

Program Optimisations via Hylomorphisms for Extraction of Executable Code

David Castro-Perez, Marco Paviotti, and Michael Vollmer

d.castro-perez@kent.ac.uk

ITP 2025



Why Hylomorphisms in Rocq

1. Recursion schemes offer practical advantages:
 - Abstracting common patterns of recursion.
 - Reasoning about program transformations and optimisations.
2. *Every recursion scheme is a (conjugate) hylomorphism.*
3. Encoding hylomorphisms in Rocq offers three main benefits:
 - Reduce the burden of termination/productivity proofs by structuring recursion modularly so proofs can be reused.
 - Use hylomorphism laws so program calculation and optimisation reduce to plain **rewrite**.
 - Code extraction.

Hylomorphisms

Folds as Initial Algebras

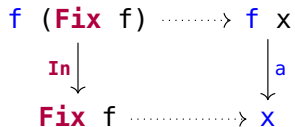
```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr g b [] = b  
foldr g b (x : xs) = g x (foldr g b xs)
```

Folds as Initial Algebras

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

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

```
fold a = c  
  where c (In e) = (a . fmap c) e
```



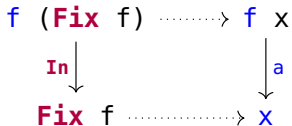
Folds as Initial Algebras

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 = c  
  where c (In e) = (a . fmap c) e
```

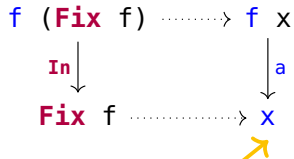


Folds as Initial Algebras

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

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

```
fold a = c  
  where c (In e) = (a ← fmap c) e
```



f-algebra

Folds as Initial Algebras

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

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

```
fold a = c  
  where c (In e) = (a . fmap c) e
```

```
graph TD
    A["f (Fix f)"] -.-> B["f x"]
    A -- In --> C["Fix f"]
    B -- a --> D["x"]
    C -.-> D
```

initial f-algebra

Folds as Initial Algebras

```
data LF a b = NilF | ConsF a b
```

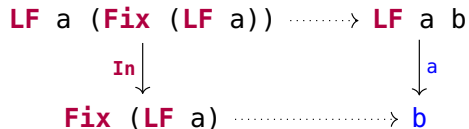
```
type Ls a = Fix (LF a)
```

```
foldr :: (a -> b -> b) -> b -> Ls a -> b
```

```
foldr f z = fold a
```

```
  where a NilF = z
```

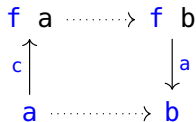
```
        a (ConsF a b) = f a b
```



Hylomorphisms: Divide-and-conquer Recursion

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

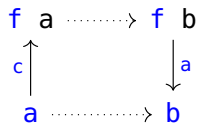
```
hylo a c = a . fmap (hylo a c) . c
```



Hylomorphisms: Divide-and-conquer Recursion

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

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

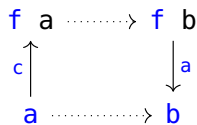


f-coalgebra
"divide"

Hylomorphisms: Divide-and-conquer Recursion

```
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

$f \text{ (Fix } f) \xrightarrow{\dots} f \ x$
 $\text{inOp} \uparrow$
 $\text{Fix } f \xrightarrow{\dots} x$
 $\downarrow a$

f-algebra

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.

Challenges for Encoding Hylos in Rocq

1. Avoiding axioms and accepting program calculation closely resembling pen-and-paper proofs.
2. Extracting idiomatic code.
3. Termination and (co)fixed-points of functors.

Challenges for Encoding Hylos in Rocq

1. **Avoiding axioms** and accepting program calculation closely resembling pen-and-paper proofs.
2. **Extracting idiomatic code.**
3. **Termination** and (co)fixed-points of functors.

Our solutions (the remainder of this talk):

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

“Extractable” Containers in Rocq

Containers

Containers are defined by a pair $S \triangleleft P$:

- a type of **shapes** $S : \text{Type}$
- a **family** of positions, indexed by shape $P : S \rightarrow \text{Type}$

Containers

Containers are defined by a pair $S \triangleleft P$:

- a type of **shapes** $S : \text{Type}$
- a **family** of positions, indexed by shape $P : S \rightarrow \text{Type}$

A **container extension** is a functor defined as follows

$$\llbracket S \triangleleft P \rrbracket X = \Sigma_{s:S} P\ s \rightarrow X$$

$$\llbracket S \triangleleft P \rrbracket f = \lambda(s, p). (s, f \circ p)$$

Containers: Example

Consider the functor $F X = 1 + X \times X$

S_F and P_F define a container that is isomorphic to F

$$\begin{array}{ll} S_F = 1 + 1 & P_F (\text{inl } \bullet) = 0 \\ & P_F (\text{inr } \bullet) = 1 + 1 \end{array}$$

Examples of objects of types $F \mathbb{N}$ (left) and $\llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$ (right):

$$\begin{array}{ll} \text{inl } \bullet & \cong (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) & \cong (\text{inr } \bullet, \lambda x, \text{case } x \{ \text{inl } \bullet \Rightarrow 7; \text{inr } \bullet \Rightarrow 9 \}) \end{array}$$

Containers: Example

Consider the functor $F X = 1 + X \times X$

S_F and P_F define a container that is isomorphic to F

Two cases (“shapes”)

$$S_F = 1 + 1 \qquad \begin{aligned} P_F (\text{inl } \bullet) &= 0 \\ P_F (\text{inr } \bullet) &= 1 + 1 \end{aligned}$$

Examples of objects of types $F \mathbb{N}$ (left) and $\llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$ (right):

$$\begin{aligned} \text{inl } \bullet &\cong (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) &\cong (\text{inr } \bullet, \lambda x, \text{case } x \{ \text{inl } \bullet \Rightarrow 7; \text{inr } \bullet \Rightarrow 9 \}) \end{aligned}$$

Containers: Example

No positions on the left shape

Consider the functor $F X = 1 + X \times X$

S_F and P_F define a container that is isomorphic to F

$$S_F = 1 + 1$$

$$P_F (\text{inl } \bullet) = 0$$

$$P_F (\text{inr } \bullet) = 1 + 1$$

Examples of objects of types $F \mathbb{N}$ (left) and $\llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$ (right):

$$\begin{aligned} \text{inl } \bullet &\cong (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) &\cong (\text{inr } \bullet, \lambda x, \text{case } x \{ \text{inl } \bullet \Rightarrow 7; \text{inr } \bullet \Rightarrow 9 \}) \end{aligned}$$

Containers: Example

Two positions on the right shape

Consider the functor $F X = 1 + X \times X$

S_F and P_F define a container that is isomorphic to F

$$S_F = 1 + 1$$
$$P_F (\text{inl } \bullet) = 0$$
$$P_F (\text{inr } \bullet) = 1 + 1$$

Examples of objects of types $F \mathbb{N}$ (left) and $\llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$ (right):

$$\begin{aligned} \text{inl } \bullet &\cong (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) &\cong (\text{inr } \bullet, \lambda x, \text{case } x \{ \text{inl } \bullet \Rightarrow 7; \text{inr } \bullet \Rightarrow 9 \}) \end{aligned}$$

Container Mechanisation for Clean Extraction

- We use $A \leadsto B$ to denote proper morphisms, where A and B are setoids.
Avoids assuming functional extensionality
- A container C has three components, Shape , Pos , and valid .
- $\text{Shape } C : \text{Type}$ and $\text{Pos } C : \text{Type}$ represent shapes and *all* possible positions.
- $\text{valid } C : \text{Shape } C * \text{Pos } C \leadsto \text{bool}$ is a **decidable** predicate stating when a pair shape/position is valid.
Avoids UIP/Axiom K.

Container Mechanisation for Clean Extraction

- We use $A \sim> B$ to denote proper morphisms, where A and B are setoids.
Avoids assuming functional extensionality
- A container C has three components, `Shape`, `Pos`, and `valid`.
- `Shape C : Type` and `Pos C : Type` represent shapes and *all* possible positions.
- `valid C : Shape C * Pos C $\sim>$ bool` is a **decidable** predicate stating when a pair shape/position is valid.
Avoids UIP/Axiom K.

Container extensions:

```
Record App C (X : Type)
:= MkCont { shape : Shape C;
            contents : {p | valid C (shape, p)} -> X
          }.
```

Extraction will treat contents equivalently to `Pos C -> X`: no unsafe coercions.

Recursive Coalgebras & Hylomorphisms

(Co)algebras & (co)fixpoints

The least/greatest fixed-points of a container extension `App C` are:

Inductive `LFix C := Lin { lin_op : App C (LFix C) }.`

CoInductive `GFix C := Gin { gin_op : App C (GFix C) }.`

Cata/anamorphisms

`cata : (App C X ~> X) ~> LFix C ~> X`

`cata_univ : forall (a : App C X ~> X) (f : LFix C ~> X),
f \o Lin =e a \o fmap f <-> f =e cata a`

`ana : (X ~> App C X) ~> X ~> GFix C`

`ana_univ : forall (c : X ~> App C X) (f : X ~> GFix C),
gin_op \o f =e fmap f \o c <-> f =e ana c`

Recursive Coalgebras

We cannot use arbitrary coalgebras, because their hylomorphisms may not exist.

Recursive Coalgebras

We cannot use arbitrary coalgebras, because their hylomorphisms may not exist.

Recursive coalgebras: coalgebras $(c : X \multimap \text{App } C \ X)$ that terminate in all inputs.

Recursive Coalgebras

We cannot use arbitrary coalgebras, because their hylomorphisms may not exist.

Recursive coalgebras: coalgebras $(c : X \multimap \text{App } C \ X)$ that terminate in all inputs.

We define $\text{RecF } c \ x$ to represent that recursively applying $c : X \multimap \text{App } C \ X$ terminates on input $x : X$.

1. Recursive coalgebras:

$$\text{RCoAlg } C \ X = \{c \mid \text{forall } x, \text{RecF } c \ x\}$$

2. Well-founded coalgebras, given a well-founded relation R ,

$$\text{WfCoAlg } C \ X = \{c \mid \text{forall } x \ p, R \ (\text{contents } (c \ x) \ p) \ x\}$$

Recursive Coalgebras

We cannot use arbitrary coalgebras, because their hylomorphisms may not exist.

Recursive coalgebras: coalgebras $(c : X \leadsto \text{App } C \text{ } X)$ that terminate in all inputs.

We define $\text{RecF } c \text{ } x$ to represent that recursively applying $c : X \leadsto \text{App } C \text{ } X$ terminates on input $x : X$.

1. Recursive coalgebras:

$$\text{RCoAlg } C \text{ } X = \{c \mid \text{forall } x, \text{RecF } c \text{ } x\}$$

2. Well-founded coalgebras, given a well-founded relation R ,

$$\text{WfCoAlg } C \text{ } X = \{c \mid \text{forall } x \text{ } p, R \text{ (contents (c } x) \text{ } p) \text{ } x\}$$

- Definitions (1) and (2) are equivalent
- Termination proofs may be easier using (1) or (2), depending on the use case (e.g. structural recursion is trivial using (1)).

Recursive Hyломorphisms

The definition of recursive hylomorphisms is structural on $\text{RecF } c \ x$:

```
Definition hylo_def (a : App F B ~> B) (c : A ~> App F A)
  : forall (x : A), RecF c x -> B :=
fix f x H :=
  match c x as cx
    return (forall e : Pos (shape cx), RecF c (contents cx e)) -> B
with
  | MkCont sx cx => fun H => a (MkCont sx (fun e => f (cx e) (H e)))
end (RecF_inv H).
```

Recursive hylomorphisms are the unique solution to the hylomorphism equation:

$\text{hylo} : (\text{App } C \ B \ \sim\!> \ B) \ \sim\!> \ \{c : A \ \sim\!> \ \text{App } C \ A \mid \text{forall } x, \text{RecF } c \ x\} \ \sim\!> \ A \ \sim\!> \ B$

$\text{hylo_unique} : \text{forall } (f : A \ \sim\!> \ B) \ (a : \text{App } C \ B \ \sim\!> \ B) \ (c : A \ \sim\!> \ \text{App } C \ A),$
 $f = e \ a \ \backslash o \ \text{fmap } f \ \backslash o \ c \ \text{<-> } f = \text{hylo } a \ c$

Hylomorphism Fusion

The following laws are straightforward consequences of `hylo_unique`.

```
Lemma hylo_fusion_l
      : h \o a =e b \o fmap h -> h \o hylo a c =e hylo b c.

Lemma hylo_fusion_r
      : c \o h =e fmap h \o d -> hylo a c \o h =e hylo a d.

Lemma deforest
      : f \o g =e id -> hylo a f \o hylo g c =e hylo a c.
```

Hylomorphism Fusion

The following laws are straightforward consequences of `hylo_unique`.

Lemma `hylo_fusion_l`

`: h \o a =e b \o fmap h -> h \o hylo a c =e hylo b c.`

The Rocq proofs are exactly as the pen-and-paper proofs: By `hylo_unique`, `hylo b c` is the only arrow making the outer square commute.

$$\begin{array}{ccccc} tb & \xleftarrow{h} & ta & \xleftarrow{\text{hylo } a \ c} & tc \\ \uparrow b & & \uparrow a & & \downarrow c \\ f \ tb & \xleftarrow{\text{fmap } h} & f \ ta & \xleftarrow{\text{fmap } (\text{hylo } a \ c)} & f \ tc \end{array}$$

Extraction

Example: Quicksort

Definition mergeF (x : App (TreeC int) (list int)) : list int :=
 match t_out x with
 | inl _ => nil
 | inr (p, l, r) => List.app l (p :: r)
end.

Definition splitF (l : list int) : App (TreeC int) (list int) :=
 match l with
 | nil => a_leaf
 | cons h t => let (l, r) := List.partition (fun x => x <=? h) t in
 a_node h l r
end.

Example: Quicksort

Definition `qsort := hylo merge splitt.`
Extraction `qsort`.

```
let rec qsort = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x0 -> leb x0 h) t in
  let x0 = fun e -> qsort (match e with
                           | Lbranch -> l
                           | Rbranch -> r) in
  app (x0 Lbranch) (h :: (x0 Rbranch))
```

Example: Quicksort

Definition qsort_times_two
: {f | f =e Lmap times_two \o hylo merge splitt}.
eapply exist. (* ... *) **rewrite** (hylo_fusion_l H); **reflexivity**.
Defined.

Extraction qsort_times_two.

```
let rec qsort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x0 -> leb x0 h) t in
  let x0 = fun p -> qsort_times_two (match p with
                                     | Lbranch -> l
                                     | Rbranch -> r) in
  app (x0 Lbranch) ((mul (Uint63.of_int (2)) h) :: (x0 Rbranch))
```

Further details in the paper

Coalgebras & Anamorphisms.

Further examples, e.g. shortcut deforestation & dynamorphisms.

Example correctness proof. Proving properties of algorithms implemented as hylomorphisms is comparable to alternative non-structural recursion encodings.

Correctness proofs of encodings using hylomorphisms can exploit *program calculation*. This can lead to more modular proofs. E.g. if we know that $\forall x, Q(f(x))$, and $\forall x, Q(x) \rightarrow P(g(x))$, then we can conclude $\forall x, P((g \circ f)(x))$, and then fuse $(g \circ f)$ into an optimised, extensionally equal version.

Summary

- Modular specification of functions, without sacrificing performance thanks to program calculation (`hylo_fusion`).
- Modular treatment of divide-and-conquer and termination proofs using recursive coalgebras.
- Idiomatic code extraction.

Summary

- Modular specification of functions, without sacrificing performance thanks to program calculation (`hylo_fusion`).
- Modular treatment of divide-and-conquer and termination proofs using recursive coalgebras.
- Idiomatic code extraction.

Possible extensions:

- Effects.
- Dealing with setoids & equalities.
- **Corecursive algebras.** (use of guarded recursion?)

Summary

- Modular specification of functions, without sacrificing performance thanks to program calculation (`hylo_fusion`).
- Modular treatment of divide-and-conquer and termination proofs using recursive coalgebras.
- Idiomatic code extraction.

Possible extensions:

- Effects.
- Dealing with setoids & equalities.
- **Corecursive algebras.** (use of guarded recursion?)

Thank you!