Recursion Schemes - Why, How and More

Sergey Vinokurov

serg.foo@gmail.com, @5ergv

2017-11-17

Outline

Introduction

Recursive datatypes 101 - Lists

Recursive datatypes 102 - taking recursion out

On to compilers

Compilers - annotating expressions

Compilers - adding variables

Compilers - adding a *bit more* type safety

Wrapping up

What

An FP pattern for ~~working with ASTs~~ *writing compilers*

What

An FP pattern for ~~working with ASTs~~ *writing compilers*

Get recursion out - can change it later

What

An FP pattern for ~~working with ASTs~~ *writing compilers*

Get recursion out - can change it later

Let’s go, there’s lot to cover. Ask questions!

What's in a list?

List is either empty or an element consed onto another list.

```
-- Recursive datatype!
data List a =
    Nil
  | Cons a (List a)
  deriving (Show)
```

Processing lists

```
-- Let's multiply all list elements together.
prodList :: List Double → Double
prodList Nil           = 1
prodList (Cons x xs) = x * prodList xs

-- And compute length
lengthList :: List a → Int
lengthList Nil           = 0
lengthList (Cons _ xs) = 1 + lengthList xs
```

Expected output

```
prodList   (Cons 1 (Cons 2 (Cons 3 Nil))) ⇒  6.0
lengthList (Cons 1 (Cons 2 (Cons 3 Nil))) ⇒  3
```

Explicit recursion leaves out without modularity

What if we want to fuse `prodList` and `lengthList`?

Why? - Say, we'd like to do only *one* traversal over a list.

Explicit recursion leaves out without modularity

What if we want to fuse `prodList` and `lengthList`?

Why? - Say, we'd like to do only *one* traversal over a list.

Can we reuse already written `prodList` and `lengthList`

Explicit recursion leaves out without modularity

What if we want to fuse `prodList` and `lengthList`?

Why? - Say, we'd like to do only *one* traversal over a list.

Can we reuse already written `prodList` and `lengthList`

Fusing two list functions

Let's see...

Expected output

`computeBoth` $(\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}))) \Rightarrow$ `(3,6.0)`

Fusing two list functions

Let's see...

Expected output
`computeBoth` $(\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}))) \Rightarrow$ `(3,6.0)`

No way to combine `prodList` and `lengthList`. Each function does it's own traversal and we cannot alter that.

Fusing two list functions

Let's see...

Expected output

computeBoth (*Cons* 1 (*Cons* 2 (*Cons* 3 *Nil*))) ⇒ (3,6.0)

No way to combine `prodList` and `lengthList`. Each function does it's own traversal and we cannot alter that.

```
computeBoth :: List Double → (Int, Double)
computeBoth Nil          = (0, 1)
computeBoth (Cons x xs) =
  let (len, pr) = computeBoth xs in (len + 1, x * pr)
```

Fusing two list functions

Let's see...

Expected output

`computeBoth` $(\mathit{Cons}\ 1\ (\mathit{Cons}\ 2\ (\mathit{Cons}\ 3\ \mathit{Nil}))) \Rightarrow$ `(3,6.0)`

No way to combine `prodList` and `lengthList`. Each function does it's own traversal and we cannot alter that.

```
computeBoth :: List Double → (Int, Double)
computeBoth Nil          = (0, 1)
computeBoth (Cons x xs) =
  let (len, pr) = computeBoth xs in (len + 1, x * pr)
```

NB Even `foldr` does not help us here (homework: convince yourself that it's the case).

The path forward

Key idea: split datatypes into recursive and non-recursive part.

Let's try it with lists. I'll introduce non-recursive part first.

Just go ahead and replace all recursive occurrences with new parameter.

```haskell
-- Base functor that captures the shape of our type.
data ListF a r =
    NilF
  | ConsF a r
  deriving (Show, Functor)
```

The Base Functor

What will happen when we start consing like we did before?

```
-- Base functor that captures the shape of our type.
data ListF a r =
    NilF
  | ConsF a r
  deriving (Show, Functor)
```

```
NilF :: ListF a r
ConsF () NilF :: ListF () (ListF a r)
ConsF () (ConsF () NilF) :: ListF () (ListF () (ListF a r))
```

A way to use base functor for lists

Can use this raw functor to specify things like ‘list no longer than 3’.

```
-- Base functor that captures the shape of our type.
data ListF a r =
    NilF
  | ConsF a r
  deriving (Show, Functor)

type List3 a = ListF a (ListF a (ListF a ()))

nullList3 :: List3 a → Bool
nullList3 NilF = True
nullList3 _    = False
```

We need recursion

Still, $List3$ is not enough.

We can specify a list of any fixed length, but we cannot nest $ListF$'s to get infinitely-long lists - we will get infinite type!

The missing recursion bit

Let's define a type that will add recursion to the *ListF*.

```
-- Recursive part of our datatype.
-- NB use newtype to avoid extra layers and have the same runtime
-- representation as we had before.
newtype Fix f = Fix (f (Fix f))

-- Unwrap one layer
unFix :: Fix f → f (Fix f)
unFix (Fix x) = x
```

Recovering *List*

Fix (*ListF* a) will give us exactly the *List* a we had before!

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

newtype Fix f = Fix (f (Fix f))

data ListF a r = NilF | ConsF a r

-- Convenient type alias that does the job
type List' a = Fix (ListF a)
```

Recovering *List*

Fix (*ListF* a) will give us exactly the *List* a we had before!

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

newtype Fix f = Fix (f (Fix f))

data ListF a r = NilF | ConsF a r

-- Convenient type alias that does the job
type List' a = Fix (ListF a)
```

- *Nil* ⇔ *Fix NilF*
- *Cons* 1 *Nil* ⇔ *Fix* (*ConsF* 1 (*Fix NilF*))
- *Cons* 1 (*Cons* 2 *Nil*) ⇔ *Fix* (*ConsF* 1 (*Fix* (*ConsF* 2 (*Fix NilF*))))

How to work with this

Great, now we can rewrite $List$ a as Fix ($ListF$ a)!!

What now??

How to work with this

Great, now we can rewrite $List$ a as Fix ($ListF$ a)!!

What now??

Let's actually look at recursion schemes.

Our first recursion scheme

The most basic recursion scheme bears proud name *catamorphism*.

The name comes from from the Greek: κατά "downwards" and μορφή "form, shape". (Wikipedia).

One cool bit is that catamorphism's definition can be inferred from its commutative diagram (how cool is that!).

Catamorphism from a commutative diagram

But let's write function's type first. We actually want something this:

```
cata :: (f a → a) → Fix f → a
```

We'll revise this a bit afer looking at the diagram.

For now just note that first argument is called algebra and that catamorphism collapses possibly infinitely many layers with an algebra.

The diagram

```
cata :: (f a → a) → Fix f → a
```

$$
\begin{tikzcd}
\mathtt{f}\ (\mathit{Fix}\ \mathtt{f}) & \mathtt{f}\ \mathtt{a} \\
\mathit{Fix}\ \mathtt{f} \arrow[r, Rightarrow, "\mathtt{cata}\ \mathtt{f}"'] & \mathtt{a}
\end{tikzcd}
$$

The diagram 2

```
cata :: (f a → a) → Fix f → a
```

$$
\begin{tikzcd}[row sep=huge, column sep=huge]
\mathtt{f}\ (\mathit{Fix}\ \mathtt{f}) & & \mathtt{f\ a} \\
\mathit{Fix}\ \mathtt{f} \arrow[u, "\mathtt{unFix}"] \arrow[rr, Rightarrow, "\mathtt{cata\ f}"'] & & \mathtt{a}
\end{tikzcd}
$$

The diagram 3

```
cata :: (f a → a) → Fix f → a
```

$$
\begin{tikzcd}
\mathtt{f}\ (\mathit{Fix}\ \mathtt{f}) & & \mathtt{f\ a} \arrow[dd, "\mathtt{f}"] \\
& & \\
\mathit{Fix}\ \mathtt{f} \arrow[uu, "\mathtt{unFix}"] \arrow[rr, Rightarrow, "\mathtt{cata\ f}"'] & & \mathtt{a}
\end{tikzcd}
$$

The complete diagram

```
fmap :: Functor f => (a → b) → f a → f b

cata :: Functor f => (f a → a) → Fix f → a
```

$$
\begin{tikzcd}
\texttt{f (}\mathit{Fix}\ \texttt{f)} \arrow[r, "\texttt{fmap (cata f)}"] & \texttt{f a} \arrow[d, "\texttt{f}"] \\
\mathit{Fix}\ \texttt{f} \arrow[u, "\texttt{unFix}"] \arrow[r, Rightarrow, "\texttt{cata f}"'] & \texttt{a}
\end{tikzcd}
$$

Read the diagram into a function

```haskell
cata :: Functor f => (f a → a) → Fix f → a
cata alg = go
  where
    go = alg · fmap go · unFix
```

$$\begin{tikzcd}
\mathtt{f}\ (\mathit{Fix}\ \mathtt{f}) \arrow[r, "\mathtt{fmap\ (cata\ f)}"] & \mathtt{f\ a} \arrow[d, "\mathtt{f}"] \\
\mathit{Fix}\ \mathtt{f} \arrow[u, "\mathtt{unFix}"] \arrow[r, "\mathtt{cata\ f}"'] & \mathtt{a}
\end{tikzcd}$$

Let’s compute length with our new tool

```
cata :: Functor f => (f a → a) → Fix f → a

lenAlg :: ListF a Int → Int
lenAlg = \case
  NilF       → 0
  ConsF _ x → x + 1

lengthListF :: List’ a → Int
lengthListF = cata lenAlg
```

Run it

```
lengthListF (Fix (ConsF 3 (Fix (ConsF 4 (Fix NilF)))))
⇒ 2
```

Let's compute product with our new tool

```
cata :: Functor f => (f a → a) → Fix f → a

prodAlg :: ListF Double Double → Double
prodAlg = \case
  NilF       → 1
  ConsF x y → x * y

prodListF :: List' Double → Double
prodListF = cata prodAlg
```

Run it

```
prodListF (Fix (ConsF 3 (Fix (ConsF 4 (Fix NilF)))))
⇒ 12.0
```

On to fusion

```
cata :: Functor f => (f a → a) → Fix f → a
fmap :: Functor f => (a → b) → f a → f b
fst  :: (a, b) → a
snd  :: (a, b) → b

fuseAlgs
  :: Functor f
  => (f a → a)
  →  (f b → b)
  →  (f (a, b) → (a, b))
fuseAlgs f g = \x → (f (fmap fst x), g (fmap snd x))

lenWithProdListF :: List' Double → (Int, Double)
lenWithProdListF = cata (fuseAlgs lenAlg prodAlg)
```

On to fusion

```
cata :: Functor f => (f a → a) → Fix f → a
fmap :: Functor f => (a → b) → f a → f b
fst  :: (a, b) → a
snd  :: (a, b) → b

fuseAlgs
  :: Functor f
  => (f a → a)
  →  (f b → b)
  →  (f (a, b) → (a, b))
fuseAlgs f g = \x → (f (fmap fst x), g (fmap snd x))

lenWithProdListF :: List' Double → (Int, Double)
lenWithProdListF = cata (fuseAlgs lenAlg prodAlg)

lenWithProdListF (Fix (ConsF 3 (Fix (ConsF 4 (Fix NilF)))))
⇒ (2,12.0)
```

Compiler datatypes

Compilers work with *abstract syntax trees*.

We need a simple tree type for illustration purposes.

Compiler datatypes

Compilers work with *abstract syntax trees*.

We need a simple tree type for illustration purposes.

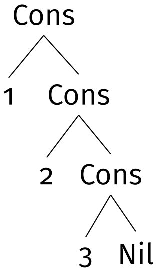
Hutton's Razor - very simple AST to try ideas on.

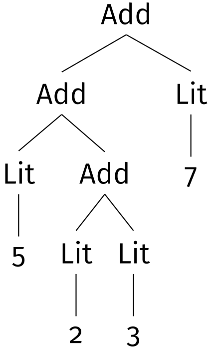
```haskell
data HuttonExpr =
    Lit Double
  | Add HuttonExpr HuttonExpr
```

Compiler datatypes from a visual standpoint

Trees branch, lists are linear.

$Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil))$





$Add\ (Add\ (Lit\ 5)\ (Add\ (Lit\ 2)\ (Lit\ 3)))\ (Lit\ 7)$

What we do with expressions - Evaluate

Let's evaluate *HuttonExpr*

```
evalHutton :: HuttonExpr → Double
evalHutton (Lit n)   = n
evalHutton (Add x y) = evalHutton x + evalHutton y
```

What we do with expressions - Evaluate

Let's evaluate *HuttonExpr*

```
evalHutton :: HuttonExpr → Double
evalHutton (Lit n)   = n
evalHutton (Add x y) = evalHutton x + evalHutton y
```

Run it

```
evalHutton (Add (Add (Lit 5) (Add (Lit 2) (Lit 3))) (Lit 7))
⇒ 17.0
```

What we do with expressions - Evaluate

Let's evaluate *HuttonExpr*

```
evalHutton :: HuttonExpr → Double
evalHutton (Lit n)   = n
evalHutton (Add x y) = evalHutton x + evalHutton y
```

Run it

```
evalHutton (Add (Add (Lit 5) (Add (Lit 2) (Lit 3))) (Lit 7))
⇒ 17.0
```

What we do with expressions - Compute height

Say, we want to place all arguments on a stack when generating code.

We'd like to know the maximum depth of the stack we'll need...

```
depthHutton :: HuttonExpr → Int
depthHutton (Lit _)   = 1
depthHutton (Add x y) =
  1 + (depthHutton x `max` depthHutton y)
```

What we do with expressions - Compute height

Say, we want to place all arguments on a stack when generating code.

We'd like to know the maximum depth of the stack we'll need...

```
depthHutton :: HuttonExpr → Int
depthHutton (Lit _)   = 1
depthHutton (Add x y) =
  1 + (depthHutton x `max` depthHutton y)
```

Run it

```
depthHutton (Add (Add (Lit 5) (Add (Lit 2) (Lit 3))) (Lit 7))
⇒ 4
```

Unfix the *HuttonExpr*

All issues that plagued `prodList` and `lengthList` will show up with `evalHutton` and `depthHutton` as well. Let's fix them

```
data HuttonExprF r =
    LitF Double
  | AddF r r
  deriving (Show, Functor, Foldable)

type HuttonExpr' = Fix HuttonExprF
```

Redefine our functions

```
data HuttonExprF r =
    LitF Double
  | AddF r r
  deriving (Show, Functor, Foldable)

fold :: (Foldable f, Monoid a) => f a → a

depthAlg :: HuttonExprF Int → Int
depthAlg = \case
  LitF _   → 1
  AddF x y → 1 + (x `max` y)
```

Redefine our functions

```
data HuttonExprF r =
    LitF Double
  | AddF r r
  deriving (Show, Functor, Foldable)

fold :: (Foldable f, Monoid a) => f a → a

depthAlg :: HuttonExprF Int → Int
depthAlg = \case
  LitF _   → 1
  AddF x y → 1 + (x `max` y)
```

Monoids simplify algebras a lot

```haskell
newtype SumMonoid = SumMonoid Double deriving (Show)

instance Monoid SumMonoid where
  mempty  = SumMonoid 0
  mappend = coerce ((+) @Double)

evalAlg :: HuttonExprF SumMonoid → SumMonoid
evalAlg = \case
  LitF x → SumMonoid x
  e      → fold e
```

Can fuse those with function we defined before

```
fuseAlgs
  :: Functor f
  => (f a → a)
  →  (f b → b)
  →  (f (a, b) → (a, b))
fuseAlgs f g = \x → (f (fmap fst x), g (fmap snd x))

evalWithDepth :: HuttonExpr’ → (Double, Int)
evalWithDepth = coerce (cata (fuseAlgs evalAlg depthAlg))

 evalWithDepth
   (Fix (AddF
     (Fix (AddF
       (Fix (LitF 5))
       (Fix (AddF (Fix (LitF 2)) (Fix (LitF 3))))))
     (Fix (LitF 7))))
```

$\Rightarrow$ (17.0,4)

Annotating all nodes with a common value

Say, we are parsing our expressions from a file.

Each expression has position that we'd like to keep around.

What do we usually do to add positions to each node?

```
-- Just an example
newtype Line     = Line Int
newtype Column   = Column Int
data    Position = Position String Line Column

-- Try adding annotations here
data HuttonExpr =
    Lit Double
  | Add HuttonExpr HuttonExpr
```

How to add position without recursion schemes - simple solution

For one, we can just go ahead and annotate each constructor

```haskell
data Position = ...

data HuttonExpr =
    Lit Position Double
  | Add Position HuttonExpr HuttonExpr
```

How to add position without recursion schemes - simple solution

For one, we can just go ahead and annotate each constructor

```haskell
data Position = ...

data HuttonExpr =
    Lit Position Double
  | Add Position HuttonExpr HuttonExpr
```

Pro: very simple

How to add position without recursion schemes - simple solution

For one, we can just go ahead and annotate each constructor

```
data Position = ...

data HuttonExpr =
    Lit Position Double
  | Add Position HuttonExpr HuttonExpr
```

Pro: very simple
Cons:

- Not really feasible if expression type has nearly *100 constructors* (a real case).

How to add position without recursion schemes - simple solution

For one, we can just go ahead and annotate each constructor

```haskell
data Position = ...

data HuttonExpr =
    Lit Position Double
  | Add Position HuttonExpr HuttonExpr
```

Pro: very simple
Cons:

- Not really feasible if expression type has nearly *100 constructors* (a real case).
- When constructing expressions we must *always* some position. This leads to lots of meaningless positions

How to add position without recursion schemes - smarter solution

Okay, we can factor out common field

```
data Position = ...

data HuttonExpr = HuttonExpr Position HuttonExprBase

data HuttonExprBase =
    Lit Double
  | Add HuttonExpr HuttonExpr
```

How to add position without recursion schemes - smarter solution

Okay, we can factor out common field

```
data Position = ...

data HuttonExpr = HuttonExpr Position HuttonExprBase

data HuttonExprBase =
    Lit Double
  | Add HuttonExpr HuttonExpr
```

Pro: still pretty simple

How to add position without recursion schemes - smarter solution

Okay, we can factor out common field

```
data Position = ...

data HuttonExpr = HuttonExpr Position HuttonExprBase

data HuttonExprBase =
    Lit Double
  | Add HuttonExpr HuttonExpr
```

Pro: still pretty simple
Cons:

- Now data type is less convenient to work with - try matching 2 or 3 levels deep in a function

How to add position without recursion schemes - smarter solution

Okay, we can factor out common field

```haskell
data Position = ...

data HuttonExpr = HuttonExpr Position HuttonExprBase

data HuttonExprBase =
    Lit Double
  | Add HuttonExpr HuttonExpr
```

Pro: still pretty simple
Cons:

- Now data type is less convenient to work with - try matching 2 or 3 levels deep in a function
- Still lots of dummy positions when constructing expressions

A solution through factored recursion

This is where *Cofree* comes in. Compare against *Fix*.

That's the cofree comonad you might've heard of. It's also helpful without *Comonad* instance.

```
newtype Fix f = Fix (f Fix)

data Cofree f a = a :< Cofree f a
  deriving (Show, Functor, Foldable)

data HuttonExprF r =
    LitF Double
  | AddF r r
  deriving (Show, Functor, Foldable)

type AnnotatedHuttonExpr = Cofree HuttonExprF Position
```

A solution through factored recursion

This is where *Cofree* comes in. Compare against *Fix*.

That's the cofree comonad you might've heard of. It's also helpful without *Comonad* instance.

```
newtype Fix f = Fix (f Fix)

data Cofree f a = a :< Cofree f a
  deriving (Show, Functor, Foldable)

data HuttonExprF r =
    LitF Double
  | AddF r r
  deriving (Show, Functor, Foldable)

type AnnotatedHuttonExpr = Cofree HuttonExprF Position
```

Pro: can ignore all positions and get *Fix*'ed *HuttonExprF*

A solution through factored recursion

This is where *Cofree* comes in. Compare against *Fix*.

That's the cofree comonad you might've heard of. It's also helpful without *Comonad* instance.

```
newtype Fix f = Fix (f Fix)

data Cofree f a = a :< Cofree f a
  deriving (Show, Functor, Foldable)

data HuttonExprF r =
    LitF Double
  | AddF r r
  deriving (Show, Functor, Foldable)

type AnnotatedHuttonExpr = Cofree HuttonExprF Position
```

Pro: can ignore all positions and get *Fix*'ed *HuttonExprF*
Cons: requires unfixed type

On to functions

We want to model functions with our *HuttonExpr*.

Functions have arguments. Functions of 1 argument with currying will be enough

Quiz: where to put variables if we had old *HuttonExpr*?

```haskell
-- If you thought annotations were easy, try adding variables here!!
data HuttonExpr =
     Lit Double
   | Add HuttonExpr HuttonExpr
```

On to functions

We want to model functions with our *HuttonExpr*.

Functions have arguments. Functions of 1 argument with currying will be enough

Quiz: where to put variables if we had old *HuttonExpr*?

```
-- If you thought annotations were easy, try adding variables here!!
data HuttonExpr =
    Lit Double
  | Add HuttonExpr HuttonExpr
```

There's simply no place to do this! We *have* to add new costructor and rewrite all our functions to handle it.

Variables - add new costructor

```
data HuttonExprVars a =
    Lit Double
  | Add HuttonExpr HuttonExpr
  | Var a
```

This is actually pretty good.

Via clever introduction of the new parameter, we can recover old expressions without variables by using *Void*, a type without costructors.

```
-- No constructors, ergo no values
-- (except undefined, but it's morally correct to forget it)
data Void

type HuttonExpr = HuttonExprVars Void
```

Variables - add new costructor 2

```haskell
data HuttonExprVars a =
    Lit Double
  | Add HuttonExpr HuttonExpr
  | Var a

type HuttonExpr = HuttonExprVars Void
```

Pro: may be good enough to get the job done

Variables - add new costructor 2

```haskell
data HuttonExprVars a =
    Lit Double
  | Add HuttonExpr HuttonExpr
  | Var a

type HuttonExpr = HuttonExprVars Void
```

Pro: may be good enough to get the job done

Cons:

- Used a parameter - our type may already have a few of those (5 is not a limit in the RealWorld™)

Variables - add new costructor 2

```haskell
data HuttonExprVars a =
    Lit Double
  | Add HuttonExpr HuttonExpr
  | Var a

type HuttonExpr = HuttonExprVars Void
```

Pro: may be good enough to get the job done

Cons:

- Used a parameter - our type may already have a few of those (5 is not a limit in the RealWorld™)
- Pattern match completeness checker will still expect us to handle new *Var* case even when working with *HuttonExprVars Void*

Can modularized recursion help us here?

Sure it can! Meet *Free* - a categorical™dual of *Cofree*.

That's the free monad you might've heard of. It's also helpful without *Monad* instance.

```
data Free f a =
    Pure a
  | Free (f (Free f a))
  deriving (Functor, Foldable)

newtype DeBruijn = DeBruijn Int

type HuttonExprVars = Free HuttonExprF DeBruijn
```

More complicated ASTs may have invariants

That was fun! However, what if our expr can produce either an integer or a bool when evaluated?

```
data IntBoolExpr =
    LitInt Int
  | LitBool Bool
  | IBAdd IntBoolExpr IntBoolExpr
  | IBLessThan IntBoolExpr IntBoolExpr
  | IBIF
      IntBoolExpr -- must be a bool expr
      IntBoolExpr -- can be any expr
      IntBoolExpr -- can be any expr
```

Can represent invalid expressions...

```
data IntBoolExpr =
    LitInt Int
  | LitBool Bool
  | IBAdd IntBoolExpr IntBoolExpr
  | IBLessThan IntBoolExpr IntBoolExpr
  | IBIF
      IntBoolExpr -- must be a bool expr
      IntBoolExpr -- can be any expr
      IntBoolExpr -- can be any expr

-- Breaches every invariant we have...
wat :: IntBoolExpr
wat =
  IBIF
    (LitInt 10)
    (LitBool True)
    (IBAdd (LitBool False) (LitInt 1))
```

Let's add types so that ill-typed terms will be unrepresentable

GADTs to the rescue

```haskell
data Expr t where
  ELitInt   :: Int  → Expr Int
  ELitBool  :: Bool → Expr Bool
  EAdd      :: Expr Int → Expr Int → Expr Int
  ELessThan :: Expr Int → Expr Int → Expr Bool
  EIf       :: Expr Bool → Expr a → Expr a → Expr a
```

It works

Does not compile

EIf (*ELitInt* 10) (*ELitBool True*) (*EAdd* (*ELitBool False*) (*ELit*

```
<interactive>:1:6: error:
    * Couldn't match type 'Int' with 'Bool'
      Expected type: Expr Bool
        Actual type: Expr Int
    * In the first argument of 'EIf', namely '(ELitInt 10)
      In the expression:
        EIf
          (ELitInt 10) (ELitBool True) (EAdd (ELitBool Fal

<interactive>:1:35: error:
    * Couldn't match type 'Int' with 'Bool'
      Expected type: Expr Bool
        Actual type: Expr Int
    * In the third argument of 'EIf', namely
        '(EAdd (ELitBool False) (ELitInt 1))'
      In the expression:
```

But I want my recursion schemes goodness back!

We just need higher-order recursion schemes.

Observe that $Expr$ has a type parameter but it is not a $Functor$. It's something different.

Let's start with a base functor, as before.

Indexed base functor

Replace all the recursive positions, add new parameter. The only difference: parameter has to be indexed.

```
data ExprF h ix where
  ELitIntF    :: Int  → ExprF h Int
  ELitBoolF   :: Bool → ExprF h Bool
  EAddF       :: h Int → h Int → ExprF h Int
  ELessThanF  :: h Int → h Int → ExprF h Bool
  EIfF        :: h Bool → h a → h a → ExprF h a
```

Indexed base functor with kind signatures

I have just added kind signatures to the parameters

```
data ExprF (h :: k → *) (ix :: k) where
  ELitIntF    :: Int  → ExprF h Int
  ELitBoolF   :: Bool → ExprF h Bool
  EAddF       :: h Int → h Int → ExprF h Int
  ELessThanF  :: h Int → h Int → ExprF h Bool
  EIfF        :: h Bool → h a → h a → ExprF h a
```

Indexed recursion combinator

Without further ado, this is how *Fix* should look like for GADTs with a parameter (with polykinds):

```
-- Just ignore the kinds if they upset your senses...
newtype HFix (f :: (k → *) → k → *) (ix :: k) =
  HFix (f (HFix f) ix)

unHFix :: HFix f ix → f (HFix f) ix
unHFix (HFix x) = x
```

We still need functors

To define `cata` we needed a *Functor*. Here it's similar enough - our functor should preserve indices:

```
class HFunctor (f :: (k → *) → k → *) where
  hfmap :: (∀ ix'· g ix' → h ix') → f g ix → f h ix

-- Can derive with TH, if you like
instance HFunctor ExprF where
  hfmap h = \case
    ELitIntF n       → ELitIntF n
    ELitBoolF b      → ELitBoolF b
    EAddF x y        → EAddF (h x) (h y)
    ELessThanF x y → ELessThanF (h x) (h y)
    EIfF c t f       → EIfF (h c) (h t) (h f)
```

Find 10 differences with cata...

```
cata :: Functor f => (f a → a) → Fix f → a
cata alg = go
  where
    go = alg · fmap go · unFix

hcata :: ∀ f g ix· HFunctor f => (∀ ix· f g ix → g ix) → HF
hcata alg = go
  where
    go :: ∀ ix· HFix f ix → g ix
    go = alg · hfmap go · unHFix
```

Now we can evaluate safely

```
data Value ix where
  VInt  :: Int  → Value Int
  VBool :: Bool → Value Bool
deriving instance Show (Value ix)

hevalAlg :: ExprF Value ix → Value ix
hevalAlg = \case
  ELitIntF   n                   → VInt n
  ELitBoolF  b                   → VBool b
  EAddF      (VInt x) (VInt y)   → VInt (x + y)
  ELessThanF (VInt x) (VInt y)   → VBool (x < y)
  EIfF       (VBool c) t f       → if c then t else f

type Expr' = HFix ExprF

heval :: Expr' ix → Value ix
heval = hcata hevalAlg
```

Does it actually run??

Yes it does!!

```
data Value ix where
  VInt  :: Int  → Value Int
  VBool :: Bool → Value Bool

heval :: Expr' ix → Value ix
```

```
heval
  (HFix (EIfF
    (HFix (ELitBoolF False))
    (HFix (ELitIntF 1))
    (HFix (ELitIntF 42))))
```

$\Rightarrow$ *VInt* 42

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time
- Fused

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time
- Fused
  - Metamorphism = `ana . cata`

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time
- Fused
  - Metamorphism = `ana · cata`
  - Hylomorphism = `cata · ana`

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time
- Fused
  - Metamorphism = `ana · cata`
  - Hylomorphism = `cata · ana`
  - Dynamorphism = `histo · ana`

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time
- Fused
  - Metamorphism = `ana · cata`
  - Hylomorphism = `cata · ana`
  - Dynamorphism = `histo · ana`
  - Chronomorphism = `futu · histo`

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time
- Fused
  - Metamorphism = `ana · cata`
  - Hylomorphism = `cata · ana`
  - Dynamorphism = `histo · ana`
  - Chronomorphism = `futu · histo`
- Just a joke

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time
- Fused
  - Metamorphism = `ana . cata`
  - Hylomorphism = `cata . ana`
  - Dynamorphism = `histo . ana`
  - Chronomorphism = `futu . histo`
- Just a joke
  - Zygohistomorphic prepromorphism

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time
- Fused
  - Metamorphism = `ana . cata`
  - Hylomorphism = `cata . ana`
  - Dynamorphism = `histo . ana`
  - Chronomorphism = `futu . histo`
- Just a joke
  - Zygohistomorphic prepromorphism

More recursion schemes

I have only scratched the surface. Look:

- Folding to a value
  - Catamorphism - you have learned it in this talk. Well done!
  - Paramorphism - folding with access to the original expression
  - Zygomorphism - neat combination of several algebras so that they can use results of each other
  - Histomorphism - dynamic programming, algebra has access to all previously computed results for a tree
- Unfolding from a value
  - Anamorphism - one level at a time
  - Futumorphism - multiple levels at a time
- Fused
  - Metamorphism = `ana · cata`
  - Hylomorphism = `cata · ana`
  - Dynamorphism = `histo · ana`
  - Chronomorphism = `futu · histo`
- Just a joke
  - Zygohistomorphic prepromorphism

Comonads can also help to unify some of those.

References

Lots of resources. In order of increasing difficulty:

- Bartosz Milewski blog post "Understanding F-Algebras"
- Tim Williams slides "Recursion Schemes by Example"
- Tim Williams blog post "Fixing GADTs"
- Wouter Swierstra Data Types à la Carte

Thank you!

Thank you!

Questions?