

# Mechanising Recursion Schemes with Magic-Free Coq Extraction

David Castro-Perez, Marco Paviotti, and Michael Vollmer

d.castro-perez@kent.ac.uk

02-05-2024



# Background

Hylomorphisms

# Fold over Lists

One way to guarantee **recursive functions** are **well-defined** is via **Recursion Schemes**.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr g b [] = b
foldr g b (x : xs) = g x (foldr g b xs)
```

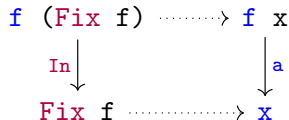
There are many different kinds of Recursion Schemes (e.g. Folds, Paramorphisms, Unfolds, Apomorphisms, ...)

# Folds as Initial Algebras

```
data Fix f = In { in0p :: f (Fix f) }
```

```
fold :: Functor f =>  
      (f x -> x) ->  
      Fix f ->  
      x
```

```
fold a = f  
  where f (In x) = (a . fmap f) x
```



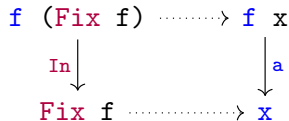
# Folds as Initial Algebras

Least Fixed-Point  
 $\text{Fix } f \cong f (\text{Fix } f)$

```
data Fix f = In { in0p :: f (Fix f) }
```

```
fold :: Functor f =>  
      (f x -> x) ->  
      Fix f ->  
      x
```

```
fold a = f  
  where f (In x) = (a . fmap f) x
```

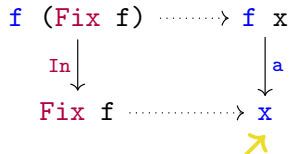


# Folds as Initial Algebras

```
data Fix f = In { in0p :: f (Fix f) }
```

```
fold :: Functor f =>  
      (f x -> x) ->  
      Fix f ->  
      x
```

```
fold a = f  
  where f (In x) = (a . fmap f) x
```



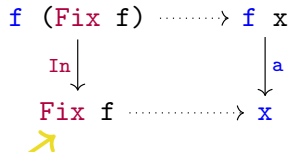
f-algebra

# Folds as Initial Algebras

```
data Fix f = In { in0p :: f (Fix f) }
```

```
fold :: Functor f =>  
    (f x -> x) ->  
    Fix f ->  
    x
```

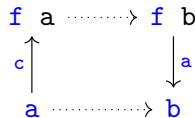
```
fold a = f  
  where f (In x) = (a . fmap f) x
```



initial `f`-algebra

# Hylomorphisms: Divide-and-conquer Computations

```
hylo :: Functor f =>  
    (f b -> b) ->  
    (a -> f a) ->  
    a -> b  
hylo a c = a . fmap (hylo a c) . c
```

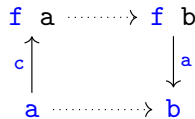




# Hylomorphisms: Divide-and-conquer Computations

```
hylo :: Functor f =>  
    (f b -> b) ->  
    (a -> f a) ->  
    a -> b
```

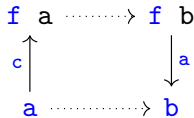
```
hylo a c = a . fmap (hylo a c) . c
```



$f$ -coalgebra  
"divide"

# Hylomorphisms: Divide-and-conquer Computations

```
hylo :: Functor f =>  
    (f b -> b) ->  
    (a -> f a) ->  
    a -> b  
hylo a c = a <-> fmap (hylo a c) . c
```



f-algebra  
"conquer"

# Folds as Hylomorphisms

```
data Fix f = In { inOp :: f (Fix f) }
```

```
fold :: Functor f =>  
      (f x -> x) ->  
      Fix f ->  
      x
```

```
fold a = a ← fmap (fold a) . inOp
```

f-coalgebra

$f \text{ (Fix } f) \cdots \rightarrow f \ x$   
 $\text{inOp} \uparrow$   
 $\text{Fix } f \cdots \rightarrow x$   
 $\downarrow a$

f-algebra

# Adjoint Folds

Given an adjunction:

$$\begin{array}{ccc} & L & \\ \mathcal{D} & \xleftarrow{\quad} & \mathcal{C} \\ & \perp & \\ & \xrightarrow{\quad} & \\ & R & \end{array}$$

- There is a correspondence of arrows  $[\cdot] : \text{Hom}_{\mathcal{D}}(L A, B) \cong \text{Hom}_{\mathcal{C}}(A, R B) : [\cdot]$ .
- An initial algebra on the right corresponds to an universal property on the left:

$$\text{Hom}_{\mathcal{D}}(L \mu F, B) \cong \text{Hom}_{\mathcal{C}}(\mu F, R B)$$

---

$\mu$  analogous to Haskell's **Fix**

$F$  is an endofunctor in  $\mathcal{C}$

$L, R$  are functors between  $\mathcal{C}$  &  $\mathcal{D}$ ; the *left* and *right* adjoints.

# Conjugate Hylomorphisms

*Every recursion scheme is a conjugate hylomorphism*

<i>recursion scheme</i>	<i>adjunction</i>	<i>conjugates</i>	<i>para-hylo equation</i>	<i>algebra</i>
(hylo-shift law)	$\text{Id} \dashv \text{Id}$	$\alpha \dashv \alpha$	$x = a \cdot (\text{id} \triangle D x \cdot \alpha C \cdot c) : A \leftarrow C$	$a : C \times D A \rightarrow A$
mutual recursion	$\Delta \dashv (\times)$	ccf	$x_1 = a_1 \cdot (\text{id} \triangle D (x_1 \triangle x_2) \cdot c) : A_1 \leftarrow C$ $x_2 = a_2 \cdot (\text{id} \triangle D (x_1 \triangle x_2) \cdot c) : A_2 \leftarrow C$	$a_1 : C \times D (A_1 \times A_2) \rightarrow A_1$ $a_2 : C \times D (A_1 \times A_2) \rightarrow A_2$
accumulator	$- \times P \dashv (-)^P$	ccf	$x = a \cdot (\text{outl} \triangle ((D (\wedge x) \cdot c) \times P)) : A \leftarrow C \times P$	$a : C \times D (A^P) \times P \rightarrow A$
course-of-values (§5.6)	$U_D \dashv \text{Cofree}_D$	ccf	$x = a \cdot (\text{id} \triangle D (D_\infty x \cdot [c]) \cdot c) : A \leftarrow C$	$a : C \times D (D_\infty A) \rightarrow A$
finite memo-table (§5.6)	$U_* \dashv \text{Cofree}_*$	ccf	$x = a \cdot (\text{id} \triangle D (D_* x \cdot [c]_*) \cdot c) : A \leftarrow C$	$a : C \times D (D_* A) \rightarrow A$

**Table 1.** Different types of para-hylos building on the canonical control functor (ccf); the coalgebra is  $c : C \rightarrow D C$  in each case.

# Why Mechanising Hylomorphisms in Coq?

- Structured Recursion Schemes have been used in Haskell to structure functional programs, but they do not ensure termination/productivity
- On the other hand, Coq does not capture all recursive definitions
- The benefits of formalising hylos in Coq is three fold:
  - Giving the Coq programmer a **library** where for most recursion schemes they do not have to prove termination properties
  - **Extracting code** into ML/Haskell to provide termination guarantees even in languages with non-termination
  - Using the laws of hylomorphisms as tactics for **program calculation** and **optimisation**

# Challenges

1. Avoiding axioms: functional extensionality, heterogeneous equality, . . . .
2. Extracting “clean” code: close to what a programmer would have written directly in OCaml.
3. Fixed-points of functors, non-termination, etc.

# Challenges

1. Avoiding axioms: functional extensionality, heterogeneous equality, . . . .
2. Extracting “clean” code: close to what a programmer would have written directly in OCaml.
3. Fixed-points of functors, non-termination, etc.

Our solutions (the remainder of this talk):

1. Machinery for building setoids, use of decidable predicates, . . .



# Challenges

1. Avoiding axioms: functional extensionality, heterogeneous equality, . . . .
2. Extracting “clean” code: close to what a programmer would have written directly in OCaml.
3. Fixed-points of functors, non-termination, etc.

Our solutions (the remainder of this talk):

1. Machinery for building setoids, use of decidable predicates, . . .
2. Avoiding type families and indexed types.

# Challenges

1. Avoiding axioms: functional extensionality, heterogeneous equality, . . . .
2. Extracting “clean” code: close to what a programmer would have written directly in OCaml.
3. Fixed-points of functors, non-termination, etc.

Our solutions (the remainder of this talk):

1. Machinery for building setoids, use of decidable predicates, . . .
2. Avoiding type families and indexed types.
3. **Containers & recursive coalgebras**

# Roadmap

**Part I:** Extractable Containers in Coq

**Part II:** Recursive Coalgebras & Coq Hylomorphisms

**Part III:** Code Extraction & Examples

# Part I

## Extractable Containers in Coq

# Part II

## Recursive Coalgebras & Coq Hylomorphisms

# Part III

## Code Extraction & Examples

# Wrap-up

