Program Calculation with Hylomorphisms in Rocq

David Castro-Perez

David.CastroPerez@rhul.ac.uk



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

- Part II: joint, Marco Paviotti, Michael Vollmer (ITP 2025)
- Part III: since ITP 2025

Rewrite Rules You Already Trust

A database optimiser:

```sql
SELECT o.id
FROM orders o JOIN users u ON o.user_id = u.id
WHERE u.country = 'UK';
-- the optimiser filters users before joining:
--   filter(join(orders, users)) = join(orders, filter(users))
```

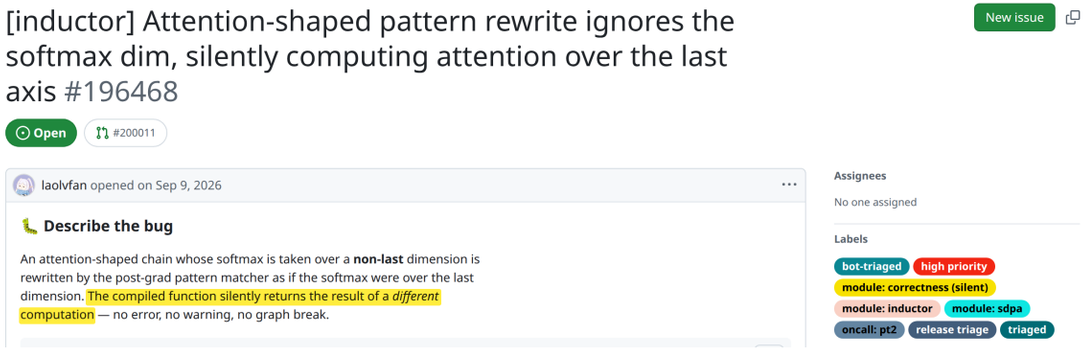
An ML compiler:

```python
def attn(q, k, v):
    s = torch.softmax(q @ k.transpose(-2, -1) / 8, dim=-1)
    return s @ v                    # matmul, softmax, matmul
fast = torch.compile(attn)          # rewritten: one fused attention op
```


Rewrite Rules You Already Trust

- SQL pushdown; PyTorch attention fusion
- Bug: dim 0 silently run as dim -1
- Trust needs proved laws, checked side conditions

Should you really trust them?



[inductor] Attention-shaped pattern rewrite ignores the softmax dim, silently computing attention over the last axis #196468

New issue

Open #200011

laolvfan opened on Sep 9, 2026

Describe the bug

An attention-shaped chain whose softmax is taken over a **non-last** dimension is rewritten by the post-grad pattern matcher as if the softmax were over the last dimension. The compiled function silently returns the result of a *different* computation — no error, no warning, no graph break.

Assignees

No one assigned

Labels

bot-triaged · high priority · module: correctness (silent) · module: inductor · module: sdpa · oncall: pt2 · release triage · triaged

Write `dim=0` instead, and the same rewrite still fires: the compiled code ignores it and normalises over the last axis, as if you had written `dim=-1`. A rewrite rule whose side condition (softmax over the **last** axis) is never checked.

https://github.com/pytorch/pytorch/issues/196468

Rewrite Rules You Already Trust

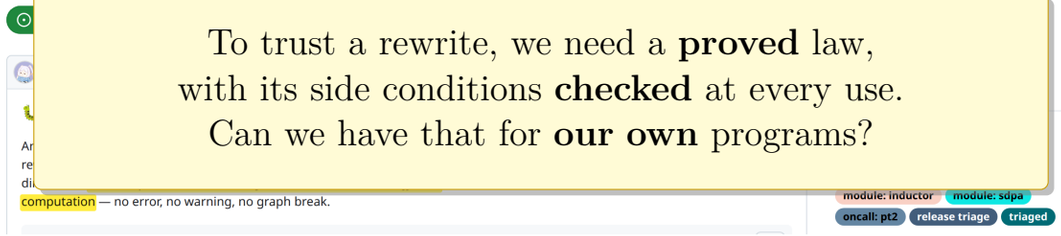
- SQL pushdown; PyTorch attention fusion
- Bug: dim 0 silently run as dim -1
- Trust needs proved laws, checked side conditions

Rewrite Rules You Already Trust

Should you really trust them?

[inductor] Attention-shaped pattern rewrite ignores the softmax dim, silently computing attention over the last axis #196468

New issue



To trust a rewrite, we need a **proved** law,
with its side conditions **checked** at every use.
Can we have that for **our own** programs?

Write `dim=0` instead, and the same rewrite still fires: the compiled code ignores it and normalises over the last axis, as if you had written `dim=-1`. A rewrite rule whose side condition (softmax over the **last** axis) is never checked.

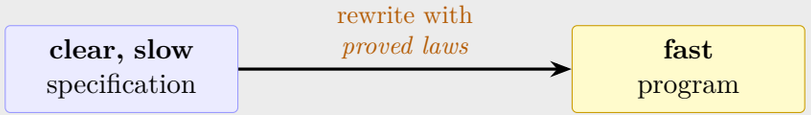
https://github.com/pytorch/pytorch/issues/196468

Rewrite Rules You Already Trust

slide 4 / 129 step 3 of 3

- SQL pushdown; PyTorch attention fusion
- Bug: dim 0 silently run as dim -1
- Trust needs proved laws, checked side conditions

This Talk



1. **Program calculation**: recursion schemes, their laws, and parallelism.
   The classic theory; our ICFP 2016 work.
2. **Hylomorphisms in Rocq**: containers, recursive coalgebras, the laws; no axioms; clean extracted code.
   Joint work with M. Paviotti and M. Vollmer (ITP 2025).
3. **Calculating in Rocq**: `Derive`, clean extraction, dynamic programming over arrays, monoid algebras.
   Since ITP 2025.

This Talk

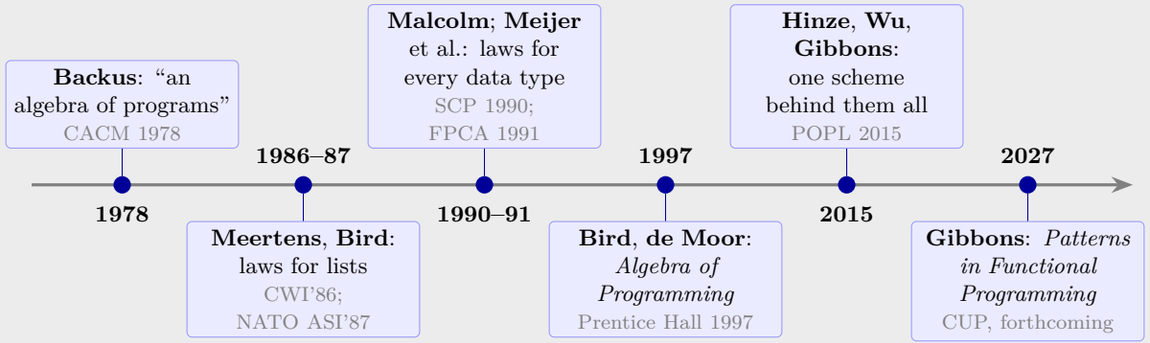
- Spec in, fast program out, every step an equation
- Theory; the library; calculations

Part I

Program Calculation

Programs as Algebra

Specify *clearly*; **calculate** an efficient program; every step is an *equation*.



J. Backus. **Can Programming Be Liberated from the von Neumann Style?** CACM 1978.
R. Bird. **An Introduction to the Theory of Lists**. 1987.
G. Malcolm. **Data Structures and Program Transformation**. SCP 1990.
E. Meijer, M. Fokkinga, R. Paterson. **Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire**. FPCA 1991.
R. Bird, O. de Moor. **Algebra of Programming**. Prentice Hall, 1997.
R. Hinze, N. Wu, J. Gibbons. **Conjugate Hylomorphisms, or: The Mother of All Structured Recursion Schemes**. POPL 2015.
J. Gibbons. **Patterns in Functional Programming**. CUP, forthcoming (2027).

Programs as Algebra

- Backus 1978 to Gibbons 2027
- Quote: Backus, Turing lecture

Programs as Algebra

Specify *clearly*; **calculate** an efficient program; every step is an *equation*.

> "Conventional programming languages are growing ever more enormous, but not stronger. [...] Associated with the functional style of programming is an **algebra of programs** whose variables range over programs and whose operations are combining forms. This algebra can be used to **transform programs** and to solve equations whose 'unknowns' are programs in much the same way one transforms equations in high school algebra. These transformations are given by algebraic laws [...]"
>
> — J. Backus, Can Programming Be Liberated from the von Neumann Style? Turing Award lecture, CACM 21(8), 1978

J. Backus. **Can Programming Be Liberated from the von Neumann Style?** CACM 1978.
R. Bird. **An Introduction to the Theory of Lists**. 1987.
G. Malcolm. **Data Structures and Program Transformation**. SCP 1990.
E. Meijer, M. Fokkinga, R. Paterson. **Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire**. FPCA 1991.
R. Bird, O. de Moor. **Algebra of Programming**. Prentice Hall, 1997.
R. Hinze, N. Wu, J. Gibbons. **Conjugate Hylomorphisms, or: The Mother of All Structured Recursion Schemes**. POPL 2015.
J. Gibbons. **Patterns in Functional Programming**. CUP, forthcoming (2027).

Programs as Algebra

- Backus 1978 to Gibbons 2027
- Quote: Backus, Turing lecture

Initial Algebras and Catamorphisms

$$\mathsf{List}\,A \;\cong\; 1 + A \times \mathsf{List}\,A \;=\; F\,(\mathsf{List}\,A) \qquad \mathsf{in} = [\mathsf{nil}, \mathsf{cons}] : F\,(\mathsf{List}\,A) \to \mathsf{List}\,A$$

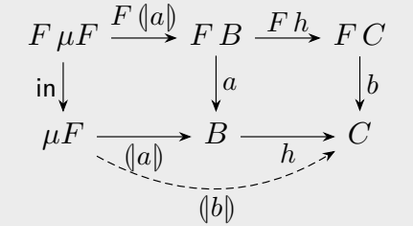
$$\begin{array}{ccc} F\,\mu F & \xrightarrow{F\,(\!|a|\!)} & F\,B \\ {\scriptstyle \mathsf{in}}\downarrow & & \downarrow{\scriptstyle a} \\ \mu F & \xrightarrow[(\!|a|\!)]{} & B \end{array} \qquad f = (\!|a|\!) \iff f \circ \mathsf{in} = a \circ F\,f$$

- An F-algebra $a : F\,B \to B$; the fold $(\!|a|\!)$ is the *unique* algebra morphism out of $(\mu F, \mathsf{in})$.
- For lists, $(\!|[\mathsf{base}, \mathsf{step}]|\!) = \mathsf{foldr}\ \mathsf{step}\ \mathsf{base}$.

Initial Algebras and Catamorphisms

- Universal property: every law below is uniqueness
- F f: fmap, recursive positions only

Fusion



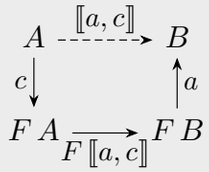
$$h \circ a = b \circ F\,h \implies h \circ (\!|a|\!) = (\!|b|\!)$$

Two traversals become one, if the **side condition** holds.

Fusion

- Side condition: h an algebra morphism
- Proof: outer rectangle commutes; uniqueness

Hylomorphisms: Divide and Conquer



$$[\![a, c]\!] = a \circ F\,[\![a, c]\!] \circ c$$

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

data TreeF x = Leaf | Node x Int x  deriving Functor   -- one layer

split []      = Leaf                                   -- coalgebra
split (p : t) = Node [x | x <- t, x < p] p [x | x <- t, x >= p]

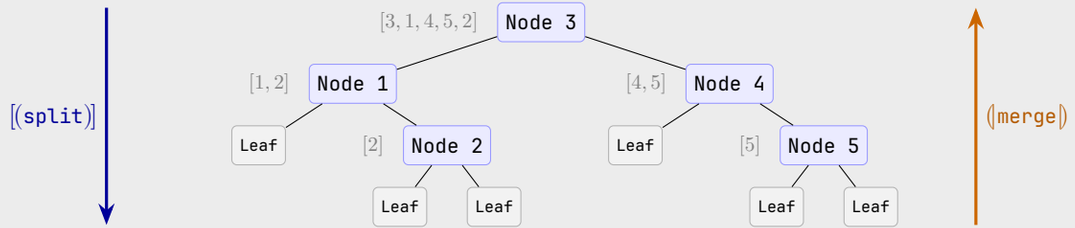
merge Leaf         = []                                -- algebra
merge (Node l p r) = l ++ [p] ++ r

qsort = hylo merge split
```


Hylomorphisms: Divide and Conquer

- c divides, a conquers
- Names as in the library: merge, split

The Call Tree of Quicksort

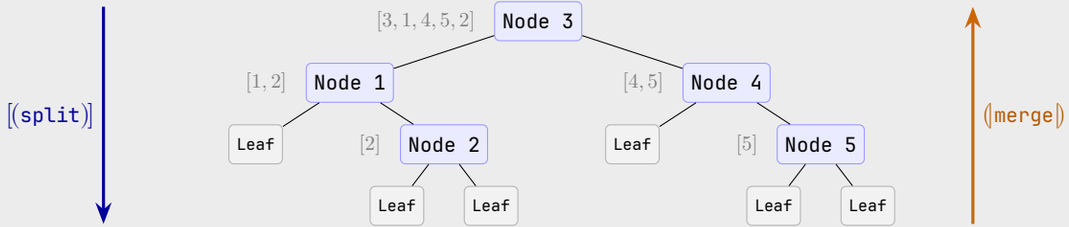


- A value of $\mu\,\texttt{TreeF}$: the **call tree**. $[\![a, c]\!] = (\!|a|\!) \circ [(c)]$, when the tree is finite.

The Call Tree of Quicksort

- Unfold builds it, fold consumes it
- hylo: never built

The Call Tree of Quicksort



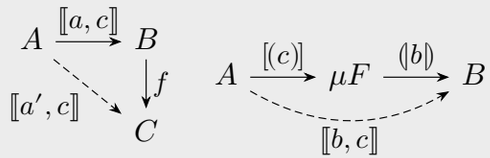
- A value of $\mu\,\texttt{TreeF}$: the **call tree**. $[\![a, c]\!] = (\!|a|\!) \circ [(c)]$, when the tree is finite.
- A hylomorphism consumes each layer as it is produced: the tree is **virtual**.

The Call Tree of Quicksort

- Unfold builds it, fold consumes it
- hylo: never built

Recursive Coalgebras and Their Laws

c *recursive*: a unique solution $[\![a, c]\!]$ for every a.



$$
\begin{array}{rcll}
f \circ a = a' \circ F\, f & \Longrightarrow & f \circ [\![a, c]\!] = [\![a', c]\!] & \text{(fusion, left)} \\
c \circ f = F\, f \circ c' & \Longrightarrow & [\![a, c]\!] \circ f = [\![a, c']\!] & \text{(fusion, right)} \\
c' \circ a = \mathsf{id} & \Longrightarrow & [\![b, c']\!] \circ [\![a, c]\!] = [\![b, c]\!] & \text{(deforestation)}
\end{array}
$$

V. Capretta, T. Uustalu, V. Vene. **Recursive Coalgebras from Comonads**. Inf. Comput. 2006.
R. Hinze, N. Wu, J. Gibbons. **Conjugate Hylomorphisms, or: The Mother of All Structured Recursion Schemes**. POPL 2015.

Recursive Coalgebras and Their Laws

- All three from uniqueness
- Deforestation: no intermediate structure

Calculating Sort-Then-Double

```
  map double . qsort
=   { definition }
  map double . hylo merge split
=   { fusion, left: map double . merge = merge2 . fmap (map double) }
  hylo merge2 split

merge2 = merge . double_pivot   -- double_pivot (Node l p r) = Node l (2*p) r
```


Calculating Sort-Then-Double

- Spec: sort, then double
- Side condition by hand; in Rocq, Part III

Calculating Sort-Then-Double

```
  map double . qsort
=   { definition }
  map double . hylo merge split
=   { fusion, left: map double . merge = merge2 . fmap (map double) }
  hylo merge2 split

merge2 = merge . double_pivot   -- double_pivot (Node l p r) = Node l (2*p) r
```

The **side condition** of this fusion, checked here only **by hand**:

```
map double (merge (Node l p r))
  ==  map double l ++ [double p] ++ map double r
  ==  merge2 (fmap (map double) (Node l p r))
```


Calculating Sort-Then-Double

- Spec: sort, then double
- Side condition by hand; in Rocq, Part III

Calculating Sort-Then-Double

```
  map double . qsort
=   { definition }
  map double . hylo merge split
=   { fusion, left: map double . merge = merge2 . fmap (map double) }
  hylo merge2 split

merge2 = merge . double_pivot   -- double_pivot (Node l p r) = Node l (2*p) r
```

The **side condition** of this fusion, checked here only **by hand**:

```
map double (merge (Node l p r))
  ==  map double l ++ [double p] ++ map double r
  ==  merge2 (fmap (map double) (Node l p r))
```

Two passes and a temporary list $\Longrightarrow$ **one pass**, doubling each pivot.

Calculating Sort-Then-Double

- Spec: sort, then double
- Side condition by hand; in Rocq, Part III

List Homomorphisms Are Hylomorphisms

```haskell
data SplitF x = Empty | One Int | Two x x  deriving Functor

splitBy cut []  = Empty
splitBy cut [x] = One x
splitBy cut xs  = Two l r  where (l, r) = cut xs  -- l ++ r == xs, l, r /= []

hom e f op cut = hylo alg (splitBy cut)           -- the cut is a parameter
  where alg Empty     = e
        alg (One x)   = f x
        alg (Two a b) = op a b                     -- op associative

halves xs = splitAt (length xs `div` 2) xs         -- depth log n: parallel
heads  xs = splitAt 1 xs                           -- depth n: a loop
```

R. Bird. **An Introduction to the Theory of Lists**. Logic of Programming and Calculi of Discrete Design, 1987.

List Homomorphisms Are Hylomorphisms

- The cut is a parameter of divide
- Halves: parallel; heads: a loop

List Homomorphisms Are Hylomorphisms

```haskell
data SplitF x = Empty | One Int | Two x x  deriving Functor

splitBy cut []  = Empty
splitBy cut [x] = One x
splitBy cut xs  = Two l r  where (l, r) = cut xs  -- l ++ r == xs, l, r /= []

hom e f op cut = hylo alg (splitBy cut)           -- the cut is a parameter
  where alg Empty     = e
        alg (One x)   = f x
        alg (Two a b) = op a b                     -- op associative

halves xs = splitAt (length xs `div` 2) xs         -- depth log n: parallel
heads  xs = splitAt 1 xs                           -- depth n: a loop
```

- $h\,(xs +\!\!+\, ys) = h\,xs \odot h\,ys$: **first homomorphism theorem**, $h = \mathsf{reduce}\,(\odot) \circ \mathsf{map}\,f$.

R. Bird. **An Introduction to the Theory of Lists**. Logic of Programming and Calculi of Discrete Design, 1987.

List Homomorphisms Are Hylomorphisms

- The cut is a parameter of divide
- Halves: parallel; heads: a loop

List Homomorphisms Are Hylomorphisms

```haskell
data SplitF x = Empty | One Int | Two x x  deriving Functor

splitBy cut []  = Empty
splitBy cut [x] = One x
splitBy cut xs  = Two l r  where (l, r) = cut xs  -- l ++ r == xs, l, r /= []

hom e f op cut = hylo alg (splitBy cut)           -- the cut is a parameter
  where alg Empty     = e
        alg (One x)   = f x
        alg (Two a b) = op a b                     -- op associative

halves xs = splitAt (length xs `div` 2) xs         -- depth log n: parallel
heads  xs = splitAt 1 xs                           -- depth n: a loop
```

- $h\,(xs +\!\!+ ys) = h\,xs \odot h\,ys$: **first homomorphism theorem**, $h = \mathsf{reduce}\,(\odot) \circ \mathsf{map}\,f$.
- Any cut, same result (induction on the call tree): the cut is a **scheduling** decision.

R. Bird. **An Introduction to the Theory of Lists**. Logic of Programming and Calculi of Discrete Design, 1987.

List Homomorphisms Are Hylomorphisms

- The cut is a parameter of divide
- Halves: parallel; heads: a loop

The Third Homomorphism Theorem

Theorem. If h is both leftwards and rightwards,

$$h\,(x : xs) = x \oplus h\,xs \qquad \text{and} \qquad h\,(xs +\!\!+ [x]) = h\,xs \otimes x,$$

then h is a list homomorphism, with

```
combine a b = h (g a ++ g b)    -- g a: any list xs with h xs == a
```

J. Gibbons. **The Third Homomorphism Theorem**. JFP 1996. (Conjectured by Bird, proved by Meertens, 1989.)
K. Morita et al. **Automatic Inversion Generates Divide-and-Conquer Parallel Programs**. PLDI 2007.
A. Farzan, V. Nicolet. **Synthesis of Divide and Conquer Parallelism for Loops**. PLDI 2017.

The Third Homomorphism Theorem

- Two sequential programs agree: a parallel one exists
- g: a right inverse of h

The Third Homomorphism Theorem

Theorem. If h is both leftwards and rightwards,

$$h\,(x : xs) = x \oplus h\,xs \qquad \text{and} \qquad h\,(xs +\!\!+ [x]) = h\,xs \otimes x,$$

then h is a list homomorphism, with

```
combine a b = h (g a ++ g b)    -- g a: any list xs with h xs == a
```

- A loop from the left remembers only $h\,xs$; one from the right only $h\,ys$. So $h\,(xs +\!\!+ ys)$ depends only on both.

J. Gibbons. **The Third Homomorphism Theorem**. JFP 1996. (Conjectured by Bird, proved by Meertens, 1989.)
K. Morita et al. **Automatic Inversion Generates Divide-and-Conquer Parallel Programs**. PLDI 2007.
A. Farzan, V. Nicolet. **Synthesis of Divide and Conquer Parallelism for Loops**. PLDI 2017.

The Third Homomorphism Theorem

- Two sequential programs agree: a parallel one exists
- g: a right inverse of h

The Third Homomorphism Theorem

Theorem. If h is both leftwards and rightwards,

$$h\,(x : xs) = x \oplus h\,xs \qquad \text{and} \qquad h\,(xs +\!\!+ [x]) = h\,xs \otimes x,$$

then h is a list homomorphism, with

```
combine a b = h (g a ++ g b)    -- g a: any list xs with h xs == a
```

- A loop from the left remembers only $h\,xs$; one from the right only $h\,ys$. So $h\,(xs +\!\!+ ys)$ depends only on both.
- Insertion sort, from either side $\Longrightarrow$ mergesort, by calculation.

J. Gibbons. **The Third Homomorphism Theorem**. JFP 1996. (Conjectured by Bird, proved by Meertens, 1989.)
K. Morita et al. **Automatic Inversion Generates Divide-and-Conquer Parallel Programs**. PLDI 2007.
A. Farzan, V. Nicolet. **Synthesis of Divide and Conquer Parallelism for Loops**. PLDI 2017.

The Third Homomorphism Theorem

- Two sequential programs agree: a parallel one exists
- g: a right inverse of h

The Third Homomorphism Theorem

Theorem. If h is both leftwards and rightwards,

$$h\,(x : xs) = x \oplus h\,xs \qquad \text{and} \qquad h\,(xs +\!\!+ [x]) = h\,xs \otimes x,$$

then h is a list homomorphism, with

```
combine a b = h (g a ++ g b)    -- g a: any list xs with h xs == a
```

- A loop from the left remembers only $h\,xs$; one from the right only $h\,ys$. So $h\,(xs +\!\!+ ys)$ depends only on both.
- Insertion sort, from either side $\Longrightarrow$ mergesort, by calculation.
- Combiners derived automatically: weak right inverses (Morita et al.); synthesised for loops (Farzan and Nicolet).

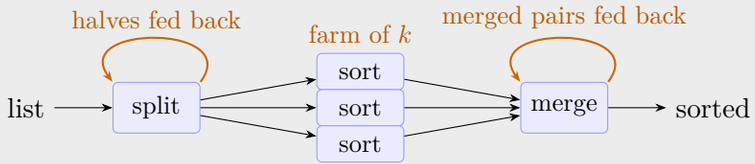
J. Gibbons. **The Third Homomorphism Theorem**. JFP 1996. (Conjectured by Bird, proved by Meertens, 1989.)
K. Morita et al. **Automatic Inversion Generates Divide-and-Conquer Parallel Programs**. PLDI 2007.
A. Farzan, V. Nicolet. **Synthesis of Divide and Conquer Parallelism for Loops**. PLDI 2017.

The Third Homomorphism Theorem

- Two sequential programs agree: a parallel one exists
- g: a right inverse of h

One Specification, Many Parallel Programs

```
msort = hylo merge halves                    -- merge: two sorted lists
cutoff n xs | length xs > n = Left (halves xs)  -- halves: fed back
            | otherwise     = Right xs          -- short: passed on
```



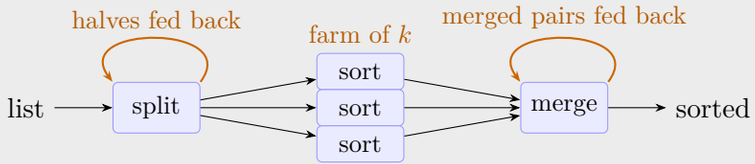
D. Castro, K. Hammond, S. Sarkar. **Farms, Pipes, Streams and Reforestation: Reasoning about Structured Parallel Processes using Types and Hylomorphisms**. ICFP 2016.
D. Castro, K. Hammond, S. Sarkar, Y. Alguwaifli. **Automatically Deriving Cost Models for Structured Parallel Processes Using Hylomorphisms**. FGCS 2018.

One Specification, Many Parallel Programs

- Ours, ICFP 2016
- Reforestation exposes stages
- Cost model picks sigma, n, k

One Specification, Many Parallel Programs

```
msort = hylo merge halves                    -- merge: two sorted lists
cutoff n xs | length xs > n = Left (halves xs)  -- halves: fed back
            | otherwise     = Right xs          -- short: passed on
```



$$\mathsf{psort} : \mathsf{List\ Int} \xrightarrow{\sigma} \mathsf{List\ Int} \qquad \sigma = \mathsf{FB}\,(\mathsf{FUN}\,A) \parallel \mathsf{FARM}\,k\,(\mathsf{FUN}\,A) \parallel \mathsf{FB}\,(\mathsf{FUN}\,A)$$

Types carry the parallel structure; type checking: is there an equal program with this structure (normalise with the hylo laws, unify)? A **cost model** picks σ, n and k.

D. Castro, K. Hammond, S. Sarkar. **Farms, Pipes, Streams and Reforestation: Reasoning about Structured Parallel Processes using Types and Hylomorphisms**. ICFP 2016.
D. Castro, K. Hammond, S. Sarkar, Y. Alguwaifli. **Automatically Deriving Cost Models for Structured Parallel Processes Using Hylomorphisms**. FGCS 2018.

One Specification, Many Parallel Programs

- Ours, ICFP 2016
- Reforestation exposes stages
- Cost model picks sigma, n, k

What Was Missing

- **Laws as axioms**: the ICFP 2016 type system assumed them (sound for a cpo semantics); programs could diverge.

Tools: HYLO (Hu et al., ICFP'96); MAG (de Moor, Sittampalam, TCS'01); Pointless Haskell, DrHylo (Cunha et al., IFL'05); SkeTo (Matsuzaki, Emoto, HLPP'10); HERMIT (Farmer et al., Haskell'12).

What Was Missing

- Laws assumed; side conditions on trust
- Same in other tools
- So: a proof assistant

What Was Missing

- **Laws as axioms**: the ICFP 2016 type system assumed them (sound for a cpo semantics); programs could diverge.
- **Side conditions on trust**: `hylo merge halves` becomes `... || fb (fun merge)` only if `merge` is associative and commutative; the library took the programmer's word for it.

Tools: HYLO (Hu et al., ICFP'96); MAG (de Moor, Sittampalam, TCS'01); Pointless Haskell, DrHylo (Cunha et al., IFL'05); SkeTo (Matsuzaki, Emoto, HLPP'10); HERMIT (Farmer et al., Haskell'12).

What Was Missing

- Laws assumed; side conditions on trust
- Same in other tools
- So: a proof assistant

What Was Missing

- **Laws as axioms**: the ICFP 2016 type system assumed them (sound for a cpo semantics); programs could diverge.
- **Side conditions on trust**: `hylo merge halves` becomes `... || fb (fun merge)` only if `merge` is associative and commutative; the library took the programmer's word for it.
- **Unverified tools**: they implement the schemes or the rewrites; calculations by hand or by trusted rewriting.

Tools: HYLO (Hu et al., ICFP'96); MAG (de Moor, Sittampalam, TCS'01); Pointless Haskell, DrHylo (Cunha et al., IFL'05); SkeTo (Matsuzaki, Emoto, HLPP'10); HERMIT (Farmer et al., Haskell'12).

What Was Missing

- Laws assumed; side conditions on trust
- Same in other tools
- So: a proof assistant

What Was Missing

- **Laws as axioms**: the ICFP 2016 type system assumed them (sound for a cpo semantics); programs could diverge.
- **Side conditions on trust**: `hylo merge halves` becomes `... || fb (fun merge)` only if `merge` is associative and commutative; the library took the programmer's word for it.
- **Unverified tools**: they implement the schemes or the rewrites; calculations by hand or by trusted rewriting.
- **No idiomatic code**: calculated programs stay higher-order; we trust the compiler to inline them.

Tools: HYLO (Hu et al., ICFP'96); MAG (de Moor, Sittampalam, TCS'01); Pointless Haskell, DrHylo (Cunha et al., IFL'05); SkeTo (Matsuzaki, Emoto, HLPP'10); HERMIT (Farmer et al., Haskell'12).

What Was Missing

- Laws assumed; side conditions on trust
- Same in other tools
- So: a proof assistant

What Was Missing

- **Laws as axioms**: the ICFP 2016 type system assumed them (sound for a cpo semantics); programs could diverge.
- **Side conditions on trust**: `hylo merge halves` becomes `... || fb (fun merge)` only if `merge` is associative and commutative; the library took the programmer's word for it.
- **Unverified tools**: they implement the schemes or the rewrites; calculations by hand or by trusted rewriting.
- **No idiomatic code**: calculated programs stay higher-order; we trust the compiler to inline them.

Can a proof assistant **prove** each law once, **check** every rewrite, side conditions included, and still produce **efficient** code?

Tools: HYLO (Hu et al., ICFP'96); MAG (de Moor, Sittampalam, TCS'01); Pointless Haskell, DrHylo (Cunha et al., IFL'05); SkeTo (Matsuzaki, Emoto, HLPP'10); HERMIT (Farmer et al., Haskell'12).

What Was Missing

- Laws assumed; side conditions on trust
- Same in other tools
- So: a proof assistant

Part II

Hylomorphisms in Rocq

Joint work with Marco Paviotti and Michael Vollmer

Program Optimisations via Hylomorphisms for Extraction of Executable Code, ITP 2025

Program Calculation in Proof Assistants

Work		Approach	Limits
Mu, Ko, Jansson, JFP'09	Agda	relational derivations (AoPA)	extraction not discussed
Tesson et al., AMAST'10	Coq/Rocq	list homomorphisms (SyDPaCC), parallel OCaml (BSML)	lists only
Emoto et al., ITP'14; Loulergue et al., IJPP'17	Coq/Rocq	list homomorphisms, extracted to parallel code	lists only
Sculthorpe, Hutton, JFP'14	Agda	worker/wrapper, with its correctness conditions	one optimisation
Murata, Emoto, APLAS'19	Coq/Rocq	recursion schemes	no hylomorphisms, so no divide and conquer; function extensionality
Larchey-Wendling, Monin, 2022	Coq/Rocq	general recursion, extracted (the Braga method)	one algorithm at a time
Abreu et al., POPL'23	Coq/Rocq	divide and conquer, termination by typing	function extensionality, impredicative `Set`; extracted code with casts, hard to read

We want all of: **any** data type, laws as **equations**, **no axioms**, **clean** extracted code.

Program Calculation in Proof Assistants

- Each misses one of the four
- Q: Abreu et al.? Typing for termination; funext; casts

challenge: termination

Why Not Just Write `hylo`?

```
Fixpoint hylo {F : Type -> Type} {A B : Type}
    (fmap : forall X Y, (X -> Y) -> F X -> F Y)
    (conquer : F B -> B) (divide : A -> F A) (x : A) : B :=
  conquer (fmap A B (hylo fmap conquer divide) (divide x)).
```

```
Error: Cannot guess decreasing argument of fix.
```


Why Not Just Write `hylo`?

- Generic hylo: rejected
- Workarounds: per function, no laws

challenge: termination

Why Not Just Write `hylo`?

```
Fixpoint hylo {F : Type -> Type} {A B : Type}
    (fmap : forall X Y, (X -> Y) -> F X -> F Y)
    (conquer : F B -> B) (divide : A -> F A) (x : A) : B :=
  conquer (fmap A B (hylo fmap conquer divide) (divide x)).
```

```
Error: Cannot guess decreasing argument of fix.
```

- Termination depends on the coalgebra, which is a parameter.

Why Not Just Write `hylo`?

- Generic hylo: rejected
- Workarounds: per function, no laws

challenge: termination

Why Not Just Write `hylo`?

```
Fixpoint hylo {F : Type -> Type} {A B : Type}
    (fmap : forall X Y, (X -> Y) -> F X -> F Y)
    (conquer : F B -> B) (divide : A -> F A) (x : A) : B :=
  conquer (fmap A B (hylo fmap conquer divide) (divide x)).
```

```
Error: Cannot guess decreasing argument of fix.
```

- Termination depends on the coalgebra, which is a parameter.
- `Program Fixpoint`, `Equations`, `Function`: one termination proof per function, and **no laws**.

Why Not Just Write `hylo`?

- Generic hylo: rejected
- Workarounds: per function, no laws

Three Challenges

1. **No axioms**: no function extensionality, no axiom K.
2. **Clean extracted code**: what a programmer would write; no `Obj.magic`.
3. **Non-structural recursion**: divide and conquer.

Three Challenges

- Solutions are the rest of this part
- Each definition: for the proofs and for the code

Three Challenges

1. **No axioms**: no function extensionality, no axiom K.
2. **Clean extracted code**: what a programmer would write; no `Obj.magic`.
3. **Non-structural recursion**: divide and conquer.

Solutions:

1. Setoids; decidable validity of positions (UIP for free).

Three Challenges

- Solutions are the rest of this part
- Each definition: for the proofs and for the code

Three Challenges

1. **No axioms**: no function extensionality, no axiom K.
2. **Clean extracted code**: what a programmer would write; no `Obj.magic`.
3. **Non-structural recursion**: divide and conquer.

Solutions:

1. Setoids; decidable validity of positions (UIP for free).
2. One type of positions for all shapes, not a family.

Three Challenges

- Solutions are the rest of this part
- Each definition: for the proofs and for the code

Three Challenges

1. **No axioms**: no function extensionality, no axiom K.
2. **Clean extracted code**: what a programmer would write; no `Obj.magic`.
3. **Non-structural recursion**: divide and conquer.

Solutions:

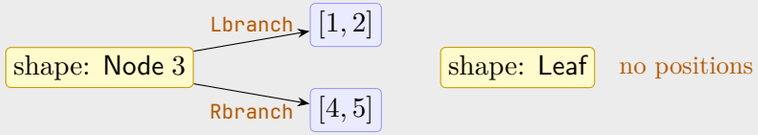
1. Setoids; decidable validity of positions (UIP for free).
2. One type of positions for all shapes, not a family.
3. Recursive coalgebras: recursion on a termination proof.

Three Challenges

- Solutions are the rest of this part
- Each definition: for the proofs and for the code

Containers

$$[\![S \lhd P]\!]\, X = \sum_{s:S} (P\, s \to X) \qquad [\![S \lhd P]\!]\, f\, (s, g) = (s, f \circ g)$$



Every polynomial functor is a container: one `fmap`, one `hylo`, one set of laws.

M. Abbott, T. Altenkirch, N. Ghani. **Categories of Containers**. FoSSaCS 2003.

Containers

- Shapes, positions per shape
- fmap: post-composition

challenges: no axioms, clean extraction

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*.

Containers in Rocq: A Bad Attempt

- Textbook encoding
- Real extraction output

challenges: no axioms, clean extraction

Containers in Rocq: A Bad Attempt

```
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)
```

OCaml cannot type `Pos C s`: `Obj.magic` everywhere. Safe only until foreign code calls it with a wrong value.

Containers in Rocq: A Bad Attempt

- Textbook encoding
- Real extraction output

challenge: no axioms

A Bad Attempt: 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
```

- `injection` gives an equality of dependent pairs (`existT`): `inj_pair2` needs **axiom K**.
- `fmap id = id`, as functions: **function extensionality**.

A Bad Attempt: Equality

- Real Print Assumptions output

Extractable Containers (I)

1. Decidable equality gives UIP (Hedberg): no axiom.
2. A predicate subtype `{x | P x}`, `P` in `Prop`, extracts to the supertype.

Extractable Containers (I)

- Validity proofs: equalities on bool
- Positions equal iff their values are

Extractable Containers (I)

1. Decidable equality gives UIP (Hedberg): no axiom.
2. A predicate subtype `{x | P x}`, `P` in `Prop`, extracts to the supertype.

```
Module DecBool := DecidableEqDep(BoolEq).                    (* Hedberg *)
Definition bool_irrelevance {b1 b2 : bool} (p1 p2 : b1 = b2) : p1 = p2 :=
  DecBool.UIP b1 b2 p1 p2.

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


Extractable Containers (I)

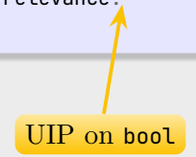
- Validity proofs: equalities on bool
- Positions equal iff their values are

Extractable Containers (I)

1. Decidable equality gives UIP (Hedberg): no axiom.
2. A predicate subtype `{x | P x}`, `P` in `Prop`, extracts to the supertype.

```
Module DecBool := DecidableEqDep(BoolEq).                    (* Hedberg *)
Definition bool_irrelevance {b1 b2 : bool} (p1 p2 : b1 = b2) : p1 = p2 :=
  DecBool.UIP b1 b2 p1 p2.

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



Extractable Containers (I)

- Validity proofs: equalities on bool
- Positions equal iff their values are

Extractable Containers (II)

solved: K, `Obj.magic`

```
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 }.
```

- One type `P` of **all** positions; `valid` says which are positions of a shape.
- Extraction: `cont` is a plain function `P -> X`.

Extractable Containers (II)

- A class: Sh, P fixed, no types as fields
- Valid: b = true, unique and erased

Extractable Containers (II)

solved: K, `Obj.magic`

```
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 }.
```

decidable validity

- One type `P` of **all** positions; `valid` says which are positions of a shape.
- Extraction: `cont` is a plain function `P -> X`.

Extractable Containers (II)

slide 49 / 129 step 2 of 3

- A class: Sh, P fixed, no types as fields
- Valid: b = true, unique and erased

Extractable Containers (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 }.
```

decidable validity

unique, erased

- One type `P` of **all** positions; `valid` says which are positions of a shape.
- Extraction: `cont` is a plain function `P -> X`.

Extractable Containers (II)

- A class: Sh, P fixed, no types as fields
- Valid: b = true, unique and erased

Example: the Quicksort Functor

```
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 l	false	false
Node a	true	true

`TreeF unit int`: $X \mapsto 1 + \texttt{int} \times X \times X$.

Example: the Quicksort Functor

- Leaf: no children; Node: two
- Polynomial functors: Sum, Prod, K, Id instances

solved: function extensionality

Setoids and Morphisms

```
#[mode = "!"]                                (* never search setoid ?A *)
Class setoid A := MkSetoid
  { eqRel : A -> A -> Prop;                  (* x =e y *)
    e_refl : ...; e_sym : ...; e_trans : ... }.

Structure morph `{setoid A} `{setoid B} := MkMorph       (* A ~> B *)
  { app :> A -> B;
    app_eq : forall x y, x =e y -> app x =e app y }.

Inductive AppR `{setoid X} (x y : App F X) : Prop :=     (* x =e y *)
| AppR_ext (Es : shape x =e shape y)
    (Ek : forall e1 e2, val e1 = val e2 -> cont x e1 =e cont y e2).
```

(simplified)

- `=e` on `A ~> B` is pointwise: no function extensionality. Laws are `=e` equations: generalised rewriting.
- Extraction erases `app_eq`: a morphism is its function.

Setoids and Morphisms

- Every type a setoid; =e default eq
- Layers equal: same shape, same children at equal positions
- Q: mode "!"? Search on setoid ?A diverged

Recursive Coalgebras (I)

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

V. Capretta, T. Uustalu, V. Vene. **Recursive Coalgebras from Comonads**. Inf. Comput. 2006.
J. Adámek, S. Milius, L. S. Moss. **On Well-Founded and Recursive Coalgebras**. FoSSaCS 2020.

Recursive Coalgebras (I)

- No solution, or every one
- Well-founded implies recursive

Recursive Coalgebras (I)

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

$$c = \mathsf{id},\ a = \mathsf{succ} \qquad h\, x = h\, x + 1 \qquad \textbf{no solution}$$

V. Capretta, T. Uustalu, V. Vene. **Recursive Coalgebras from Comonads**. Inf. Comput. 2006.
J. Adámek, S. Milius, L. S. Moss. **On Well-Founded and Recursive Coalgebras**. FoSSaCS 2020.

Recursive Coalgebras (I)

- No solution, or every one
- Well-founded implies recursive

Recursive Coalgebras (I)

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

$c = \mathsf{id},\ a = \mathsf{succ}$	$h\, x = h\, x + 1$	**no** solution
$c = a = \mathsf{id}$	$h\, x = h\, x$	**every** h is a solution

V. Capretta, T. Uustalu, V. Vene. **Recursive Coalgebras from Comonads**. Inf. Comput. 2006.
J. Adámek, S. Milius, L. S. Moss. **On Well-Founded and Recursive Coalgebras**. FoSSaCS 2020.

Recursive Coalgebras (I)

- No solution, or every one
- Well-founded implies recursive

Recursive Coalgebras (I)

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

$c = \mathsf{id},\ a = \mathsf{succ}$	$h\, x = h\, x + 1$	**no** solution
$c = a = \mathsf{id}$	$h\, x = h\, x$	**every** h is a solution

$c = \mathsf{id}$ never shrinks its input: $x \mapsto x \mapsto x \mapsto \cdots$

Recursive c: a unique h for every a. Sufficient: **well-founded** c, every input reaches the leaves.

V. Capretta, T. Uustalu, V. Vene. **Recursive Coalgebras from Comonads**. Inf. Comput. 2006.
J. Adámek, S. Milius, L. S. Moss. **On Well-Founded and Recursive Coalgebras**. FoSSaCS 2020.

Recursive Coalgebras (I)

- No solution, or every one
- Well-founded implies recursive

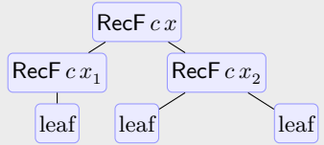
solved: termination

Recursive Coalgebras (II)

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

Definition RecP (c : Coalg F A) := forall x, RecF c x.

Structure RCoalg F A := Rec { coalg :> Coalg F A; recP : RecP coalg }.
```



- `RecF c x <-> Acc (child c) x`; in practice, a measure that `c` decreases (`mk_wf_coalg`).
- `RecP c <-> forall x, FinF (ana c x)`: only finite trees.

Recursive Coalgebras (II)

- Proof shaped like the call tree
- Induction hypothesis per position, as fmap exposes it

Recursive Hylomorphisms

```
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 as cx
          return (forall e : Pos F (shape cx), RecF c (cont cx e)) -> B
    with
    | MkCont sx cx =>
        fun H => a (MkCont sx (fun e => let ce := cx e in f ce (H e)))
    end (RecF_inv H).

Definition hylo : Alg F B ~> RCoalg F A ~> A ~> B := ...
  (* a c x |-> hylo_def a c x (terminating c x) *)
```


Recursive Hylomorphisms

slide 58 / 129 step 1 of 4

- Structural recursion on H, not x
- RecF_inv: Defined, so the guard checker sees through it
- Result independent of H (hylo_def_irr)

Recursive Hylomorphisms

termination proof in Prop: erased

```
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 as cx
          return (forall e : Pos F (shape cx), RecF c (cont cx e)) -> B
    with
    | MkCont sx cx =>
        fun H => a (MkCont sx (fun e => let ce := cx e in f ce (H e)))
    end (RecF_inv H).

Definition hylo : Alg F B ~> RCoalg F A ~> A ~> B := ...
  (* a c x |-> hylo_def a c x (terminating c x) *)
```


Recursive Hylomorphisms

slide 59 / 129 step 2 of 4

- Structural recursion on H, not x
- RecF_inv: Defined, so the guard checker sees through it
- Result independent of H (hylo_def_irr)

Recursive Hylomorphisms

```
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 as cx
          return (forall e : Pos F (shape cx), RecF c (cont cx e)) -> B
    with
    | MkCont sx cx =>
        fun H => a (MkCont sx (fun e => let ce := cx e in f ce (H e)))
    end (RecF_inv H).

Definition hylo : Alg F B ~> RCoalg F A ~> A ~> B := ...
  (* a c x |-> hylo_def a c x (terminating c x) *)
```

termination proof in Prop: erased

structurally smaller sub-proofs

Recursive Hylomorphisms

slide 60 / 129 step 3 of 4

- Structural recursion on H, not x
- RecF_inv: Defined, so the guard checker sees through it
- Result independent of H (hylo_def_irr)

Recursive Hylomorphisms

```
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 as cx
          return (forall e : Pos F (shape cx), RecF c (cont cx e)) -> B
    with
    | MkCont sx cx =>
        fun H => a (MkCont sx (fun e => let ce := cx e in f ce (H e)))
    end (RecF_inv H).

Definition hylo : Alg F B ~> RCoalg F A ~> A ~> B := ...
  (* a c x |-> hylo_def a c x (terminating c x) *)
```

termination proof in Prop: erased

convoy: H is about c x

structurally smaller sub-proofs

Recursive Hylomorphisms

slide 61 / 129 step 4 of 4

- Structural recursion on H, not x
- RecF_inv: Defined, so the guard checker sees through it
- Result independent of H (hylo_def_irr)

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

- hylo_uniq: the universal property
- Fusion: uniqueness, four steps (names as above)

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.
```

```
Lemma hylo_fusion_l (E : h \o a =e b \o fmap h)
  : h \o hylo a c =e hylo b c.
Proof.
  rewrite hylo_uniq, fmap_comp.
  arewrite <- E; arewrite <- (hylo_unr a c).
  reflexivity.
Qed.
```

The paper proof:

$$\begin{array}{ccc} F\,C & \xrightarrow{F\,[\![a,c]\!]} & F\,A \\ {\scriptstyle c}\uparrow & & \downarrow{\scriptstyle a} \\ C & \xrightarrow[[\![a,c]\!]]{} & A \end{array} \qquad \begin{array}{ccc} F\,h & \to & F\,B \\ \downarrow & & \downarrow \\ h & \to & B \end{array}$$

Uniqueness and the Laws

slide 63 / 129 step 2 of 3

- hylo_uniq: the universal property
- Fusion: uniqueness, four steps (names as above)

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.
```

```
Lemma hylo_fusion_l (E : h \o a =e b \o fmap h)
  : h \o hylo a c =e hylo b c.
Proof.
  rewrite hylo_uniq, fmap_comp.
  arewrite <- E; arewrite <- (hylo_unr a c).
  reflexivity.
Qed.
```

The paper proof:

$$\begin{array}{ccccc} F\,C & \xrightarrow{F\,[\![a,c]\!]} & F\,A & \xrightarrow{F\,h} & F\,B \\ {\scriptstyle c}\uparrow & & \downarrow{\scriptstyle a} & & \downarrow{\scriptstyle b} \\ C & \xrightarrow[[\![a,c]\!]]{} & A & \xrightarrow[h]{} & B \end{array}$$

Uniqueness and the Laws

slide 64 / 129 step 3 of 3

- hylo_uniq: the universal property
- Fusion: uniqueness, four steps (names as above)

Initial Algebras, from Recursive Coalgebras

```
Class Initial F T `{setoid T} :=
  { l_in  : App F T ~> T;
    l_out : RCoalg F T;                    (* a recursive coalgebra *)
    l_in_out : l_in \o l_out =e id;
    l_out_in : l_out \o l_in =e id }.

Definition cata (g : Alg F A) : T ~> A := hylo g l_out.
Definition rana (h : RCoalg F A) : A ~> T := hylo l_in h.

Lemma cata_univ : f =e cata g <-> f =e g \o fmap f \o l_out.
Corollary cata_ana_hylo : cata g \o rana h =e hylo g h.
```

(simplified)

- Initial: an algebra whose inverse is a **recursive** coalgebra. `cata_univ` is `hylo_uniq`.
- `LFix F` is one instance; any two are isomorphic. Later: **arrays**.

Initial Algebras, from Recursive Coalgebras

- cata, rana: hylos
- rana: no GFix to LFix conversion
- Instances: LFix, finite GFix, arrays

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
```

- Laws and calculations: closed. Machine integers: Rocq's primitive `int`, extracted to `Uint63`.

No Axioms

- Print Assumptions
- Q: why avoid consistent axioms? Future combinations

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
```

- Laws and calculations: closed. Machine integers: Rocq's primitive `int`, extracted to `Uint63`.
- **Why?** K and function extensionality are harmless on their own. But low-level targets may need axioms of their own, not yet known: an axiom-free base leaves room for them.

No Axioms

- Print Assumptions
- Q: why avoid consistent axioms? Future combinations

Generic for the Proofs, Clean for the Code

Definition	Why	Naive extraction	Final code
one type of positions, a `bool` check	no axiom K	no `Obj.magic`	plain types
`RecF`	induction per position	erased	erased
setoids	no function extensionality	dictionaries	inlined (`finished`)
one generic `hylo`	laws proved **once**, for every container and recursive coalgebra	a record per call	inlined (`finished`)
children as a function	one `fmap` for every container	a closure per call	CPS coalgebras

Generic for the Proofs, Clean for the Code

- Generic: the only way to prove laws once
- Cost: naive extraction only; gone in Part III

Generic for the Proofs, Clean for the Code

Definition	Why	Naive extraction	Final code
one type of positions, a `bool` check	no axiom K	no `Obj.magic`	plain types
`RecF`	induction per position	erased	erased
setoids	no function extensionality	dictionaries	inlined (`finished`)
one generic `hylo`	laws proved **once**, for every container and recursive coalgebra	a record per call	inlined (`finished`)
children as a function	one `fmap` for every container	a closure per call	CPS coalgebras

- Genericity is not traded for speed: it is how the laws are proved once.

Generic for the Proofs, Clean for the Code

- Generic: the only way to prove laws once
- Cost: naive extraction only; gone in Part III

Generic for the Proofs, Clean for the Code

Definition	Why	Naive extraction	Final code
one type of positions, a `bool` check	no axiom K	no `Obj.magic`	plain types
`RecF`	induction per position	erased	erased
setoids	no function extensionality	dictionaries	inlined (`finished`)
one generic `hylo`	laws proved **once**, for every container and recursive coalgebra	a record per call	inlined (`finished`)
children as a function	one `fmap` for every container	a closure per call	CPS coalgebras

- Genericity is not traded for speed: it is how the laws are proved once.
- Its run-time cost is removed **after** calculating, in Rocq, and proved correct (Part III).

Generic for the Proofs, Clean for the Code

- Generic: the only way to prove laws once
- Cost: naive extraction only; gone in Part III

Part III

Calculating in Rocq

Since ITP 2025

Calculating Quicksort with `Derive`

```
Derive qsort in (cata merge \o rana tsplit =e morph_of qsort) as qsort_calc.
Proof.
  rewrite cata_ana_hylo, <- (hylo_cps_hylo merge tsplit_k).
  finished.
Qed.
```

```
let rec qsort = function
| [] -> []
| i :: l ->
  let (l0, l1) = partition (fun x0 -> Uint63.lt x0 i) l in
  app (qsort l0) (i :: (qsort l1))
```

- `qsort` starts as an unknown; the rewrites build it; `finished` fills it in, reduced for extraction.
- `qsort` is transparent: it extracts. `qsort_calc` is the (opaque) proof.

Calculating Quicksort with `Derive`

- Spec: fold after unfold
- Extracted verbatim

Fusing a Map into Quicksort

The side condition of Part I, proved once:

```
Lemma Lmap_merge (f : int ~> int)
  : Lmap f \o merge =e merge \o natural (nt_shape id f) \o fmap (Lmap f).
```


Fusing a Map into Quicksort

slide 73 / 129 step 1 of 3

- merge2 of Part I: merge after natural (nt_shape id f)
- deforest: side condition l_out_in, proved on the spot
- One pass

Fusing a Map into Quicksort

The side condition of Part I, proved once:

```
Lemma Lmap_merge (f : int ~> int)
  : Lmap f \o merge =e merge \o natural (nt_shape id f) \o fmap (Lmap f).
```

```
Derive qsort_times_two
  in (Lmap times_two \o cata merge \o rana tsplit =e morph_of qsort_times_two)
  as qsort_times_two_calc.
Proof.
  rewrite map_merge_fusion, hylo_ana, deforest by apply l_out_in.
  rewrite <- (hylo_cps_hylo _ tsplit_k).
  finished.
Qed.
```


Fusing a Map into Quicksort

- merge2 of Part I: merge after natural (nt_shape id f)
- deforest: side condition l_out_in, proved on the spot
- One pass

Fusing a Map into Quicksort

The side condition of Part I, proved once:

```
Lemma Lmap_merge (f : int ~> int)
  : Lmap f \o merge =e merge \o natural (nt_shape id f) \o fmap (Lmap f).
```

```
let rec qsort_times_two = function
| [] -> []
| i :: l ->
  let (l0, l1) = partition (fun x0 -> Uint63.lt x0 i) l in
  app (qsort_times_two l0)
    ((Uint63.mul (Uint63.of_int (2)) i) :: (qsort_times_two l1))
```


Fusing a Map into Quicksort

- merge2 of Part I: merge after natural (nt_shape id f)
- deforest: side condition l_out_in, proved on the spot
- One pass

Quicksort, Step by Step

```
(Lmap times_two \o cata merge) \o rana tsplit
```


Quicksort, Step by Step

slide 76 / 129 step 1 of 4

- One law per step

Quicksort, Step by Step

```
  (Lmap times_two \o cata merge) \o rana tsplit
=e  (* map_merge_fusion: double inside merge *)
  hylo (merge \o natural (nt_shape id times_two)) l_out \o rana tsplit
```


Quicksort, Step by Step

- One law per step

Quicksort, Step by Step

```
  (Lmap times_two \o cata merge) \o rana tsplit
=e  (* map_merge_fusion: double inside merge *)
  hylo (merge \o natural (nt_shape id times_two)) l_out \o rana tsplit
=e  (* hylo_ana, deforest: unfold then fold, with no tree in between *)
  hylo (merge \o natural (nt_shape id times_two)) tsplit
```


Quicksort, Step by Step

- One law per step

Quicksort, Step by Step

```
  (Lmap times_two \o cata merge) \o rana tsplit
=e  (* map_merge_fusion: double inside merge *)
  hylo (merge \o natural (nt_shape id times_two)) l_out \o rana tsplit
=e  (* hylo_ana, deforest: unfold then fold, with no tree in between *)
  hylo (merge \o natural (nt_shape id times_two)) tsplit
=e  (* hylo_cps_hylo: continuation-passing, for clean extraction *)
  hylo_cps (merge \o natural (nt_shape id times_two)) tsplit_k
```


Quicksort, Step by Step

- One law per step

A False Side Condition, Caught

Specification: `cata merge \o rana tsplit \o Lmap negate`. Fusion moves the map into the coalgebra:

```
    hylo merge tsplit \o Lmap negate =e hylo merge h2
if  tsplit \o Lmap negate =e fmap (Lmap negate) \o h2
```

Choose `h2 := tsplit` (wrong), and Rocq leaves:

```
goal 1:  hylo merge tsplit =e morphism qsort_negate
goal 2:  tsplit \o Lmap negate =e fmap (Lmap negate) \o tsplit
```


A False Side Condition, Caught

- The PyTorch failure mode
- Caught: unprovable goal

A False Side Condition, Caught

Specification: `cata merge \o rana tsplit \o Lmap negate`. Fusion moves the map into the coalgebra:

```
    hylo merge tsplit \o Lmap negate =e hylo merge h2
if  tsplit \o Lmap negate =e fmap (Lmap negate) \o h2
```

Choose `h2 := tsplit` (wrong), and Rocq leaves:

```
goal 1:  hylo merge tsplit =e morphism qsort_negate
goal 2:  tsplit \o Lmap negate =e fmap (Lmap negate) \o tsplit
```

- **Goal 1**: plain quicksort; the negation **silently vanishes**.

A False Side Condition, Caught

- The PyTorch failure mode
- Caught: unprovable goal

A False Side Condition, Caught

Specification: `cata merge \o rana tsplit \o Lmap negate`. Fusion moves the map into the coalgebra:

```
    hylo merge tsplit \o Lmap negate =e hylo merge h2
if  tsplit \o Lmap negate =e fmap (Lmap negate) \o h2
```

Choose `h2 := tsplit` (wrong), and Rocq leaves:

```
goal 1:  hylo merge tsplit =e morphism qsort_negate
goal 2:  tsplit \o Lmap negate =e fmap (Lmap negate) \o tsplit
```

- **Goal 1**: plain quicksort; the negation **silently vanishes**.
- **Goal 2**, the side condition, is **false**: on `[1, 2]`, pivot `-1` with children `[-2]`, `[]`; against pivot `1` with `[]`, `[-2]`.

A False Side Condition, Caught

- The PyTorch failure mode
- Caught: unprovable goal

A False Side Condition, Caught

Specification: `cata merge \o rana tsplit \o Lmap negate`. Fusion moves the map into the coalgebra:

```
    hylo merge tsplit \o Lmap negate =e hylo merge h2
if  tsplit \o Lmap negate =e fmap (Lmap negate) \o h2
```

Choose `h2 := tsplit` (wrong), and Rocq leaves:

```
goal 1:  hylo merge tsplit =e morphism qsort_negate
goal 2:  tsplit \o Lmap negate =e fmap (Lmap negate) \o tsplit
```

- **Goal 1**: plain quicksort; the negation **silently vanishes**.
- **Goal 2**, the side condition, is **false**: on `[1, 2]`, pivot `-1` with children `[-2]`, `[]`; against pivot `1` with `[]`, `[-2]`.
- The calculation cannot be finished. (Right `h2`: negate the pivot, flip the comparison.)

A False Side Condition, Caught

- The PyTorch failure mode
- Caught: unprovable goal

challenge: clean extraction

Generic Code Is Not Clean Code

Extracting the unreduced `hylo merge tsplit`:

```
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_spec =
  hylo def_eq treeF (list_eq def_eq) (list_eq def_eq) merge tsplit
```


Generic Code Is Not Clean Code

slide 84 / 129 step 1 of 4

- Type-safe, but generic
- Setoid and container dictionaries
- Inline before extracting

challenge: clean extraction

Generic Code Is Not Clean Code

Extracting the unreduced `hylo merge tsplit`:

```
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_spec =
  hylo def_eq treeF (list_eq def_eq) (list_eq def_eq) merge tsplit
```

dictionaries at run time

Generic Code Is Not Clean Code

- Type-safe, but generic
- Setoid and container dictionaries
- Inline before extracting

challenge: clean extraction

Generic Code Is Not Clean Code

Extracting the unreduced `hylo merge tsplit`:

```
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_spec =
  hylo def_eq treeF (list_eq def_eq) (list_eq def_eq) merge tsplit
```

dictionaries at run time

a record per call

Generic Code Is Not Clean Code

- Type-safe, but generic
- Setoid and container dictionaries
- Inline before extracting

challenge: clean extraction

Generic Code Is Not Clean Code

Extracting the unreduced `hylo merge tsplit`:

```
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_spec =
  hylo def_eq treeF (list_eq def_eq) (list_eq def_eq) merge tsplit
```

dictionaries at run time

a record per call

Fix: `finished` reduces the derived program, unfolding the `unfold_code` hints (and `Extraction Inline`).

Generic Code Is Not Clean Code

- Type-safe, but generic
- Setoid and container dictionaries
- Inline before extracting

Closures in the Extracted Code

```
let rec qsort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x0 -> ltb 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))
```


Closures in the Extracted Code

slide 88 / 129 step 1 of 4

- Real ITP output
- g: the children, by position
- x0: used twice, so not inlined

Closures in the Extracted Code

```
let rec qsort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x0 -> ltb 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))
```

ITP 2025's code, verbatim. The generic `hylo`, with c inlined, is a case of a case:

$$[\![a, c]\!]\, x \;=\; \textbf{case}\ \underbrace{\big(\textbf{case}\ x\ \textbf{of}\ [\,] \Rightarrow (\mathsf{Leaf}, !) \mid p : t \Rightarrow (\mathsf{Node}\, p,\ g_{l,r})\big)}_{c\,x}\ \textbf{of}\ (s, g) \Rightarrow a\,(s,\ [\![a, c]\!] \circ g)$$

Closures in the Extracted Code

- Real ITP output
- g: the children, by position
- x0: used twice, so not inlined

Closures in the Extracted Code

```
let rec qsort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x0 -> ltb 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))
```

ITP 2025's code, verbatim. The generic `hylo`, with c inlined, is a case of a case:

$$[\![a, c]\!]\, x \;=\; \textbf{case}\; \underbrace{\big(\textbf{case}\; x\; \textbf{of}\; [\,] \Rightarrow (\mathsf{Leaf}, !) \mid p : t \Rightarrow (\mathsf{Node}\, p,\, g_{l,r})\big)}_{c\, x}\; \textbf{of}\; (s, g) \Rightarrow a\,(s,\, [\![a, c]\!] \circ g)$$

- Stuck in Rocq for a variable x: a never meets a known layer.

Closures in the Extracted Code

- Real ITP output
- g: the children, by position
- x0: used twice, so not inlined

challenge: clean extraction

Closures in the Extracted Code

```
let rec qsort_times_two = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x0 -> ltb 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))
```

ITP 2025's code, verbatim. The generic `hylo`, with c inlined, is a case of a case:

$$[\![a, c]\!]\, x \;=\; \textbf{case}\ \underbrace{\big(\textbf{case}\ x\ \textbf{of}\ [\,] \Rightarrow (\mathsf{Leaf}, !) \mid p : t \Rightarrow (\mathsf{Node}\, p,\ g_{l,r})\big)}_{c\, x}\ \textbf{of}\ (s, g) \Rightarrow a\,(s,\ [\![a, c]\!] \circ g)$$

- Stuck in Rocq for a variable x: a never meets a known layer.
- What comes out: the cases commuted, but a applies $[\![a, c]\!] \circ g_{l,r}$ twice, so it is let-bound: the closure `x0`.

Closures in the Extracted Code

- Real ITP output
- g: the children, by position
- x0: used twice, so not inlined

The CPS Transform of a Coalgebra

Definition. For $c\,x = t$, its *CPS transform*

$$\overline{c} : \forall R.\ \Pi_{x:A}\left(\Pi_{y:FA}\ y = c\,x \to R\right) \to R \qquad \overline{c}_R\,x\,k = \langle t\rangle_k$$

is defined on the syntax of t:

$$\begin{aligned} \langle \mathbf{case}\ e\ \mathbf{of}\ \{p_i \Rightarrow t_i\}\rangle_k &= \mathbf{case}\ e\ \mathbf{of}\ \{p_i \Rightarrow \langle t_i\rangle_k\} && (\text{branch } i \text{ knows } e = p_i) \\ \langle L\rangle_k &= k\,L\,E_L && (L \text{ a layer};\ E_L : L = c\,x) \end{aligned}$$

The CPS Transform of a Coalgebra

- c, with the continuation pushed into every branch
- E: the layer is c x, from the branch equations

The CPS Transform of a Coalgebra

Definition. For $c\,x = t$, its *CPS transform*

$$\overline{c} : \forall R.\ \Pi_{x:A}\left(\Pi_{y:FA}\ y = c\,x \to R\right) \to R \qquad \overline{c}_R\,x\,k = \langle t\rangle_k$$

is defined on the syntax of t:

$$\begin{aligned} \langle \textbf{case}\ e\ \textbf{of}\ \{p_i \Rightarrow t_i\}\rangle_k &= \textbf{case}\ e\ \textbf{of}\ \{p_i \Rightarrow \langle t_i\rangle_k\} && (\text{branch } i \text{ knows } e = p_i)\\ \langle L\rangle_k &= k\,L\,E_L && (L \text{ a layer};\ E_L : L = c\,x) \end{aligned}$$

Lemma (CPS law). $\overline{c}_R\,x\,k = k\,(c\,x)$ refl.

The CPS Transform of a Coalgebra

- c, with the continuation pushed into every branch
- E: the layer is c x, from the branch equations

The CPS Transform of a Coalgebra

Definition. For $c\,x = t$, its *CPS transform*

$$\overline{c} : \forall R.\ \Pi_{x:A}\left(\Pi_{y:FA}\ y = c\,x \to R\right) \to R \qquad \overline{c}_R\,x\,k = \langle t\rangle_k$$

is defined on the syntax of t:

$$\begin{aligned} \langle \textbf{case}\ e\ \textbf{of}\ \{p_i \Rightarrow t_i\}\rangle_k &= \textbf{case}\ e\ \textbf{of}\ \{p_i \Rightarrow \langle t_i\rangle_k\} && (\text{branch } i \text{ knows } e = p_i) \\ \langle L\rangle_k &= k\,L\,E_L && (L \text{ a layer};\ E_L : L = c\,x) \end{aligned}$$

Lemma (CPS law). $\overline{c}_R\,x\,k = k\,(c\,x)$ refl.

- Built by a tactic, `cps_from_coalg` (each case becomes a `convoy`); the law by `cps_law`.

The CPS Transform of a Coalgebra

- c, with the continuation pushed into every branch
- E: the layer is c x, from the branch equations

CPS Hylomorphisms

Definition. By structural recursion on $H : \mathsf{RecF}\, c\, x$:

$$[\![a, c]\!]^{\mathsf{cps}}\, x\, H = \overline{c}_B\, x\, \Big(\lambda\, (s, g)\, E.\ a\, (s,\ \lambda e.\ [\![a, c]\!]^{\mathsf{cps}}\, (g\, e)\, (H_E\, e))\Big)$$

where $H_E\, e : \mathsf{RecF}\, c\, (g\, e)$ is a sub-proof of H, transported along $E : (s, g) = c\, x$.

CPS Hylomorphisms

slide 95 / 129 step 1 of 2

- E: children of y are children of c x
- Rocq: hylo_cps a tsplit_k; proof by induction on RecF

CPS Hylomorphisms

Definition. By structural recursion on $H : \mathsf{RecF}\, c\, x$:

$$[\![a, c]\!]^{\mathsf{cps}}\, x\, H = \overline{c}_B\, x \left(\lambda\, (s, g)\, E.\ a\, (s,\ \lambda e.\ [\![a, c]\!]^{\mathsf{cps}}\, (g\, e)\, (H_E\, e))\right)$$

where $H_E\, e : \mathsf{RecF}\, c\, (g\, e)$ is a sub-proof of H, transported along $E : (s, g) = c\, x$.

Theorem. $[\![a, c]\!]^{\mathsf{cps}} = [\![a, c]\!]$.

Proof, with $h = [\![a, c]\!]^{\mathsf{cps}}$ (proofs H elided):

$$\begin{array}{rcll} h\, x & = & \overline{c}_B\, x \left(\lambda\, (s, g)\, E.\ a\, (s,\ h \circ g)\right) & \{\text{definition of } h\} \\ & = & a\, (F\, h\, (c\, x)) & \{\text{CPS law}\} \end{array}$$

So $h = a \circ F\, h \circ c$, whose unique solution is $[\![a, c]\!]$ (`hylo_uniq`).

CPS Hylomorphisms

- E: children of y are children of c x
- Rocq: hylo_cps a tsplit_k; proof by induction on RecF

CPS Is Case-of-Case

For $c\,x = \mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow L_i\}$:

$$\overline{c}_R\,x\,k = \mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow k\,L_i\,E_i\},$$

so

$$\begin{aligned} [\![a, c]\!]\,x &= \mathbf{case}\ (\mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow L_i\})\ \mathbf{of}\ (s, g) \Rightarrow a\,(s,\ [\![a, c]\!] \circ g) \\ [\![a, c]\!]^{\mathsf{cps}}\,x &= \mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow a\,(F\,[\![a, c]\!]^{\mathsf{cps}}\,L_i)\} \end{aligned}$$

CPS Is Case-of-Case

- Commuting conversion, done once, proved
- Reduced in Rocq (finished)

CPS Is Case-of-Case

For $c\,x = \mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow L_i\}$:

$$\overline{c}_R\,x\,k = \mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow k\,L_i\,E_i\},$$

so

$$\begin{aligned} [\![a, c]\!]\,x &= \mathbf{case}\ (\mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow L_i\})\ \mathbf{of}\ (s, g) \Rightarrow a\,(s,\ [\![a, c]\!] \circ g) \\ [\![a, c]\!]^{\mathsf{cps}}\,x &= \mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow a\,(F\,[\![a, c]\!]^{\mathsf{cps}}\,L_i)\} \end{aligned}$$

- The commuting conversion, done **once**, in $\overline{c}$; the theorem proves it correct.

CPS Is Case-of-Case

- Commuting conversion, done once, proved
- Reduced in Rocq (finished)

CPS Is Case-of-Case

For $c\,x = \mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow L_i\}$:

$$\overline{c}_R\,x\,k = \mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow k\,L_i\,E_i\},$$

so

$$\begin{aligned} [\![a, c]\!]\,x &= \mathbf{case}\ (\mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow L_i\})\ \mathbf{of}\ (s, g) \Rightarrow a\,(s,\ [\![a, c]\!] \circ g) \\ [\![a, c]\!]^{\mathsf{cps}}\,x &= \mathbf{case}\ x\ \mathbf{of}\ \{p_i \Rightarrow a\,(F\,[\![a, c]\!]^{\mathsf{cps}}\,L_i)\} \end{aligned}$$

- The commuting conversion, done **once**, in $\overline{c}$; the theorem proves it correct.
- Each L_i is a known layer, so Rocq itself reduces $a\,(F\,h\,L_i)$, before extraction.

CPS Is Case-of-Case

- Commuting conversion, done once, proved
- Reduced in Rocq (finished)

solved: clean extraction

Quicksort in CPS

The CPS transform of quicksort's `split`:

$$\overline{\mathsf{split}}_R\, x\, k = \mathbf{case}\ x\ \mathbf{of}\ \begin{cases} [\,] \Rightarrow k\,(\mathsf{Leaf}, !)\ E_1 \\ p : t \Rightarrow \mathbf{let}\ (l, r) = \mathsf{partition}\,(<p)\,t\ \mathbf{in}\ k\,(\mathsf{Node}\,p,\ g_{l,r})\ E_2 \end{cases}$$

Quicksort in CPS

- let (l, r) is a case too
- merge's case fires; positions known

solved: clean extraction

Quicksort in CPS

The CPS transform of quicksort's `split`:

$$\overline{\mathsf{split}}_R\, x\, k = \mathbf{case}\ x\ \mathbf{of}\ \begin{cases} [\,] \Rightarrow k\,(\mathsf{Leaf}, !)\ E_1 \\ p : t \Rightarrow \mathbf{let}\ (l, r) = \mathsf{partition}\,(<p)\,t\ \mathbf{in}\ k\,(\mathsf{Node}\,p,\ g_{l,r})\ E_2 \end{cases}$$

With $h = [\![\mathsf{merge}, \mathsf{split}]\!]^{\mathsf{cps}}$, Rocq reduces:

$$\begin{aligned} h\,(p : t) &= \mathbf{let}\ (l, r) = \mathsf{partition}\,(<p)\,t\ \mathbf{in}\ \mathsf{merge}\,(\mathsf{Node}\,p,\ h \circ g_{l,r}) \\ &= \mathbf{let}\ (l, r) = \mathsf{partition}\,(<p)\,t\ \mathbf{in}\ h\,l +\!\!+ p : h\,r \end{aligned}$$

No closure left for extraction.

Quicksort in CPS

- let (l, r) is a case too
- merge's case fires; positions known

Since ITP 2025: Usability

- Properness and termination: **instance search** (`Proper`, `Terminates`); a named function is a morphism, or a recursive coalgebra, by canonical structures.

Since ITP 2025: Usability

- Before: proofs by hand for every function
- arewrite: law anywhere in a chain
- Mode: search ate all memory

Since ITP 2025: Usability

- Properness and termination: **instance search** (`Proper`, `Terminates`); a named function is a morphism, or a recursive coalgebra, by canonical structures.
- Curried functions as morphisms: an **Elpi coercion** (`B -> I -> A -> B` to `B ~> I ~> A ~> B`).

Since ITP 2025: Usability

- Before: proofs by hand for every function
- arewrite: law anywhere in a chain
- Mode: search ate all memory

Since ITP 2025: Usability

- Properness and termination: **instance search** (`Proper`, `Terminates`); a named function is a morphism, or a recursive coalgebra, by canonical structures.
- Curried functions as morphisms: an **Elpi coercion** (`B -> I -> A -> B` to `B ~> I ~> A ~> B`).
- Rewriting modulo associativity of `\o`: `arewrite`, steps proved by conversion.

Since ITP 2025: Usability

- Before: proofs by hand for every function
- arewrite: law anywhere in a chain
- Mode: search ate all memory

Since ITP 2025: Usability

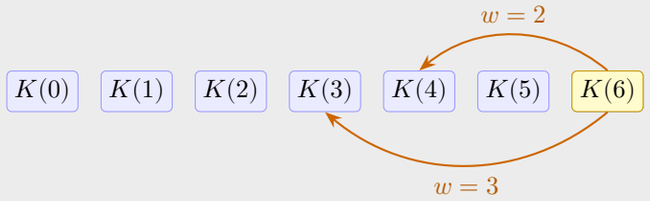
- Properness and termination: **instance search** (`Proper`, `Terminates`); a named function is a morphism, or a recursive coalgebra, by canonical structures.
- Curried functions as morphisms: an **Elpi coercion** (`B -> I -> A -> B` to `B ~> I ~> A ~> B`).
- Rewriting modulo associativity of `\o`: `arewrite`, steps proved by conversion.
- Diverging instance search on `setoid ?A`: `#[mode = "!"]`.

Since ITP 2025: Usability

- Before: proofs by hand for every function
- arewrite: law anywhere in a chain
- Mode: search ate all memory

Dynamic Programming: Knapsack

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



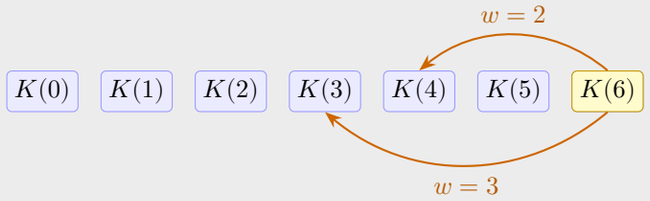
Items (w, v): $(2, 3)$ and $(3, 4)$, so $K(6) = \max(K(4) + 3, K(3) + 4) = 9$.

Dynamic Programming: Knapsack

- Recursion over items: recomputes
- Goal: over capacities, with a table

Dynamic Programming: Knapsack

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



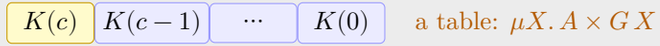
Items (w, v): $(2, 3)$ and $(3, 4)$, so $K(6) = \max(K(4) + 3, K(3) + 4) = 9$.

The specification: a hylomorphism over the items that fit, `hylo naive_alg (naive_c wvs)`. Exponential: it **recomputes**.

Dynamic Programming: Knapsack

- Recursion over items: recomputes
- Goal: over capacities, with a table

Dynamorphisms in Rocq




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

Definition dyna `{!Initial (Memo G A) T}
  (a : App G T ~> A) (c : RCoalg G B) : B ~> A
  := mhead \o l_out \o hylo (l_in \o mcons \o pair a id) c.

Theorem dyna_iso : dyna a c =e dyna (a \o fmap (iso U T)) c.  (* any table *)

Instance ATab_initial : Initial (Memo NatF A) (ATab A).    (* arrays *)
```

- The table is **any** initial algebra of `Memo G A`: proofs use `LFix`, code uses arrays.

J. Kabanov, V. Vene. **Recursion Schemes for Dynamic Programming**. MPC 2006.
R. Hinze, N. Wu, J. Gibbons. **Conjugate Hylomorphisms, or: The Mother of All Structured Recursion Schemes**. POPL 2015.

Dynamorphisms in Rocq

- dyna: hylo into the table, take the head
- ATab: persistent arrays, proved initial (implicit arguments dropped)

Changing the Recursion

```
Theorem reenumerate (an : App F A ~> A) (cn : RCoalg F B)
  (e : RCoalg NatF B) (alg : (int -> option A) ~> A)
  (Hlook : forall (g : B ~> A) b,
      an (fmap g (cn b)) =e alg (look_rec e g b))
  : hylo an cn =e dyna (alg \o look_with) e.
```

- `an`, `cn`: the naive recursion (over items).
- `e`: the recursion we want (count down capacities).
- `Hlook`: `an` only needs results that `e` reaches.
- `look_with`: lookups jump into the array table, $O(1)$.

Changing the Recursion

- Hylo equals dyna on another coalgebra
- Proof: uniqueness of hylo (implicit arguments dropped)

Knapsack, Step by Step

```
knapsack_naive wvs
```


Knapsack, Step by Step

- reenumerate: O(1) lookups
- dyna_loop: bottom-up loop

Knapsack, Step by Step

```
  knapsack_naive wvs
=e  (* unfold: the exponential specification *)
  hylo naive_alg (naive_c wvs)
```


Knapsack, Step by Step

- reenumerate: O(1) lookups
- dyna_loop: bottom-up loop

Knapsack, Step by Step

```
  knapsack_naive wvs
=e  (* unfold: the exponential specification *)
  hylo naive_alg (naive_c wvs)
=e  (* reenumerate: recurse on capacities, look results up in O(1) *)
  dyna (knap_alg wvs \o look_with (w:=fast_drop) fast_drop_walk) out_int_c
```


Knapsack, Step by Step

- reenumerate: O(1) lookups
- dyna_loop: bottom-up loop

Knapsack, Step by Step

```
  knapsack_naive wvs
=e  (* unfold: the exponential specification *)
  hylo naive_alg (naive_c wvs)
=e  (* reenumerate: recurse on capacities, look results up in O(1) *)
  dyna (knap_alg wvs \o look_with (w:=fast_drop) fast_drop_walk) out_int_c
=e  (* dyna_loop: the table of a count-down is built by a loop *)
  (mhead \o l_out) \o
    (morphism fun n : int =>
       loop n (fun _ : int => memo_step (knap_step wvs))
         (0%uint63, memo_base 0%uint63))
```


Knapsack, Step by Step

slide 113 / 129 step 4 of 4

- reenumerate: O(1) lookups
- dyna_loop: bottom-up loop

Knapsack: What Comes Out

```
let knap_step wvs x =
  imax (Uint63.of_int (0))
    (let rec f = function
     | [] -> min_int
     | h :: t ->
       imax
         (match fast_drop (fst h) x with
          | Some a -> Uint63.add (head a) (snd h)
          | None -> min_int)
         (f t)
     in f wvs)

let knapsack_arr wvs x =
  head
    (let rec f = function
     | (i, s) ->
       if Uint63.lt i x
       then f ((Uint63.add i (Uint63.of_int (1))), (push (knap_step wvs s) s))
       else s
     in f ((Uint63.of_int (0)), (single (Uint63.of_int (0)))))
```


Knapsack: What Comes Out

slide 114 / 129 step 1 of 2

- Real extraction output
- Loop counting up; push one entry
- fast_drop: O(1) lookup

Knapsack: What Comes Out

```
let rec jump n t =
  let { blk = b; top = i; prev = p } = t in
  if Uint63.le n i
  then Some { blk = b; top = (Uint63.sub i n); prev = p }
  else (match p with
        | Some q -> jump (Uint63.sub (Uint63.sub n i) (Uint63.of_int (1))) q
        | None -> None)

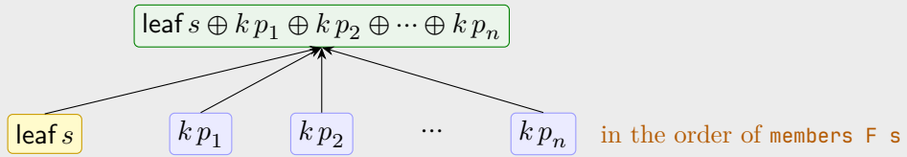
let fast_drop j t =
  if Uint63.equal j (Uint63.of_int (0))
  then None
  else jump (Uint63.sub j (Uint63.of_int (1))) t
```

`fast_drop`: jump straight into the right block of the array, $O(1)$. Knapsack is linear in the capacity.

Knapsack: What Comes Out

- Real extraction output
- Loop counting up; push one entry
- fast_drop: O(1) lookup

Monoid Algebras




```
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))).
```

```
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.
```

List homomorphisms, for **any** finite container and **any** recursive coalgebra.

Monoid Algebras

- Finite containers: members lists the positions
- One fusion law: Bird's promotions, Horner's rule

Maximum Segment Sum

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

Derive mss_linear in (mss =e morph_of mss_linear) as mss_calc.
Proof.
  unfold mss, segs.
  arewrite (map_promotion sum).
  arewrite (reduce_promotion max_monoid).
  arewrite (lmap_comp maximum (lmap sum)).
  arewrite (lmap_comp (maximum \o lmap sum) inits).
  rewrite horner_mss.                          (* max-plus semiring *)
  arewrite (scan_lemma odot (Some 0)).
  arewrite (scan_tupling max_monoid odot (Some 0)).
  arewrite (foldr_lmap (tup_step max_monoid odot) some_m (Some 0, Some 0)).
  finished.
Qed.
```


Maximum Segment Sum

- Each step a library law
- Horner: max-plus semiring
- Scan, tupling: one pass

Maximum Segment Sum

```
let mss_linear x =
  let (y, _) =
    let rec f = function
    | [] -> ((Some Bigint.zero), (Some Bigint.zero))
    | h :: t ->
      let p = f t in
      let z0 = omax (Some Bigint.zero) (oplus (Some h) (snd p)) in
      ((omax z0 (fst p)), z0)
    in f x
  in
  y
```

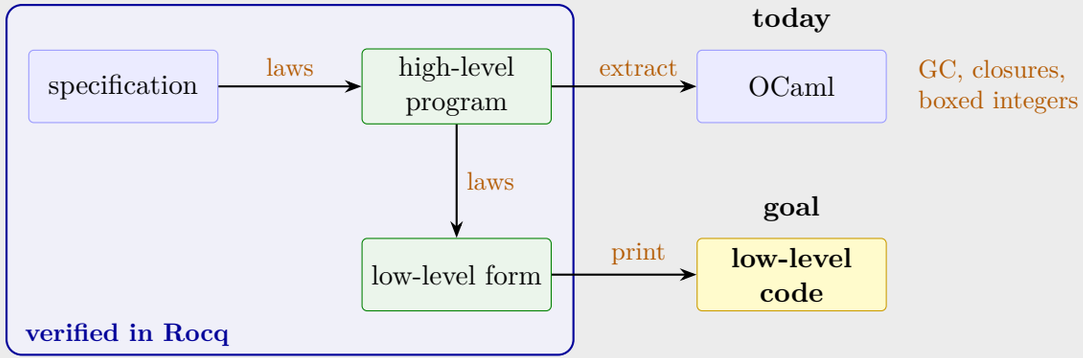
Bird's calculation, from cubic to linear, machine-checked.

Maximum Segment Sum

- Each step a library law
- Horner: max-plus semiring
- Scan, tupling: one pass

ongoing

Ongoing: Towards Low-Level Calculation



Laws that change how data is **represented** and how work is **scheduled**, every step checked in Rocq.

Ongoing: Towards Low-Level Calculation

- One slide only
- Q: details? Early days

Summary

- **Program calculation**: hylomorphism laws turn specifications into efficient, parallel programs; until now, laws and side conditions were trusted.
- **Hylomorphisms in Rocq**: extractable containers, setoids, recursive coalgebras; the laws proved once, axiom-free.
- **Calculating in Rocq**: `Derive`, every side condition checked; clean OCaml, down to native arrays; dynamorphisms, monoid algebras.

Program optimisation as a verified calculation, not an untrusted compiler transformation.

D. Castro-Perez, M. Paviotti, M. Vollmer. **Program Optimisations via Hylomorphisms for Extraction of Executable Code**. ITP 2025.
Library: https://github.com/dcastrop/rocq-hylomorphisms

Summary

- One line per part
- Back to PyTorch

Thank you!

Questions?

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

Part II: joint work with Marco Paviotti
and Michael Vollmer (ITP 2025).

- Questions

Backup

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	$x = a \cdot (id \vartriangle \mathsf{D}\, (\mathsf{D}_\infty\, x \cdot \llparenthesis c \rrparenthesis) \cdot c) \;:\; A \leftarrow C$	$a : C \times \mathsf{D}\, (\mathsf{D}_\infty\, A) \to A$
finite memo-table (§5.6)	$\mathsf{U}_* \dashv \mathsf{Cofree}_*$	ccf	$x = a \cdot (id \vartriangle \mathsf{D}\, (\mathsf{D}_*\, x \cdot \llparenthesis c \rrparenthesis_*) \cdot c) \;:\; A \leftarrow C$	$a : C \times \mathsf{D}\, (\mathsf{D}_*\, A) \to A$

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.

Conjugate Hylomorphisms

- Accumulators: curry/uncurry
- Dynamic programming: cofree comonad

Proving Hylomorphisms Correct

```
Lemma hylo_ind (P : A -> B -> Prop) (a : Alg F B) (c : RCoalg F A)
  : (forall x, (forall e, P (cont (c x) e) (hylo a c (cont (c x) e))) ->
       P x (hylo a c x)) ->
    forall x, P x (hylo a c x).
Proof. intros H x; induction (recP c x) as [x _ Ih]; apply H, Ih. Qed.

Lemma hylo_correct : forall l : list int,
  Sorted leb (hylo merge tsplit l) /\ Permutation l (hylo merge tsplit l).
Proof.
  intros l; induction (recP tsplit l) as [x _ Ih]; rewrite hylo_unr.
  ...
```

(simplified)

- Induction on the termination proof; unroll `hylo` once (`hylo_unr`).
- Proved once for the specification, carried along every calculation step.

Proving Hylomorphisms Correct

- QSort.v: sorted and permutation

Terminal Coalgebras and Finite Trees

```
CoInductive GFix F := GFix_in { GFix_out : App F (GFix F) }.
Definition GFixR : GFix F -> GFix F -> Prop := paco2 GFixR_ bot2.

Lemma gfix_ana_univ : f =e gfix_ana h <-> f =e gfix_in \o fmap f \o h.

Definition FinF : GFix F -> Prop := RecF g_out.          (* finite *)
Corollary ana_rec_term : (forall x, FinF (ana h x)) <-> RecP h.

Lemma rana_ana : ana (T := GFix F) f =e ana l_out \o rana f.
```

(simplified)

- `GFix`: bisimilarity with Paco; anamorphisms into it, with their universal property.
- A coalgebra is recursive iff its anamorphism only builds finite trees.
- `rana`: straight into the initial algebra, no conversion.

Terminal Coalgebras and Finite Trees

- Terminal class, as Initial
- rana_ana: rana and ana agree

CPS Coalgebras in Rocq

```
Record CCoalg (c : RCoalg F A) := MkCCoalg
  { ccoalg : forall R (x : A), (forall y, y = c x -> R) -> R;
    ccoalg_law : forall R x k, ccoalg R x k = k (c x) eq_refl }.

Definition hylo_cps_def (a : Alg F B) (cc : CCoalg c)
  : forall x, RecF c x -> B :=
  fix f x H := ccoalg cc B x (fun y E =>
    a (MkCont (shape y) (fun e => f (cont y e) (RecF_inv_eq E H e)))).

Theorem hylo_cps_hylo : hylo_cps a cc =e hylo a c.
```

(simplified)

- `hylo_cps` takes any `CCoalg c`: the CPS law is a field. The slides' $\overline{c}$ is `cps_from_coalg`'s.
- `RecF_inv_eq`: a match on `E`, so the recursion stays structural on `H`.

CPS Coalgebras in Rocq

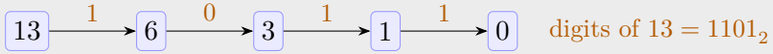
- HyloCPS.v
- tsplit_k : CCoalg tsplit, by cps_from_coalg

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).

Definition power (x : int) : int ~> int :=
  cata (pow_alg x) \o rana (digits 2 eq_refl).

Derive fast_power in (power x =e morph_of fast_power) as fast_power_calc.
```



digits of $13 = 1101_2$

```
let rec fast_power x x0 =
  if Uint63.equal x0 (Uint63.of_int (0))
  then (Uint63.of_int (1))
  else Uint63.mul (digit_pow x (Uint63.rem x0 (Uint63.of_int (2))))
         (sq (fast_power x (Uint63.div x0 (Uint63.of_int (2)))))
```


Coalgebras Need Not Follow the Data

- Power: fold over the binary digits
- Deforested: fast exponentiation

Knapsack: the `Derive`

```
Derive knapsack_arr in (knapsack_naive wvs =e morph_of knapsack_arr)
  as knapsack_arr_calc.
Proof.
  unfold knapsack_naive.
  (* Memoise: the subproblems are read from a table *)
  erewrite reenumerate.
  (* The table of a count-down is built by a loop *)
  erewrite dyna_loop.
  (* The loop's step, its first entry, and the algebra on the table *)
  2: exact knap_step_calc.
  2: exact knap_base.
  2: exact knap_look.
  finished.
Qed.
```


Knapsack: the `Derive`

- Side goals: the loop's step, its first entry, the lookup algebra

What Changed Since the Paper

- Rocq 9; properness and termination by instance search; an Elpi coercion for curried morphisms.
- `Derive`, `arewrite` and `finished` for calculations; CPS coalgebras for extraction.
- `Initial` and `Terminal` classes; `RecF` iff `well_founded (child c)`; `GFix` with Paco.
- Finite containers and monoid algebras; dynamorphisms over arrays.
- New examples: MSS, fast power, knapsack.

What Changed Since the Paper

- Library: github.com/dcastrop/rocq-hylomorphisms