# Structured Traversals for (Mutually) Recursive Algebraic Datatypes

Cass Alexandru

October 5, 2022

Presentation generated from `.lhs` sources[1] using `lhs2TeX`

[1] `https://github.com/cxandru/talk-multirec`

# Context & Conventions

- Language: Haskell, with numerous language extensions
- Syntactic (e.g. `LambdaCase`)
- Clarifying (e.g. `TypeApplications`, `KindSignatures`)
- Limited dependent types & type-level programming (e.g. `DataKinds`), for mutual recursion

# Context & Conventions

- Language: Haskell, with numerous language extensions
- Syntactic (e.g. `LambdaCase`)
- Clarifying (e.g. `TypeApplications`, `KindSignatures`)
- Limited dependent types & type-level programming (e.g. `DataKinds`), for mutual recursion

```
{-# LANGUAGE LambdaCase #-}
foo :: [a] → b
foo = λcase
  [] → ...
  (x:xs) → ...
```

# Context & Conventions

- Language: Haskell, with numerous language extensions
- Syntactic (e.g. `LambdaCase`)
- Clarifying (e.g. `TypeApplications`, `KindSignatures`)
- Limited dependent types & type-level programming (e.g. `DataKinds`), for mutual recursion

```
{-# LANGUAGE LambdaCase #-}
foo :: [a] → b
foo = λcase
  [] → ...
  (x:xs) → ...
```

- Composition in diagrammatic order: $f ; g$ reads "$f$, then $g$"
- Haskell: `f .> g`

# Structure

```
length :: [a] → Int
length = λcase
[] → 0
(_:xs) → 1 + length xs

filter :: (a → Bool) → [a] → [a]
filter p = go where
  go [] = []
  go (x:xs) = if p x then [x] else [] ++ go xs
```

```
length :: [a] → Int
length = λcase
[] → 0
(_:xs) → 1 + length xs

filter :: (a → Bool) → [a] → [a]
filter p = go where
  go [] = []
  go (x:xs) = if p x then [x] else [] ++ go xs
```

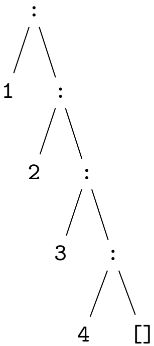
```
length :: [a] → Int
length = λcase
[] → 0
(_:xs) → 1 + length xs
```

```
filter :: (a → Bool) → [a] → [a]
filter p = go where
  go [] = []
  go (x:xs) = if p x then [x] else [] ++ go xs
```

- List Design pattern?
- Design Patterns are a poor person's abstraction
- Recognize common structure & find correct abstract notion

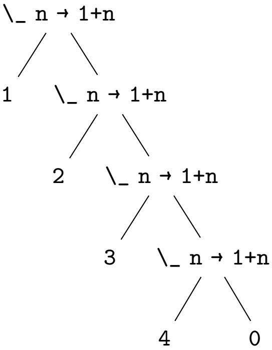
# Functions replace constructors

List

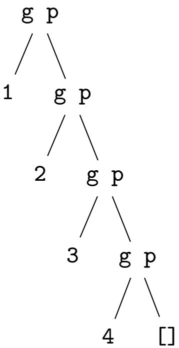



Traversals

`length`



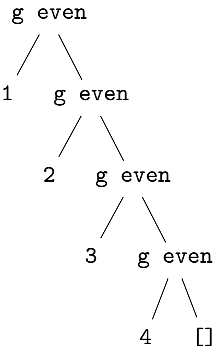

`filter p`




```
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
```



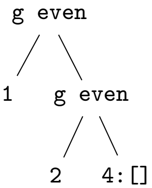
# Example Evaluation of `filter even`

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

```
  g even
   /  \
  1   g even
       /  \
      2   g even
           /  \
          3   4:[]
```

# 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
```

```
  g even
   /  \
1    2:4:[]
```

# Example Evaluation of `filter even`

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

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

# Insight

Even though the functions were defined recursively, their behaviour can be understood non-recursively as simply replacing the two constructors of the datatype `[a]` by functions of the same arity.

```haskell
data List a = Nil | Cons a (List a)
```

GADT Syntax:

```
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)
```

GADT Syntax:

```
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)

list :: b → (a → b → b) → List a → b
list nil cons = fold where
  fold Nil = nil
  fold (x `Cons` xs) = x `cons` fold xs
```

GADT Syntax:

```
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)

list :: b → (a → b → b) → List a → b
list nil cons = fold where
  fold Nil = nil
  fold (x `Cons` xs) = x `cons` fold xs
```

GADT Syntax:

```
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)

list :: b → (a → b → b) → List a → b
list nil cons = fold where
  fold Nil = nil
  fold (x `Cons` xs) = x `cons` fold xs
```

GADT Syntax:

```
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)

list :: b → (a → b → b) → List a → b
list nil cons = fold where
  fold Nil = nil
  fold (x ‘Cons‘ xs) = x ‘cons‘ fold xs

length’ = list 0 (\_ n → 1+n)

filter’ p = list []
  (λx xs → (if p x then [x] else []) ++ xs)
```

# Now for Expressions

```
data Expr where
  Lit :: Int → Expr
  Plus :: Expr → Expr → Expr

expr :: (Int → b) → (b → b → b) → Expr → b
expr lit plus = fold where
  fold (Lit i) = lit i
  fold (l `Plus` r) = (fold l) `plus` (fold r)
```

- We want to define a polytypic "fold", subsuming `list`, `expr`, which encapsulates the whole "replace constructors with functions" pattern
- We need a deeper understanding of what the datatypes we are working with *are*
- $\rightsquigarrow$ Introduce a little ~~Anarchy~~Category Theory

# Category

A category $\mathcal{C}$ consists of collections $\mathcal{C}_0$ of objects and $\mathcal{C}_1$ of morphisms (or arrows) between them, with the following structure:

- For every two arrows $f : A \to B$, $g : B \to C$, there is a composite arrow $f; g : A \to C$
- For every object $A : \mathcal{C}_0$ there is an identity morphism $1_A : A \to A$
- Such that the following hold:
  - Composition is associative, that is: $f; (g; h) = (f; g); h$.
  - Composition satisfies unit laws: For every $f : A \to B$. $\quad 1_A; f = f,\ f; 1_B = f$.

# Isomorphisms

Given a category $\mathcal{C}$ and two objects $A, B : \mathcal{C}_0$, we say $A$ and $B$ are isomorphic via $f : A \to B$, if there exists a $g : B \to A$ which is both a left- and right-inverse:

$$
\begin{tikzcd}
 & B \arrow[dr, "g"] & \\
A \arrow[ur, "f"] \arrow[rr, "1_A"] & & A
\end{tikzcd}
\qquad
\begin{tikzcd}
 & A \arrow[dr, "f"] & \\
B \arrow[ur, "g"] \arrow[rr, "1_B"] & & B
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[loop left, "1_A"] \arrow[r, bend left, "f"] & B \arrow[l, bend left, "g"] \arrow[loop right, "1_B"]
\end{tikzcd}
$$

# Functors

Let $\mathcal{C}, \mathcal{D}$ be Categories. A *Functor* $F$ is a pair of maps $(F_0, F_1)$, on the objects and morphisms of the category respectively, such that commuting diagrams are preserved, e.g.:

$$
\begin{tikzcd}
 & B \arrow[dr, "g"] & \\
A \arrow[ur, "f"] \arrow[rr, "h"] & & C
\end{tikzcd}
\quad \overset{F}{\Rightarrow} \quad
\begin{tikzcd}
 & F_0 B \arrow[dr, "F_1 g"] & \\
F_0 A \arrow[ur, "F_1 f"] \arrow[rr, "F_1 h"] & & F_0 C
\end{tikzcd}
$$

In particular, this means identities & composition are preserved.

# Functors

Let $\mathcal{C}, \mathcal{D}$ be Categories. A *Functor F* is a pair of maps $(F_0, F_1)$, on the objects and morphisms of the category respectively, such that commuting diagrams are preserved, e.g.:

$$\begin{tikzcd} & B \arrow[dr, "g"] & \\ A \arrow[ur, "f"] \arrow[rr, "h"] & & C \end{tikzcd} \quad \xRightarrow{F} \quad \begin{tikzcd} & F_0 B \arrow[dr, "F_1 g"] & \\ F_0 A \arrow[ur, "F_1 f"] \arrow[rr, "F_1 h"] & & F_0 C \end{tikzcd}$$

In particular, this means identities & composition are preserved. We will often only write out the definition of a functor on objects.

# Functors

Let $\mathcal{C}, \mathcal{D}$ be Categories. A *Functor* $F$ is a pair of maps $(F_0, F_1)$, on the objects and morphisms of the category respectively, such that commuting diagrams are preserved, e.g.:

$$\begin{tikzcd} & B \arrow[dr, "g"] & \\ A \arrow[ur, "f"] \arrow[rr, "h"'] & & C \end{tikzcd} \quad \xRightarrow{F} \quad \begin{tikzcd} & F_0 B \arrow[dr, "F_1 g"] & \\ F_0 A \arrow[ur, "F_1 f"] \arrow[rr, "F_1 h"'] & & F_0 C \end{tikzcd}$$

In particular, this means identities & composition are preserved. We will often only write out the definition of a functor on objects. When $\mathcal{C} = \mathcal{D}$, we say $F$ is an *Endofunctor*.

# Building the Functor kit

- Identity ($IX := X$) is a functor.
- Constant-to-$A$ ($K_A X := A$), for $A : C_0$, is a functor.

Categories can have products ($\times_{\mathcal{C}}$) and/or coproducts ($+_{\mathcal{C}}$). Think of coproducts as indexed unions in our case. Then if $F, G$ are functors so are:

- $(F \times G)X := FX \times GX$
- $(F + G)X := FX + GX$

# Algebra

Let $F : \mathcal{C} \to \mathcal{C}$ be an endofunctor $A : \mathcal{C}$, $\varphi : FA \to A$. Then $FA \xrightarrow{\varphi} A$ (or $(A, \varphi)$) is an *F-Algebra*, and $A$ its *Carrier*.

Algebra

$$\begin{tikzcd} FA \arrow[d, "\varphi"] \\ A \end{tikzcd}$$

Algebra-Hom:

$(A, \varphi) \xrightarrow{f} (B, \psi)$

$$\begin{tikzcd} FA \arrow[r, "Ff"] \arrow[d, "\varphi"] & FB \arrow[d, "\psi"] \\ A \arrow[r, "f"] & B \end{tikzcd}$$

Initial Algebra: $(A, \alpha)$ s.t. $\forall (B, \psi)$.

$$\begin{tikzcd} FA \arrow[r, dashed, "Fh"] \arrow[d, "\alpha"] & FB \arrow[d, "\psi"] \\ A \arrow[r, dashed, "h"] & B \end{tikzcd}$$

# Lambek's Lemma

If $F$ has an initial algebra $(A, \alpha)$, then $A$ is isomorphic to $FA$ via $\alpha$. Proof (We show only $h; \alpha = 1_A$): Consider the algebra $(FA, F\alpha)$:

$$\begin{tikzcd} FA \arrow[d, "\alpha"] \\ A \end{tikzcd}$$

# Lambek's Lemma

If $F$ has an initial algebra $(A, \alpha)$, then $A$ is isomorphic to $FA$ via $\alpha$. Proof (We show only $h; \alpha = 1_A$): Consider the algebra $(FA, F\alpha)$:

$$\begin{tikzcd} FA \arrow[r, dashed, "Fh"] \arrow[d, "\alpha"] & F(FA) \arrow[d, "F\alpha"] \\ A \arrow[r, dashed, "h"] & FA \end{tikzcd}$$

# Lambek's Lemma

If $F$ has an initial algebra $(A, \alpha)$, then $A$ is isomorphic to $FA$ via $\alpha$. Proof (We show only $h; \alpha = 1_A$): Consider the algebra $(FA, F\alpha)$:

$$\begin{tikzcd}
FA \arrow[r, dashed, "Fh"] \arrow[d, "\alpha"] & F(FA) \arrow[r, "F\alpha"] \arrow[d, "F\alpha"] & FA \arrow[d, "\alpha"] \\
A \arrow[r, dashed, "h"] & FA \arrow[r, "\alpha"] & A
\end{tikzcd}$$

# Lambek's Lemma

If $F$ has an initial algebra $(A, \alpha)$, then $A$ is isomorphic to $FA$ via $\alpha$. Proof (We show only $h; \alpha = 1_A$): Consider the algebra $(FA, F\alpha)$:

$$\begin{tikzcd}
FA \arrow[r, dashed, "Fh"] \arrow[d, "\alpha"] \arrow[rr, bend left, "F1_A"] & F(FA) \arrow[r, "F\alpha"] \arrow[d, "F\alpha"] & FA \arrow[d, "\alpha"] \\
A \arrow[r, dashed, "h"] \arrow[rr, bend right, "1_A"'] & FA \arrow[r, "\alpha"] & A
\end{tikzcd}$$

# Fixed Point

The carrier $A$ of the initial algebra $(A, \alpha)$ of a functor $F$ is a least fixed point of $F$. Least, that is, in that there is a morphism from it to any other algebra, by initiality.

# Fixed Point

The carrier $A$ of the initial algebra $(A, \alpha)$ of a functor $F$ is a least fixed point of $F$. Least, that is, in that there is a morphism from it to any other algebra, by initiality.

$$\begin{tikzcd}
FA \arrow[r, dashed, "Fh"] \arrow[d, shift left, "\alpha"] \arrow[dr, phantom, "\circlearrowleft"] & FB \arrow[d, "\psi"] \\
A \arrow[u, shift left, "\alpha^{-1}"] \arrow[r, dashed, "h"] & B
\end{tikzcd}$$

We get a recursive definition for h: $h = \alpha^{-1}; Fh; \psi$

# Back Again

We will now return to the motivating problem and see how what we have just learned is applicable to its solution. In particular, we will see that:

- Our recursive datatypes correspond to fixpoints of associated "structural/base" functors, and are carriers of their initial algebras
- The non-recursive business logic of the traversals, that is, the functions to replace the constructors, correspond to algebras over this functor
- The unique morphism $h$ we get, given $(B, \psi)$, is the polytypic *fold* we were looking for

# Functors in Haskell

- Haskell can be seen as a category, where the objects are types and the arrows functions between them.
- Endofunctors in Haskell can be implemented as type constructors ($* \to *$)
- Definition on arrows `a → b` via typeclass *Functor*, defining function `fmap`
- $F_1(h : A \to B) : F_0 A \to F_0 B \quad \sim$
  `fmap @f (h :: a → b) :: f a → f b`

# Structural Functors

- We obtain the structural functors for our datatypes by factoring the recursion out of their definition, then adding it back in via a fixed-point operator.
- We compare with how this can be done on the value level:

```
type ℤ = Int
-- Recursive Definition:
fac :: ℤ → ℤ
fac = λcase
  0 → 1
  n → n * fac (n-1)
-- Fixpoint operator:
fix :: ∀ a· (a → a) → a
fix f = f (fix f)
-- Factoring fac:
fac' = fix @(ℤ → ℤ) facF where
  facF :: (ℤ → ℤ) → (ℤ → ℤ)
  facF rec n = n * rec (n-1)
```

```
-- Recursive Definition:
type List :: * → *
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)

-- Fixpoint operator (typelevel)
data family Fix' (f :: k → k) :: k
newtype instance Fix' f = In' (f (Fix' f))
newtype instance Fix' f ix = IIn (f (Fix' f) ix)
-- Factoring List
type List'' = Fix' @(* → *) ListF'
type ListF' :: (* → *) → * → *
data ListF' rec a = NilF' | ConsF' a (rec a)
```

# Structural Functors

- We obtain the structural functors for our datatypes by factoring the recursion out of their definition, then adding it back in via a fixed-point operator.
- We compare with how this can be done on the value level:

This encoding of typelevel recursion is quite close to term-level recursion. *However*, we have to deal with much more type-level programming and this approach deviates from that of the original version of this talk, and the author did not have enough time to explore it in full.

```
type ℤ = Int
-- Recursive Definition:
fac :: ℤ → ℤ
fac = λcase
  0 → 1
  n → n * fac (n-1)
-- Fixpoint operator:
fix :: ∀ a· (a → a) → a
fix f = f (fix f)
-- Factoring fac:
fac' = fix @(ℤ → ℤ) facF where
  facF :: (ℤ → ℤ) → (ℤ → ℤ)
  facF rec n = n * rec (n-1)
```

```
-- Recursive Definition
type List :: * → *
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)

-- Fixpoint operator (typelevel)
data family Fix' (f :: k → k) :: k
newtype instance Fix' f = In' (f (Fix' f))
newtype instance Fix' f ix = IIn (f (Fix' f) ix)
-- Factoring List
type List'' = Fix' @(* → *) ListF'
type ListF' :: (* → *) → * → *
data ListF' rec a = NilF' | ConsF' a (rec a)
```

# Structural Functors

- We obtain the structural functors for our datatypes by factoring the recursion out of their definition, then adding it back in via a fixed-point operator.
- We compare with how this can be done on the value level:

```
type ℤ = Int
-- Recursive Definition:
fac :: ℤ → ℤ
fac = λcase
  0 → 1
  n → n * fac (n-1)
-- Fixpoint operator:
fix :: ∀ a· (a → a) → a
fix f = f (fix f)
-- Factoring fac:
fac' = fix @(ℤ → ℤ) facF where
  facF :: (ℤ → ℤ) → (ℤ → ℤ)
  facF rec n = n * rec (n-1)
```

```
-- Recursive Definition:
type List :: * → *
data List a where
  Nil :: List a
  Cons :: a → (List a) → (List a)

-- Fixpoint operator (typelevel)
type Fix :: (* → *) → *
newtype Fix f = In (f (Fix f))
-- Factoring List
type List' a = Fix (ListF a)
data ListF a l =
  NilF | ConsF a l
```

# Business Logic as Algebra

So far we've been passing the functions meant to replace the constructors of a type as separate arguments to the fold for each type. Via uncurrying we can pass those functions as a tuple.

# Business Logic as Algebra

So far we've been passing the functions meant to replace the constructors of a type as separate arguments to the fold for each type. Via uncurrying we can pass those functions as a tuple.

We can transform a tuple of functions meant to replace the constructors of a type as an *algebra* using the transformation $B^A \times B^C \sim B^{A+C}$.

# Business Logic as Algebra

So far we've been passing the functions meant to replace the constructors of a type as separate arguments to the fold for each type. Via uncurrying we can pass those functions as a tuple.

We can transform a tuple of functions meant to replace the constructors of a type as an *algebra* using the transformation $B^A \times B^C \sim B^{A+C}$.

The exponential $B^A$ is the function type $A \to B$, and if $A, C$ are constructors, then the type itself is the sum of its constructors $(A + C)$.

# Business Logic as Algebra

So far we've been passing the functions meant to replace the constructors of a type as separate arguments to the fold for each type. Via uncurrying we can pass those functions as a tuple.
We can transform a tuple of functions meant to replace the constructors of a type as an *algebra* using the transformation $B^A \times B^C \sim B^{A+C}$.
The exponential $B^A$ is the function type $A \to B$, and if $A, C$ are constructors, then the type itself is the sum of its constructors $(A + C)$.

```
type Algebra f a = f a → a
listBL :: b → (a → b → b) → Algebra (ListF a) b
listBL nil cons = λcase
  NilF → nil
  x ‘ConsF‘ b → x ‘cons‘ b
```

# Defining a Functor instance

- Do we have to manually define Functor instances?
- Not if working with *polynomial functors*
- Recall the functor kit from theory

```
type ListFG a l = (K () :+: K a :×: I) l
inG :: ListF a l → ListFG a l
inG = λcase
  NilF → InL $ K ()
  a `ConsF` l → InR (K a :×: I l)
outG :: ListFG a l → ListF a l
outG = λcase
  InL _ → NilF
  InR (K a :×: I l) → a `ConsF` l
instance Functor (ListF a) where
  fmap f = inG .> fmap f .> outG
```

# Generic/Polytypic Programming

- `fmap` is a polytypic function
- Our iso to the generic representation is still boilerplate, though
- Some essentially polytypic functions derivable in Haskell, (e.g. `(≡)`, via **`deriving`** *Eq*), `fmap` using
  {-# LANGUAGE DeriveFunctor #-}
- Whole topic on its own

# Cigar!

Recall the recursive definition $h : A \to B = \alpha^{-1}; Fh; \psi$. To implement it, we only still lack $\alpha^{-1} : A \to FA$. Recall that a carrier of the initial algebra for functor $F$ is $Fix\ F$. `unFix :: Fix f → f (Fix f)` is easily defined:

```
unFix (In f) = f
```

- Finally, all parts are assembled for our polytypic *fold* function!
- Also called a *catamorphism*, like "cataclysm": Collapses a structure into a value (which of course can also again be a structure)

```
cata :: Functor f => Algebra f b → (Fix f) → b
cata ψ = unFix .> fmap (cata ψ) .> ψ
```

# Structure

# A simple AST

Consider the datatype

```
data Expr = Lit Int | Var Char | Plus Expr Expr |
  LetIn Decl Expr
data Decl = Bind Char Expr | Seq Decl Decl
```

Consider an example expression

```
e1 :: Expr
e1 = LetIn
  (('x' `Bind` (Lit 3))
  `Seq`
   ('y' `Bind` (Lit 4))
  )  (Var 'x') `Plus` (Var 'y')
```

# Mutual Recursion

- $\mathit{Expr}$, $\mathit{Decl}$ are a mutually recursive data family. So it is unclear how we should factor out the correct "structural functor".
- As we did with fixed points, we will take inspiration from how such situations can be handled on the value level.

# Mutual Recursion

- *Expr*, *Decl* are a mutually recursive data family. So it is unclear how we should factor out the correct "structural functor".
- As we did with fixed points, we will take inspiration from how such situations can be handled on the value level.

```
foo = let
  even = λcase 0 → True; n → odd (n-1)
  odd = λcase 0 → False; n → even (n-1)
  in even 42
```

# Mutual Recursion

- *Expr*, *Decl* are a mutually recursive data family. So it is unclear how we should factor out the correct "structural functor".
- As we did with fixed points, we will take inspiration from how such situations can be handled on the value level.

```
foo = let
  even = λcase 0 → True; n → odd (n-1)
  odd = λcase 0 → False; n → even (n-1)
  in even 42
```

- What if the language provides only a singly recursive `let`?
- Use a tupling trick:

```
bar = let (even, odd) = (
            λcase 0 → True; n → odd (n-1),
            λcase 0 → False; n → even (n-1)
            )
  in even 42
```

# Product category

For two categories $\mathcal{C}$ and $\mathcal{D}$, the *product category* $C \times D$ consists of the following:

- as objects pairs $(C, D)$, $C : \mathcal{C}_0$, $D : \mathcal{D}_0$
- as morphisms pairs $(f : X \to Y, g : A \to B)$, $f : \mathcal{C}_1$, $g : \mathcal{D}_1$, such that composition is defined componentwise

# Functors

In Haskell *Functor* represents functors in Hask, so we cannot use it for functors in the product category Hask×Hask. We can write some pseudocode though, for how our structural functor ought to look:

```
data ExprF (e,d) = (
   Lit Int | Var Char | Plus e e | LetIn d e
 , Bind Char e | Seq d d
                   )
```

What we need is another way to represent tuples. A preliminary observation is that an $n$-tuple $A^n$ can be seen as a function from the finite set of cardinality $n$ to $A$: $n \rightarrow A$.

# Remodeling

The functor for our example family has kind `(2 → *) → (2 → *)`. First we should define this kind 2.
We can define define an Enum with two accessors for our expr/decl family, and use it as a kind (lifting its constructors to singleton types), using the GHC extension `DataKinds`

# Remodeling

```
data Tag = E | D
data family ASTF1 :: (Tag → *) → (Tag → *)
data instance ASTF1 c E =
  Lit1 Int | Var1 Char | Plus1 (c E) (c E) |
  LetIn1 (c D) (c E)
data instance ASTF1 c D =
  Bind1 Char (c E) | Seq1 (c D) (c D)
```

# Remodeling

```
data Tag = E | D
data family ASTF1 :: (Tag → *) → (Tag → *)
data instance ASTF1 c E =
  Lit1 Int | Var1 Char | Plus1 (c E) (c E) |
  LetIn1 (c D) (c E)
data instance ASTF1 c D =
  Bind1 Char (c E) | Seq1 (c D) (c D)
```

The type of *ASTF1* can be read as “given a table of types to be used in the recursive positions of Expr and Decl, respectively, return a table, where at tag E you find the configured Expr type, at D Decl”.

# Remodeling

```
data Tag = E | D
data family ASTF1 :: (Tag → *) → (Tag → *)
data instance ASTF1 c E =
  Lit1 Int | Var1 Char | Plus1 (c E) (c E) |
  LetIn1 (c D) (c E)
data instance ASTF1 c D =
  Bind1 Char (c E) | Seq1 (c D) (c D)
```

The two instances define the the component of the structural functor[2] for *Expr* and *Decl* respectively. The tags on the rhs are used to access the components of c. If c ≅ (e,d), then c *E* = e, c *D* = d.

[2] Actually, for technical reasons you need to write a GADT. I used type family syntax here since I believe it makes it clearer what is going on

# IFunctor

We will need a new typeclass, *IFunctor* (standing for *indexed functor*), for functors of the discussed shape. What is the type of morphisms in (k → ∗)? We saw that in the product category, morphisms were of the form $(f, g)$. Generalizing, we are looking at a collection of maps of the form:

$$\begin{array}{cccc} A_1 & A_2 & \cdots & A_k \\ \downarrow{\scriptstyle f_1} & \downarrow{\scriptstyle f_2} & & \downarrow{\scriptstyle f_k} \\ A'_1 & A'_2 & \cdots & A'_k \end{array}$$

```
class IFunctor (f :: (k → ∗) → (k → ∗)) where
  iFmap :: (∀ (i :: k)· r i → r' i) → f r ix → f r' ix
```

# Fixpoint & Algebra

Nothing very fancy happens here. Both result in a family and are defined pointwise via `ix ::k`.

```
type IFix :: ((k → *) → (k → *)) → (k → *)
newtype IFix f (ix :: k) =
  IIn (f (IFix f) ix)

iUnFix :: IFix f ix → f (IFix f) ix
iUnFix (IIn f) = f

type Algebra f r ix = f r ix → r ix
```

# Cata

```
iCata :: ∀ k (f :: (k → *) → (k → *))
  (r :: (k → *))
  (ix :: k)· IFunctor f  =>
  (∀ (i :: k)· Algebra f r i) → (IFix f ix) → r ix
iCata ψ = iUnFix .> iFmap (iCata ψ) .> ψ
```

# Demo!

# Structure

# Remarks

- There are a wealth of topics in the main paper and related literature not broached in this talk
- A small taste:
    - *unfolds*, producing datastructures, and more schemes
    - fusion laws derived from Category Theory

# Conclusion

- using Category Theory, we were able to give a uniform implementation for a whole class of traversals
- We generalized from single to mutual recursion, noting that we didn't need any additional categorical notions
- Implementation details for mutual recursion are quite messy - boilerplate still exists, but ergonomics & reach of recursion schemes have been increasing since their theoretical beginnings.