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

University of Kent

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.

R. Hinze, N. Wu, J. Gibbons: **Conjugate Hylomorphisms - Or: The Mother of All Structured Recursion Schemes**. POPL 2015.

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

```haskell
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
```

$$
\begin{tikzcd}
\texttt{f (Fix f)} \arrow[r, dotted] \arrow[d, "\texttt{In}"'] & \texttt{f x} \arrow[d, "\texttt{a}"] \\
\texttt{Fix f} \arrow[r, dotted] & \texttt{x}
\end{tikzcd}
$$

Folds as Initial Algebras

Least Fixed-Point

`Fix f` $\cong$ `f (Fix f)`

```haskell
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
```

$$
\begin{tikzcd}
\texttt{f (Fix f)} \arrow[r, dotted] \arrow[d, "\texttt{In}"'] & \texttt{f x} \arrow[d, "\texttt{a}"] \\
\texttt{Fix f} \arrow[r, dotted] & \texttt{x}
\end{tikzcd}
$$

Folds as Initial Algebras

```haskell
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
```

$$
\begin{tikzcd}
\texttt{f (Fix f)} \arrow[r, dotted] \arrow[d, "\texttt{In}"'] & \texttt{f x} \arrow[d, "\texttt{a}"] \\
\texttt{Fix f} \arrow[r, dotted] & \texttt{x}
\end{tikzcd}
$$

f-algebra

Folds as Initial Algebras

```haskell
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
```

$$
\begin{tikzcd}
\mathtt{f\ (Fix\ f)} \arrow[r, dotted] \arrow[d, "\mathtt{In}"'] & \mathtt{f\ x} \arrow[d, "\mathtt{a}"] \\
\mathtt{Fix\ f} \arrow[r, dotted] & \mathtt{x}
\end{tikzcd}
$$

initial f-algebra

Folds as Initial Algebras

```haskell
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
```

$$
\begin{tikzcd}
\texttt{LF a (Fix (LF a))} \arrow[r, dotted] \arrow[d, "\texttt{In}"'] & \texttt{LF a b} \arrow[d, "\texttt{a}"] \\
\texttt{Fix (LF a)} \arrow[r, dotted] & \texttt{b}
\end{tikzcd}
$$

Hylomorphisms: Divide-and-conquer Recursion

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

$$\begin{tikzcd} \mathtt{f\ a} \arrow[r, dotted] & \mathtt{f\ b} \arrow[d, "\mathtt{a}"] \\ \mathtt{a} \arrow[u, "\mathtt{c}"] \arrow[r, dotted] & \mathtt{b} \end{tikzcd}$$

Hylomorphisms: Divide-and-conquer Recursion

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

$$\begin{tikzcd} f\ a \arrow[r, dotted] & f\ b \arrow[d, "a"] \\ a \arrow[u, "c"] \arrow[r, dotted] & b \end{tikzcd}$$

f-coalgebra
"divide"

Hylomorphisms: Divide-and-conquer Recursion

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

$$\begin{tikzcd} \mathtt{f\ a} \arrow[r, dotted] & \mathtt{f\ b} \arrow[d, "\mathtt{a}"] \\ \mathtt{a} \arrow[u, "\mathtt{c}"] \arrow[r, dotted] & \mathtt{b} \end{tikzcd}$$

f-algebra
“conquer”

Folds as Hylomorphisms

f-coalgebra

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

fold :: Functor f =>
          (f x -> x) ->
          Fix f ->
          x
fold a = a . fmap (fold a) . inOp
```

$$
\begin{tikzcd}
\texttt{f (Fix f)} \arrow[r, dotted] & \texttt{f x} \arrow[d, "\texttt{a}"] \\
\texttt{Fix f} \arrow[u, "\texttt{inOp}"] \arrow[r, dotted] & \texttt{x}
\end{tikzcd}
$$

f-algebra

Conjugate Hylomorphisms

Every recursion scheme is a conjugate hylomorphism

recursion scheme	*adjunction*	*conjugates*	*para-hylo equation*	*algebra*
(hylo-shift law)	$\mathsf{Id} \dashv \mathsf{Id}$	$\alpha \dashv \alpha$	$x = a \cdot (id \vartriangle \mathsf{D}\, x \cdot \alpha\, C \cdot c) \ : \ A \leftarrow C$	$a : C \times \mathsf{D}\, A \to A$
mutual recursion	$\Delta \dashv (\times)$	ccf	$x_1 = a_1 \cdot (id \vartriangle \mathsf{D}\,(x_1 \vartriangle x_2) \cdot c) \ : \ A_1 \leftarrow C$ $x_2 = a_2 \cdot (id \vartriangle \mathsf{D}\,(x_1 \vartriangle x_2) \cdot c) \ : \ A_2 \leftarrow C$	$a_1 : C \times \mathsf{D}\,(A_1 \times A_2) \to A_1$ $a_2 : C \times \mathsf{D}\,(A_1 \times A_2) \to A_2$
accumulator	$- \times P \dashv (-)^P$	ccf	$x = a \cdot (outl \vartriangle ((\mathsf{D}\,(\Lambda\, x) \cdot c) \times P)) \ : \ A \leftarrow C \times P$	$a : C \times \mathsf{D}\,(A^P) \times P \to A$
course-of-values (§5.6)	$\mathsf{U}_\mathsf{D} \dashv \mathsf{Cofree}_\mathsf{D}$	ccf	$x = a \cdot (id \vartriangle \mathsf{D}\,(\mathsf{D}_\infty\, x \cdot [\![c]\!]) \cdot c) \ : \ A \leftarrow C$	$a : C \times \mathsf{D}\,(\mathsf{D}_\infty\, A) \to A$
finite memo-table (§5.6)	$\mathsf{U}_* \dashv \mathsf{Cofree}_*$	ccf	$x = a \cdot (id \vartriangle \mathsf{D}\,(\mathsf{D}_*\, x \cdot [\![c]\!]_*) \cdot c) \ : \ A \leftarrow C$	$a : C \times \mathsf{D}\,(\mathsf{D}_*\, A) \to A$

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

R. Hinze, N. Wu, J. Gibbons: **Conjugate Hylomorphisms - Or: The Mother of All Structured Recursion Schemes**. POPL 2015.

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 $S \lhd P$:

- a type of **shapes** S : Type
- a **family** of positions, indexed by shape $P : S \to$ Type

Abbott, Altenkirch, Ghani: **Categories of Containers**. FoSSaCS 2003.

Containers

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

- a type of **shapes** S : Type
- a **family** of positions, indexed by shape $P : S \to$ Type

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

$$
\begin{aligned}
[\![S \triangleleft P]\!]\; X &= \Sigma_{s:S} P\; s \to X \\
[\![S \triangleleft P]\!]\; f &= \lambda(s, p).\; (s, f \circ p)
\end{aligned}
$$

Abbott, Altenkirch, Ghani: **Categories of Containers**. FoSSaCS 2003.

Containers: Example

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

S_F and P_F define a container that is isomorphic to F

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

Examples of objects of types $F\ \mathbb{N}$ (left) and $[\![S_F \lhd P_F]\!]\ \mathbb{N}$ (right):

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

Containers: Example

Consider the functor $F\ X = \boxed{1} + \boxed{X \times X}$

S_F and P_F define a container that is isomorphic to F

Two cases ("shapes")

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

Examples of objects of types $F\ \mathbb{N}$ (left) and $[\![S_F \lhd P_F]\!]\ \mathbb{N}$ (right):

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

Containers: Example

No positions on the left shape

Consider the functor $F\ X = \boxed{1} + X \times X$

S_F and P_F define a container that is isomorphic to F

$$S_F = \boxed{1} + 1 \qquad \begin{array}{l} \boxed{P_F\ (\mathsf{inl}\ \bullet) = 0} \\ P_F\ (\mathsf{inr}\ \bullet) = 1 + 1 \end{array}$$

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

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

Containers: Example

Two positions on the right shape

Consider the functor $F\ X = 1 + \boxed{X \times X}$

S_F and P_F define a container that is isomorphic to F

$$S_F = 1 + \boxed{1} \qquad \begin{array}{l} P_F\ (\mathsf{inl}\ \bullet) = 0 \\ \boxed{P_F\ (\mathsf{inr}\ \bullet) = 1 + 1} \end{array}$$

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

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

Container Mechanisation for Clean Extraction

- We use `A ~> 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 ~> 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 ~> 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 ~> 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.

J. Adámek, S. Milius, L.S. Moss: **On Well-Founded and Recursive Coalgebras**. FoSSaCS 2020.

Recursive Coalgebras

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

Recursive coalgebras: coalgebras (`c : X ~> App C X`) that terminate in all inputs.

J. Adámek, S. Milius, L.S. Moss: **On Well-Founded and Recursive Coalgebras**. FoSSaCS 2020.

Recursive Coalgebras

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

Recursive coalgebras: coalgebras (`c : X ~> App C X`) that terminate in all inputs.

We define `RecF c x` to represent that recursively applying `c : X ~> App C X` terminates on input `x : X`.

1. Recursive coalgebras:
   `RCoAlg C X = {c | forall x, RecF c x}`
2. Well-founded coalgebras, given a well-founded relation `R`,
   `WfCoalg C X = {c | forall x p, R (contents (c x) p) x}`

J. Adámek, S. Milius, L.S. Moss: **On Well-Founded and Recursive Coalgebras**. FoSSaCS 2020.

Recursive Coalgebras

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

Recursive coalgebras: coalgebras (`c : X ~> App C X`) that terminate in all inputs.

We define `RecF c x` to represent that recursively applying `c : X ~> App C X` terminates on input `x : X`.

1. Recursive coalgebras:
   `RCoAlg C X = {c | forall x, RecF c x}`
2. Well-founded coalgebras, given a well-founded relation `R`,
   `WfCoalg C X = {c | forall x p, R (contents (c x) p) 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)).

J. Adámek, S. Milius, L.S. Moss: **On Well-Founded and Recursive Coalgebras**. FoSSaCS 2020.

Recursive Hylomorphisms

The definition of recursive hylomorphisms is structural on `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:

```
hylo : (App C B ~> B) ~> {c : A ~> App C A | forall x, RecF c x} ~> A ~> B

hylo_unique : forall (f : A ~> B) (a : App C B ~> B) (c : A ~> App C A),
  f =e a \o fmap f \o c <-> f = hylo a c
```

Hylomorphism Fusion

The following laws <u>are straightforward consequences</u> 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.
```

L

L

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.

$$
\begin{tikzcd}
tb & ta \arrow[l, "h"'] & tc \arrow[l, "\text{hylo}\ a\ c"'] \arrow[d, "c"] \\
f\ tb \arrow[u, "b"] & f\ ta \arrow[l, "\text{fmap}\ h"] \arrow[u, "a"'] & f\ tc \arrow[l, "\text{fmap}\ (\text{hylo}\ a\ c)"]
\end{tikzcd}
$$

R. Hinze, N. Wu, J. Gibbons: **Conjugate Hylomorphisms - Or: The Mother of All Structured Recursion Schemes**. POPL 2015.

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!