

Program Calculation with Hylomorphisms in Rocq

David Castro-Perez

`David.CastroPerez@rhul.ac.uk`



Based on joint work with Marco Paviotti and Michael Vollmer (University of Kent):
Program Optimisations via Hylomorphisms for Extraction of Executable Code, ITP 2025.

Program calculation:

- Simple specification
- Rewrite with laws
- Efficient program

Our work:

- Hylomorphisms in Rocq
- Axiom-free
- Idiomatic OCaml extraction

A Specification and a Program

$\text{map}(\lambda x. 2x) \circ \text{sort}$

```
let rec sort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x -> x < h) t in
  sort_times_two l @ (h * 2) :: sort_times_two r
```

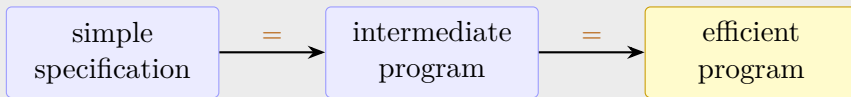
A Specification and a Program

$\text{map}(\lambda x. 2x) \circ \text{sort}$

```
let rec sort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x -> x < h) t in
  sort_times_two l @ (h * 2) :: sort_times_two r
```

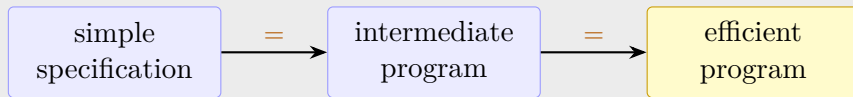
Can we *derive* the program from the specification?

Program Calculation



- Every step is an equation: correct by construction.

Program Calculation



- Every step is an equation: correct by construction.
- The laws come from *recursion schemes*.

Background

Recursion Schemes

Functors and Algebras

$$\text{List } A \cong 1 + A \times \text{List } A = F(\text{List } A)$$

- *Algebra*: $a : F B \rightarrow B$.
- *Initial algebra*: the constructors, $\text{in} = [\text{nil}, \text{cons}]$.

Catamorphisms (Folds)

$$\begin{array}{ccc} F \mu F & \xrightarrow{F \langle a \rangle} & F B \\ \text{in} \downarrow & & \downarrow a \\ \mu F & \xrightarrow[\langle a \rangle]{} & B \end{array} \quad f = \langle a \rangle \iff f \circ \text{in} = a \circ F f$$

A *unique* arrow out of the initial algebra.

Fusion

$$\begin{array}{ccccc} F \mu F & \xrightarrow{F \langle a \rangle} & F B & \xrightarrow{F h} & F C \\ \text{in} \downarrow & & \downarrow a & & \downarrow b \\ \mu F & \xrightarrow{\langle a \rangle} & B & \xrightarrow{h} & C \\ & \searrow \langle a \rangle & & \nearrow h & \\ & & \langle b \rangle & & \end{array}$$

$$h \circ a = b \circ F h \implies h \circ \langle a \rangle = \langle b \rangle$$

Two traversals become one.

Coalgebras and Anamorphisms (Unfolds)

$$\begin{array}{ccc} C & \xrightarrow{[(c)]} & \nu F \\ c \downarrow & & \downarrow \text{out} \\ F C & \xrightarrow{F[(c)]} & F \nu F \end{array}$$

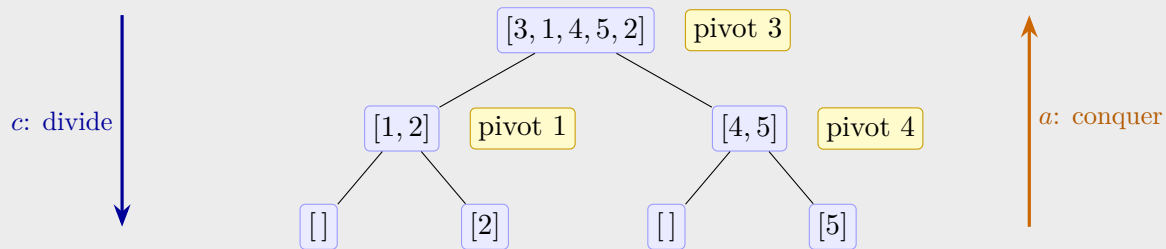
$\text{digits}_2 n = \mathbf{if} \ n = 0 \ \mathbf{then} \ \text{nil} \ \mathbf{else} \ (n \bmod 2, \ n/2)$

Hylomorphisms: Divide and Conquer

$$\begin{array}{ccc} A & \overset{h}{\dashrightarrow} & B \\ c \downarrow & & \uparrow a \\ F A & \xrightarrow{F h} & F B \end{array} \quad h = a \circ F h \circ c$$

Quicksort: c partitions, a concatenates.

The Call Tree of Quicksort



A hylomorphism never builds this tree.

Recursive Coalgebras and their Laws

c recursive: a unique solution $\llbracket a, c \rrbracket$ for every a .

$$\begin{array}{ccc}
 A & \xrightarrow{\llbracket a, c \rrbracket} & B \\
 & \searrow \llbracket a', c \rrbracket & \downarrow f \\
 & & C
 \end{array}
 \qquad
 \begin{array}{ccccc}
 A & \xrightarrow{\llbracket (c) \rrbracket} & \mu F & \xrightarrow{\llbracket (b) \rrbracket} & B \\
 & \searrow \llbracket b, c \rrbracket & & & \nearrow
 \end{array}$$

$$\begin{array}{lll}
 f \circ a = a' \circ F f & \implies & f \circ \llbracket a, c \rrbracket = \llbracket a', c \rrbracket \\
 c \circ f = F f \circ c' & \implies & \llbracket a, c \rrbracket \circ f = \llbracket a, c' \rrbracket \\
 c' \circ a = \text{id} & \implies & \llbracket b, c' \rrbracket \circ \llbracket a, c \rrbracket = \llbracket b, c \rrbracket
 \end{array}$$

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	$x = a \cdot (\text{id} \triangle D (D_\infty x \cdot \llbracket c \rrbracket) \cdot c) : A \leftarrow C$	$a : C \times D (D_\infty A) \rightarrow A$
finite memo-table (§5.6)	$U_* \dashv \text{Cofree}_*$	ccf	$x = a \cdot (\text{id} \triangle D (D_* x \cdot \llbracket c \rrbracket_*) \cdot c) : A \leftarrow C$	$a : C \times D (D_* A) \rightarrow A$

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 Hylomorphisms in Rocq?

- Haskell: the schemes, but no termination.
- Rocq: termination, but these definitions are rejected.
- We want: a *library*, *extracted code*, and the *laws as tactics*.

Challenges

1. No axioms
2. Clean extracted code
3. Non-structural recursion

Challenges

1. No axioms
2. Clean extracted code
3. Non-structural recursion

Solutions:

1. Setoids; decidable validity.

Challenges

1. **No axioms**
2. **Clean extracted code**
3. **Non-structural recursion**

Solutions:

1. Setoids; decidable validity.
2. One type of positions, not a family.

Challenges

1. No axioms
2. Clean extracted code
3. Non-structural recursion

Solutions:

1. Setoids; decidable validity.
2. One type of positions, not a family.
3. Recursion on a termination proof.

Challenges

1. No axioms
2. Clean extracted code
3. Non-structural recursion

Solutions:

1. Setoids; decidable validity.
2. One type of positions, not a family.
3. Recursion on a termination proof.

Since the paper: usability.

Roadmap

Part I: Extractable Containers

Part II: Recursive Coalgebras

Part III: Calculation and Extraction

Part IV: Dynamic Programming

Part V: Monoid Algebras

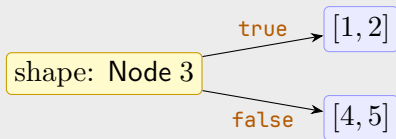
Part I

Extractable Containers in Rocq

Containers

$$\llbracket S \triangleleft P \rrbracket X = \sum_{s:S} (P\ s \rightarrow X)$$

$$\llbracket S \triangleleft P \rrbracket f\ (s, g) = (s, f \circ g)$$



shape: Leaf no positions

Containers in Rocq: A Bad Attempt

```
Record Container := { Sh : Type; Pos : Sh -> Type }.
```

```
Record Ext C X := MkExt { shape : Sh C; cont : Pos C shape -> X }.
```

```
(* quicksort: a leaf, or a pivot with two positions *)
```

```
Definition TreeC : Container :=
```

```
  { | Sh := option nat;
```

```
    Pos := fun s => match s with None => False | Some _ => bool end | }.
```

Containers in Rocq: A Bad Attempt

```
Record Container := { Sh : Type; Pos : Sh -> Type }.
```

```
Record Ext C X := MkExt { shape : Sh C; cont : Pos C shape -> X }.
```

```
(* quicksort: a leaf, or a pivot with two positions *)
```

```
Definition TreeC : Container :=
```

```
  { | Sh := option nat;
```

```
    Pos := fun s => match s with None => False | Some _ => bool end | }.
```

Positions are a type *family*.

A Bad Attempt: Extraction and Equality

```
type sh = __ (* = Obj.t *)
type pos = __
type 'x ext = { shape : sh; cont : (pos -> 'x) }

let merge x =
  let { shape = shape0; cont = k } = x in
  (match Obj.magic shape0 with
   | Some h -> app (Obj.magic k True) (Cons (h, (Obj.magic k False)))
   | None -> Nil)
```

A Bad Attempt: Extraction and Equality

```
Lemma ext_inj s (k1 k2 : Pos C s -> X) : MkExt s k1 = MkExt s k2 -> k1 = k2.
```

```
Proof. intros H; injection H; intros E.
```

```
  (* E : existT _ s k1 = existT _ s k2. Now what? *)
```

```
  apply inj_pair2 in E; exact E. Qed.
```

```
Print Assumptions ext_inj.  Axioms: Eq_rect_eq.eq_rect_eq  (axiom K)
```

```
Print Assumptions fmap_id.  Axioms: functional_extensionality_dep
```

Extractable Containers in Rocq (I)

1. Decidable equality gives UIP: **no axiom**.
2. Proofs in `Prop` are **erased**.

Extractable Containers in Rocq (I)

1. Decidable equality gives UIP: **no axiom**.
2. Proofs in Prop are **erased**.

```
Lemma elem_val_eq (e1 e2 : Pos F s) : val e1 = val e2 -> e1 = e2.  
Proof. ...; apply bool_irrelevance. Qed.
```

Extractable Containers in Rocq (I)

1. Decidable equality gives UIP: **no axiom**.
2. Proofs in Prop are **erased**.

```
Lemma elem_val_eq (e1 e2 : Pos F s) : val e1 = val e2 -> e1 = e2.  
Proof. ...; apply bool_irrelevance. Qed.
```



UIP on bool

Extractable Containers in Rocq (and II)

```
Class Cont `{setoid Sh} (P : Type) :=  
  { valid : Sh * P ~> bool }.
```

```
Record Pos `{Cont Sh P} (s : Sh) :=  
  MkElem { val : P; Valid : valid (s, val) }.
```

```
Record App `{F : Cont Sh P} (X : Type) :=  
  MkCont { shape : Sh; cont : Pos F shape -> X }.
```

Extractable Containers in Rocq (and II)

decidable validity

```
Class Cont `{setoid Sh} (P : Type) :=  
  { valid : Sh * P ~> bool }.
```

```
Record Pos `{Cont Sh P} (s : Sh) :=  
  MkElem { val : P; Valid : valid (s, val) }.
```

```
Record App `{F : Cont Sh P} (X : Type) :=  
  MkCont { shape : Sh; cont : Pos F shape -> X }.
```

Extractable Containers in Rocq (and II)

decidable validity

```
Class Cont `{setoid Sh} (P : Type) :=  
  { valid : Sh * P ~> bool }.
```

```
Record Pos `{Cont Sh P} (s : Sh) :=  
  MkElem { val : P; Valid : valid (s, val) }.
```

unique, erased

```
Record App `{F : Cont Sh P} (X : Type) :=  
  MkCont { shape : Sh; cont : Pos F shape -> X }.
```

Example: the Recursion of Quicksort

```
Inductive Ts L A := Leaf (l : L) | Node (a : A).          (* shapes *)
```

```
Inductive Tp := Lbranch | Rbranch.                        (* positions *)
```

```
Definition t_dom (x : Ts L A * Tp) : bool :=  
  match x with (Node _, _) => true | _ => false end.
```

```
Instance TreeF L A : Cont (Ts L A) Tp := { valid := t_dom }.
```

t_dom	Lbranch	Rbranch
Leaf tt	false	false
Node h	true	true

Setoids and Morphisms

- Every type has an equivalence `=e`.
- $A \rightsquigarrow B$: functions that respect it.
- Laws become `rewrite` steps.

Part I in One Slide

Problem

`Obj.magic`

axiom K

function extensionality

Cause

dependent positions

dependent pairs

= on functions

Fix

one type of positions

boolean validity

setoids

Part II

Recursive Coalgebras and Hylomorphisms

Why Not Just Write Quicksort?

```
Fixpoint qsort (l : list nat) : list nat :=  
  match l with  
  | [] => []  
  | h :: t => qsort (filter (fun x => x <? h) t)  
              ++ h :: qsort (filter (fun x => negb (x <? h)) t)  
  end.
```

Error: Recursive call to qsort has principal argument equal to
"filter (fun x : nat => negb (x <? h)) t" instead of "t".

Recursive Coalgebras (I)

$$h = a \circ F h \circ c \quad \text{with } F X = X, \text{ so } F h = h$$

Recursive Coalgebras (I)

$$h = a \circ F h \circ c \quad \text{with } F X = X, \text{ so } F h = h$$

$$c = \text{id}, a = \text{succ} \quad h x = h x + 1 \quad \text{no solution}$$

Recursive Coalgebras (I)

$$h = a \circ F h \circ c \quad \text{with } F X = X, \text{ so } F h = h$$

$$c = \text{id}, a = \text{succ} \quad h x = h x + 1 \quad \text{no solution}$$

$$c = a = \text{id} \quad h x = h x \quad \text{every } h \text{ is a solution}$$

Recursive Coalgebras (I)

$$h = a \circ F h \circ c \quad \text{with } F X = X, \text{ so } F h = h$$

$$c = \text{id}, a = \text{succ} \quad h x = h x + 1 \quad \text{no solution}$$

$$c = a = \text{id} \quad h x = h x \quad \text{every } h \text{ is a solution}$$

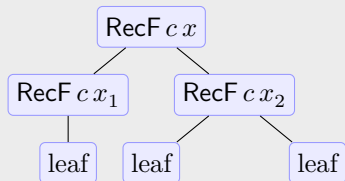
The culprit: $c = \text{id}$ never shrinks its input, $x \mapsto x \mapsto x \mapsto \dots$

Recursive c : every input reaches the leaves; then there is exactly one h , for every a .

Recursive Coalgebras (and II)

```
Inductive RecF (c : Coalg F A) : A -> Prop :=  
| RecF_fold x : (forall e, RecF c (cont (c x) e)) -> RecF c x.
```

```
Structure RCoalg F A :=  
  Rec { coalg :> Coalg F A; recP : forall x, RecF coalg x }.
```



Equivalent to `well_founded (child c)`, but the induction hypothesis is *per position*, as in `fmap`.

Recursive Hylomorphisms

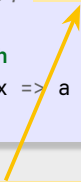
Recursion on the termination proof (simplified):

```
Definition hylo_def (a : Alg F B) (c : Coalg F A)
  : forall (x : A), RecF c x -> B :=
  fix f x H :=
    match c x with
    | MkCont sx cx => a (MkCont sx (fun e => f (cx e) (RecF_inv H e)))
  end.
```

Recursive Hylomorphisms

Recursion on the termination proof (simplified):

```
Definition hylo_def (a : Alg F B) (c : Coalg F A)
  : forall (x : A), RecF c x -> B :=
fix f x H :=
  match c x with
  | MkCont sx cx => a (MkCont sx (fun e => f (cx e) (RecF_inv H e)))
end.
```



termination proof
(in Prop: erased)

Recursive Hyломorphisms

Recursion on the termination proof (simplified):

```
Definition hylo_def (a : Alg F B) (c : Coalg F A)
  : forall (x : A), RecF c x -> B :=
fix f x H :=
  match c x with
  | MkCont sx cx => a (MkCont sx (fun e => f (cx e) (RecF_inv H e)))
end.
```

termination proof
(in Prop: erased)

structurally smaller proof

Uniqueness and the Laws

Lemma `hylo_uniq` : `f =e hylo a c <-> f =e a \o fmap f \o c.`

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` : `d \o b =e id -> hylo a d \o hylo b c =e hylo a c.`

Uniqueness and the Laws

Lemma `hylo_uniq` : `f =e hylo a c <-> f =e a \o fmap f \o c.`

Proofs follow the paper proofs:

$$\begin{array}{ccccc} F C & \xrightarrow{F \llbracket a, c \rrbracket} & F A & \xrightarrow{F h} & F B \\ \uparrow c & & \downarrow a & & \downarrow b \\ C & \xrightarrow{\llbracket a, c \rrbracket} & A & \xrightarrow{h} & B \end{array}$$

Folds after Unfolds

Definition `rana (c : RCoalg F A) : A ~> LFix F := hylo l_in c.`

Corollary `cata_ana_hylo : cata a \o rana c =e hylo a c.`

$$\begin{array}{ccccc} A & \xrightarrow{\text{rana } c} & \text{LFix } F & \xrightarrow{\text{cata } a} & B \\ & \searrow & & \nearrow & \\ & & \text{hylo } a \text{ } c & & \end{array}$$

No Axioms

Print Assumptions hylo_fusion_l.	Closed under the global context
Print Assumptions mss_calc.	Closed under the global context
Print Assumptions qsort_times_two_calc.	Axioms: PrimInt63.int, ltb, mul

Part III

Program Calculation and Extraction

Calculating a Program with Derive

```
Derive qsort in (cata merge \o rana tsplit => qsort) as qsort_calc.
```

```
Proof.
```

```
  rewrite cata_ana_hylo.
```

```
  finished.
```

```
Qed.
```

```
let rec qsort_code = function
```

```
| [] -> []
```

```
| h :: t ->
```

```
  let (l, r) = partition (fun x0 -> ltb x0 h) t in
```

```
  let x0 = fun e -> qsort_code (match e with Lbranch -> l | Rbranch -> r) in
```

```
  app (x0 Lbranch) (h :: (x0 Rbranch))
```

Fusing a Map into Quicksort

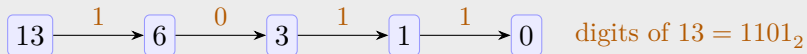
```
Derive qsort_times_two
  in (Lmap times_two \o cata merge \o rana tsplit =e qsort_times_two)
  as calc.
Proof.
  rewrite map_merge_fusion, hylo_ana, deforest by apply l_out_in. finished.
Qed.
```

```
let rec qsort_times_two_code = function
| [] -> []
| h :: t -> ... (* as qsort_code *)
  app (x0 Lbranch) ((mul (Uint63.of_int (2)) h) :: (x0 Rbranch))
```

Coalgebras Need Not Follow the Data

```
Definition digits_ (b n : int) : App (ListF int) int :=  
  MkCont (if n =? 0 then s_nil else s_cons (n mod b)) (fun _ => n / b).
```

```
Derive fast_power in (cata (pow_alg x) \o rana (digits 2) =e fast_power).
```



```
let rec fast_power_code x n =  
  if eqb n (UInt63.of_int (0)) then (UInt63.of_int (1))  
  else mul (digit_pow x (mod0 n (UInt63.of_int (2))))  
    (sq (fast_power_code x (div n (UInt63.of_int (2)))))
```

Challenge: Inlining for Extraction

Plain Recursive Extraction qsort:

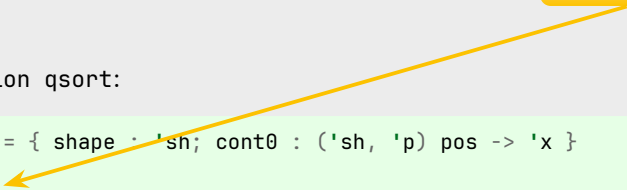
```
type ('sh, 'p, 'x) app0 = { shape : 'sh; cont0 : ('sh, 'p) pos -> 'x }

let rec hylo esh f eA eB g h x =
  let { shape = sx; cont0 = cx } = h x in
  g { shape = sx; cont0 = (fun e -> hylo esh f eA eB g h (cx e)) }

let qsort = hylo def_eq treeF (list_eq def_eq) (list_eq def_eq) merge tsplit
```

Challenge: Inlining for Extraction

dictionaries at run time



Plain Recursive Extraction qsort:

```
type ('sh, 'p, 'x) app0 = { shape : 'sh; cont0 : ('sh, 'p) pos -> 'x }  
  
let rec hylo esh f eA eB g h x =  
  let { shape = sx; cont0 = cx } = h x in  
  g { shape = sx; cont0 = (fun e -> hylo esh f eA eB g h (cx e)) }  
  
let qsort = hylo def_eq treeF (list_eq def_eq) (list_eq def_eq) merge tsplit
```

Challenge: Inlining for Extraction

dictionaries at run time

Plain Recursive Extraction qsort:

```
type ('sh, 'p, 'x) app0 = { shape : 'sh; cont0 : ('sh, 'p) pos -> 'x }  
  
let rec hylo esh f eA eB : h x =  
  let { shape = sx; cont0 = cx } = h x in  
  g { shape = sx; cont0 = (fun e -> hylo esh f eA eB g h (cx e)) }  
  
let qsort = hylo def_eq treeF (list_eq def_eq) (list_eq def_eq) merge tsplit
```

a record per call

Since the Paper: Usability

- Properness and termination: **instance search**.

Since the Paper: Usability

- Properness and termination: **instance search**.
- Rewriting modulo associativity: **arewrite**.

Since the Paper: Usability

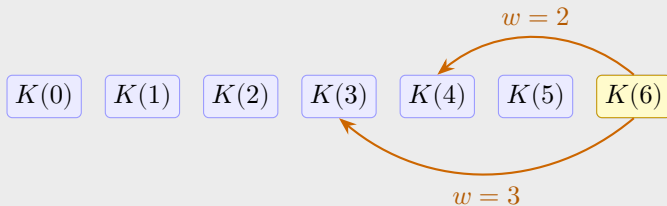
- Properness and termination: **instance search**.
- Rewriting modulo associativity: **arewrite**.
- Diverging instance search: `#[mode = "!"]`.

Part IV

Dynamic Programming

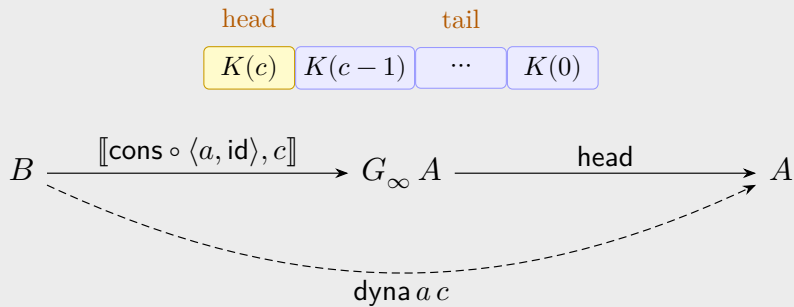
Knapsack

$$K(0) = 0 \quad K(c) = \max \left(\{0\} \cup \{ K(c - w_i) + v_i \mid 0 < w_i \leq c \} \right)$$



A plain hylomorphism recomputes: exponential.

Dynamorphisms: Algebras with a Memory



Knapsack in Rocq

Instance Memo : Cont (A * Sg) Pg := ... *(* Memo X = A * G X *)*

Definition Table := LFix Memo.

Definition dyna (a : App C (Table C A) ~> A) (c : RCoalg C B) : B ~> A
:= headT \o hylo (l_in \o Cons \o pair a id) c.

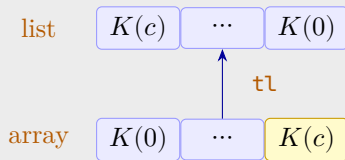
Definition knapsack_alg wvs (x : App NatF (Table NatF int)) : int :=
 match x with
 | MkCont (inl tt) _ => 0 *(* K(0) *)*
 | MkCont (inr tt) k => max_int 0 (memo_knapsack (k next) wvs)
 end. *(* memo_knapsack: u + v for each (w, v) with K(c - w) = u *)*

Definition knapsack wvs := dyna (knapsack_alg wvs) out_int_c.

From Tables to Native Arrays

```
Lemma table_to_list (* by hylo_fusion_1 *)  
  : to_list \o hylo (l_in \o Cons \o pair a id) c =e hylo (list_alg a') c.
```

```
Lemma dyna_array : to_Z n < to_Z max_length ->  
  hd d (int_hylo (list_alg a') n)  
  = (fst (int_hylo (arr_alg (n + 1) a'') n)).[n].
```



Knapsack: the Extracted Program

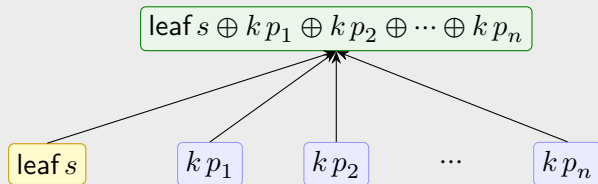
```
let rec memo_arr t c = function
| [] -> []
| (w, v) :: wvs ->
  if 0 < w && w <= c then (Parray.get t (c - w) + v) :: memo_arr t c wvs
  else memo_arr t c wvs

(* the loop, for c = 1 .. n: *)
Parray.set t c (max_int 0 (memo_arr t c wvs))
```

Part V

Monoid Algebras

Monoid Algebras



children, in the order of members

```
Definition malg_ (leaf : Sh ~> option W) (x : App F W) : W :=  
  let (s, k) := x in mconcat M (opt_cons (leaf s) (map k (members F s))).
```

```
Definition reduce (w : A ~> W) : list A ~> W :=  
  hylo (malg (ListF A) M (cons_leaf w)) ilist_coalg.
```

One Fusion Law for Every Monoid Algebra

```

Lemma malg_fusion (Hh : MonoidHom M N h)
  : h \o hylo (malg F M leaf) c =e hylo (malg F N (omap h \o leaf)) c.
Proof. apply hylo_fusion_l, malg_hom. Qed.

```

$$A \xrightarrow{[\text{malg}_M, c]} W \xrightarrow{h} V$$

$\underbrace{\hspace{10em}}_{[\text{malg}_N, c]}$

Maximum Segment Sum

Definition `mss := maximum \o lmap sum \o segs \o lmap some_m.`

Derive `mss_linear` **in** `(mss =e mss_linear)` **as** `mss_calc.`

Proof.

`unfold mss, segs.`

`arewrite (map_promotion sum).`

`arewrite (reduce_promotion max_monoid).`

`...`

(lmap_comp, twice *)*

`rewrite horner_mss.`

(max-plus semiring *)*

`arewrite (scan_lemma odot (Some 0)).`

`...`

(tupling *)*

`finished.`

Qed.

Maximum Segment Sum

```
let mss_code xs =  
  let rec f = function  
    | [] -> (Some 0, Some 0)  
    | h :: t ->  
      let (m, s) = f t in  
      let s' = match s with  
        | Some b -> Some (max 0 (h + b))  
        | None -> Some 0 in  
      (omax s' m, s')  
  in fst (f xs)
```

Wrap-up

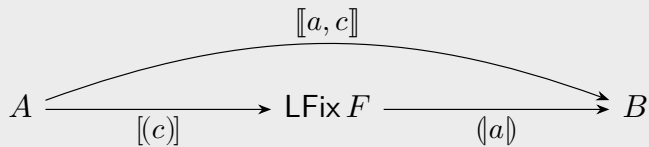
Summary

- Hylomorphisms in Rocq: axiom-free, extractable.
- Laws are theorems; calculations are `rewrite` proofs.
- Termination proved once per coalgebra.
- Idiomatic OCaml, down to native arrays.

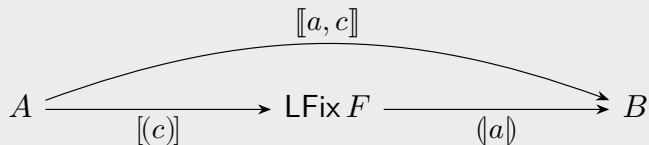
What Changed Since the Paper

- Rocq 9; properness and termination by instance search.
- `Derive` and `arewrite` for calculations.
- Finite containers and monoid algebras.
- New examples: MSS, fast power, knapsack with arrays.

Open Question 1: Which Intermediate Structure?

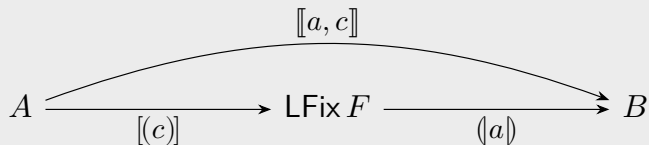


Open Question 1: Which Intermediate Structure?



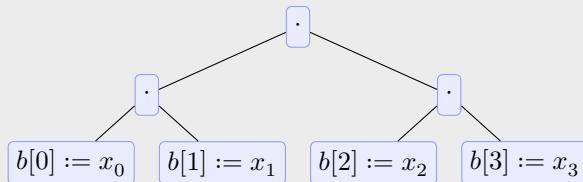
- A law that splits through a **buffer**, not a tree?

Open Question 1: Which Intermediate Structure?



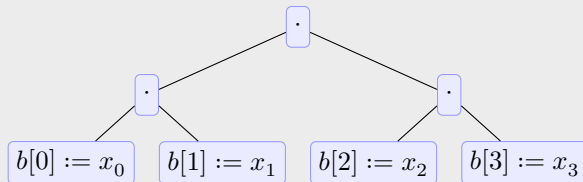
- A law that splits through a **buffer**, not a tree?
- Fused or split: a question of **cost**.

Open Question 2: Algebras that Write



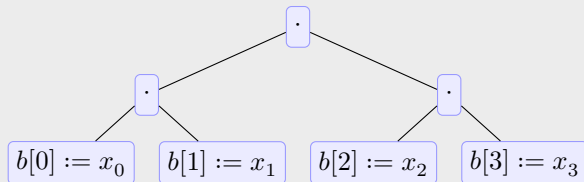
- Writes form a monoid: `malg_fusion` applies.

Open Question 2: Algebras that Write



- Writes form a monoid: `malg_fusion` applies.
- Which equivalence on buffers?

Open Question 2: Algebras that Write



- Writes form a monoid: `malg_fusion` applies.
- Which equivalence on buffers?
- Exactly once, proved per coalgebra?

| Open Question 3: Programs Without Closures

- `cont` is a function: extraction gives closures.

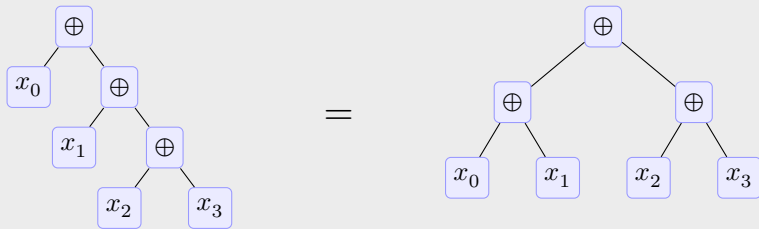
Open Question 3: Programs Without Closures

- `cont` is a function: extraction gives closures.
- A first-order syntax, **closed under the laws?**

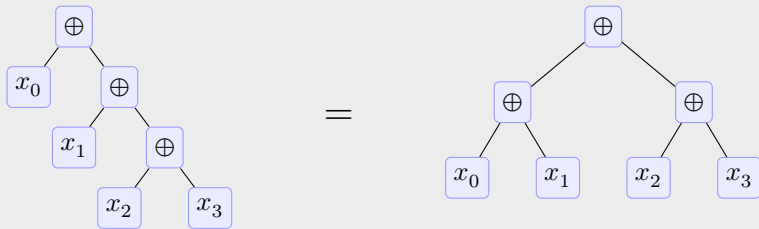
Open Question 3: Programs Without Closures

- `cont` is a function: extraction gives closures.
- A first-order syntax, **closed under the laws?**
- Where do overflow and bounds conditions live?

Open Question 4: Changing the Coalgebra

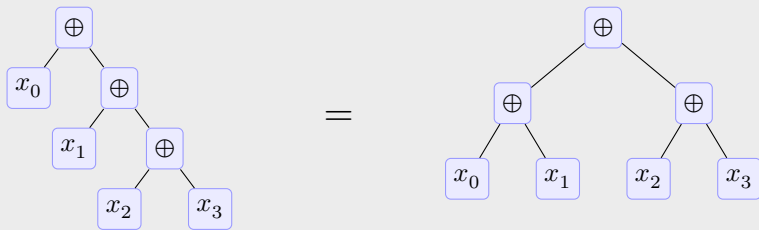


Open Question 4: Changing the Coalgebra



- A generic law: same leaves, same order, same result?

Open Question 4: Changing the Coalgebra



- A generic law: same leaves, same order, same result?
- And if the order differs?

Thank you!

Questions?

`https://github.com/
dcastrop/rocq-hylomorphisms`

Based on joint work with Marco
Paviotti and Michael Vollmer.