = \case
  Zero            -> Nil
  Suc Nil         -> Cons e Zero
  Suc (Cons _ r) -> Cons e (Suc r)

replicate, replicate' :: forall a. a -> Nat -> [a]
replicate  e = fold (unfold (swap e . fmap @Maybe out))
replicate' e = unfold (fold (fmap @(ListF a) in . swap e))
```

Sorting with bialgebras and distributive laws

```haskell
infixr 5 :⋊, :⊗
pattern a :⋊ r = Cons a r
pattern a :⊗ r = Cons a r

type L a = ListF a
type O a = ListF a

σ :: (Ord a) => L a (O a r) -> O a (L a r)
σ = \case
  Nil             -> Nil
  a :⊗ Nil        -> a :⋊ Nil
  a :⊗ (b :⋊ r)
    | a <= b      -> a :⋊ b :⊗ r
    | otherwise -> b :⋊ a :⊗ r

insSort, bubblesort :: forall a. Ord a => [a] -> [a]
insSort    = fold (unfold (σ . fmap @(L a) out))
bubblesort = unfold (fold (fmap @(O a) in . σ))
```

Sort Club

- The first rule of sorting is: The output list should be *ordered*.
  $\sum_{n\in\mathbb{N}}\{\sigma \in A^n \mid \forall i < n-1.\, \sigma_i \leq \sigma_{i+1}\}$
- The second rule of sorting is: The output is a permutation of the input.



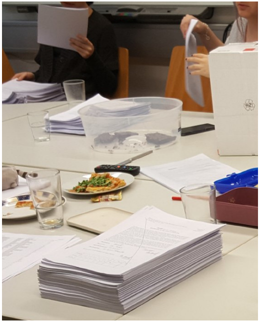
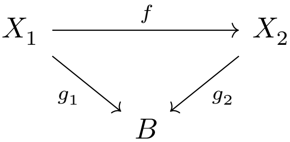


Figure: A real-life sort club

Slice Category

Given a category $\mathcal{C}$ and an object $B \colon \mathcal{C}$, the *slice category* $\mathcal{C}/B$ has:

- As objects pairs (X, g) where $X \colon \mathcal{C}$ and $g \colon X \to B$.
- As morphisms $(X_1, g_1) \to (X_2, g_2)$ $\mathcal{C}$-maps $f \colon X_1 \to X_2$ s.t. $g_2 \circ f = g_1$, i.e. the following diagram commutes:



$\{A^n\}_{n \in \mathbb{N}} \sim (A^*, \text{length}) \colon \text{Set}/\mathbb{N}$
$A^n \sim \text{length}^{-1}(n);\ \text{length}(x \in A^n) = n$

Axiomatic Multiset HIT

```
{-# OPTIONS --cubical #-}
open import Cubical.HITs.FiniteMultiset
```

- Axiomatic multiset:

```
data FMSet (A : Type ℓ) : Type ℓ where
  []    : FMSet A
  _::_  : (x : A) → (xs : FMSet A) → FMSet A
  comm  : ∀ x y xs → x :: y :: xs ≡ y :: x :: xs
```

Element-Indexed List Base Functor

```
data L {l} {A : Type l} (r : FMSet A → Type l) : FMSet A → Type l where
  []  : L r []
  _∷_ : {g : FMSet A} (x : A) → (r g) → L r (x ∷ g)

foldL : {l : Level} → {A : Type l} → {r : FMSet A → Type l} → {g : FMSet A} →
  ({g₂ : FMSet A} → L r g₂ → r g₂) → μL g → r g
foldL alg [] = alg []
foldL alg (x ∷ xs) = alg (x ∷ foldL alg xs)
```

Ordered Element-Indexed List Base Functor

```
data O (r : FMSet A → Type ℓ) : FMSet A → Type ℓ where
  []  : O r []
  _⋊_ : {g : FMSet A} (x : A) → (rg : r g) → All (x ≤_) g → O r (x ∷ g)

data νO : FMSet A → Type ℓ where
  []  : νO []
  _⋊_ : {g : FMSet A} (x : A) → (rg : νO g) → All (x ≤_) g → νO (x ∷ g)
```

Unfolding into an inductive datatype

```
unfoldO : {r : FMSet A → Type ℓ} → {g : FMSet A} →
  ({g₂ : FMSet A} → r g₂ → O r g₂) → r g → νO g
unfoldO grow seed with grow seed
... | [] = []
... | (x ⋊ seed') prf = (x ⋊ unfoldO grow seed') prf
```

Consulting Agda's termination checker

```
> agda -v term:5 DistrLaw.lagda
...
kept call from DistrLaw.with-240 ((ℓ)) ((A)) ((_≤_)) ≤-Toset total≤ ((r))
 (x ∷ g) grow ((seed)) (_⋊_ g x seed′ prf)
  to DistrLaw.unfoldO (ℓ) (A) (_≤_) (≤-Toset) (total≤) (r) (g) (grow) (seed′)
  call matrix (with guardedness):
     =    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    =    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    =    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?   -1    ?    ?   -1
     ?    ?    ?    ?    ?    ?    ?    ?    =    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?   -1
{DistrLaw.unfoldO} does termination check
```

Help??

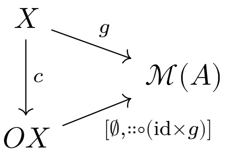
Consulting Agda's termination checker

```
> agda -v term:5 DistrLaw.lagda
...
kept call from DistrLaw.with-240 ((ℓ)) ((A)) ((_≤_)) ≤-Toset total≤ ((r))
 (x :: g) grow ((seed)) (_⋊_ g x seed′ prf)
  to DistrLaw.unfoldO (ℓ) (A) (_≤_) (≤-Toset) (total≤) (r) (g) (grow) (seed′)
  call matrix (with guardedness):
     =    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    =    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    =    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?    ?
     ?    ?    ?    ?    ?    ?    ?   -1    ?    ?   -1
     ?    ?    ?    ?    ?    ?    ?    ?    =    ?    ?
     ?    ?    ?    ?    ?    ?    ?    ?    ?    ?   -1
{DistrLaw.unfoldO} does termination check
```

Help??

Well-foundedness of O-Coalgebras

$$
\begin{aligned}
c &: (X, g) \to O(X, g) \\
c^n &: X \to 1 + X \\
c^0(x) &:= \mathsf{inr}(x) \\
c^{n+1}(x) &:= \begin{cases} \mathsf{inl}(\star) & c^n(x) = \mathsf{inl}(\star) \\ c(y) & c^n(x) = \mathsf{inr}(y) \end{cases}
\end{aligned}
$$



- c well-founded: $\forall x \in X.\, \exists n.\, c^n(x) = \mathsf{inl}(\star)$
- Idea: Use g as a ranking function into well-order $(\mathcal{M}(A), \subset)$
- Case $c(x) = \mathsf{inr}(a, r)$: $g(x) = a :: g(r) \Rightarrow g(r) \subset g(x)$ □

Distributive law

```
open IsToset ≤-Toset
σ : {g : FMSet A} {r : FMSet A → Type ℓ} → L (O r) g → O (L r) g
σ [] = []
σ (x ∷ []) = (x ⋊ []) []
σ (x ∷ (x₁ ⋊ rg) x₂) with total≤ x x₁
...| inl x≤x₁ = (x ⋊ (x₁ ∷ rg)) (x≤x₁ ∷ (≤-to-# is-trans x≤x₁ x₂))
...| inr x₁≤x = subst (O (L _)) (comm _ _ _) ((x₁ ⋊ (x ∷ rg)) (x₁≤x ∷ x₂))
```

- `comm  : ∀ x y xs → x ∷ y ∷ xs ≡ y ∷ x ∷ xs`
  `subst : (B : A → Type ℓ') (p : x ≡ y) → B x → B y`

Intrinsically verified sorting

```
insSort    : {g : FMSet A} → μL g → νO g
bubblesort : {g : FMSet A} → μL g → νO g
insSort    = foldL (unfoldO (σ ∘ mapL outO))
bubblesort = unfoldO (foldL (mapO inL ∘ σ))
```