

Mechanising Recursion Schemes with Magic-Free Coq Extraction

David Castro-Perez, Marco Paviotti, and Michael Vollmer

d.castro-perez@kent.ac.uk

02-05-2024



Background

Hylomorphisms

Fold over Lists

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

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

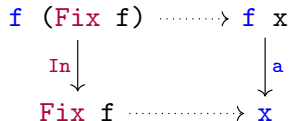
There are many different kinds of Recursion Schemes (e.g. Folds, Paramorphisms, Unfolds, Apomorphisms, ...)

Folds as Initial Algebras

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

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

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



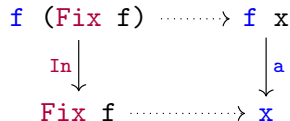
Folds as Initial Algebras

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

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

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

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

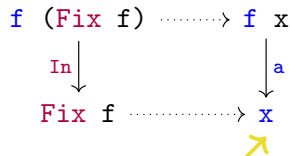


Folds as Initial Algebras

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

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

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



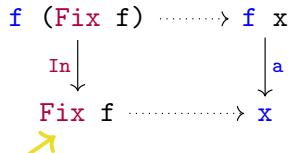
f-algebra

Folds as Initial Algebras

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

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

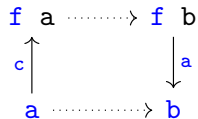
```
fold a = f  
  where f (In x) = (a . fmap f) x
```



initial `f`-algebra

Hylomorphisms: Divide-and-conquer Computations

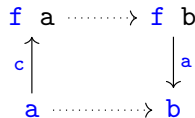
```
hylo :: Functor f =>  
    (f b -> b) ->  
    (a -> f a) ->  
    a -> b  
hylo a c = a . fmap (hylo a c) . c
```



Hylomorphisms: Divide-and-conquer Computations

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

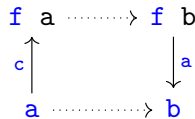
```
hylo a c = a . fmap (hylo a c) . c
```



f -coalgebra
"divide"

Hylomorphisms: Divide-and-conquer Computations

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



f-algebra
"conquer"

Folds as Hylomorphisms

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

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

```
fold a = a ← fmap (fold a) . in0p
```

f-coalgebra

$f \text{ (Fix } f) \cdots \cdots \rightarrow f \text{ } x$
 $\text{in0p} \uparrow \quad \downarrow a$
 $\text{Fix } f \cdots \cdots \rightarrow x$

f-algebra

Example: Nonstructural Recursion

```
data TreeF a b = Leaf | Node b a b
```

```
split [] = Leaf
```

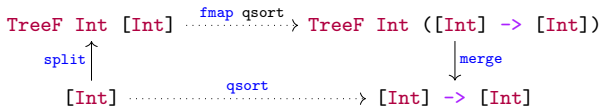
```
split (h : t) = Node l h r
```

```
  where
```

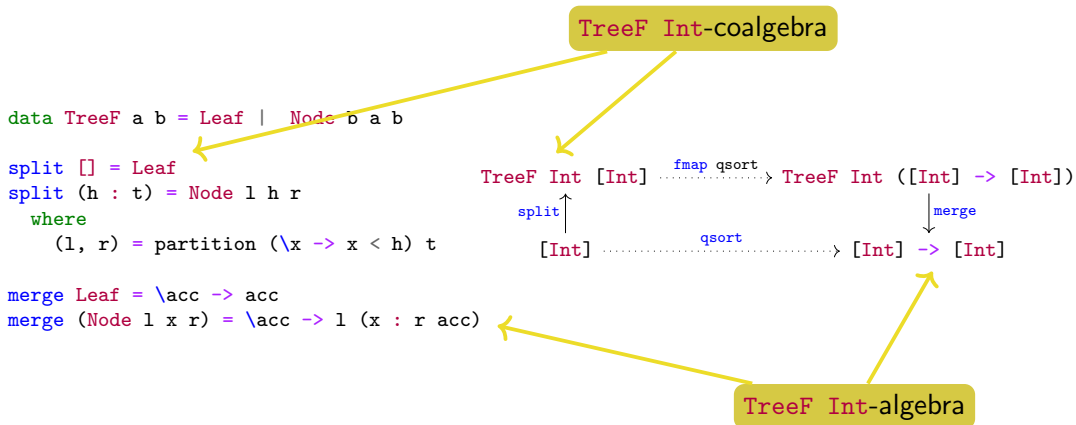
```
    (l, r) = partition (\x -> x < h) t
```

```
merge Leaf = \acc -> acc
```

```
merge (Node l x r) = \acc -> l (x : r acc)
```



Example: Nonstructural Recursion



Conjugate Hylomorphisms

Every recursion scheme is a conjugate hylomorphism

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

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

Why Mechanising Hylomorphisms in Coq?

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

Challenges

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

Challenges

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

Our solutions (the remainder of this talk):

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

Challenges

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

Our solutions (the remainder of this talk):

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

Challenges

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

Our solutions (the remainder of this talk):

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

Roadmap

Part I: Extractable Containers in Coq

Part II: Recursive Coalgebras & Coq Hylomorphisms

Part III: Code Extraction & Examples

Part I

Extractable Containers in Coq

Setoids and Morphisms

To avoid the functional extensionality axiom, we use:

- **setoids**: types with an associated equivalence
- **proper morphisms** of the respectfulness relation: functions that map related inputs to related outputs

Setoids: Given `setoid A`, and `x y : A`, we write `x =e y : Prop`.

Morphisms: Given `setoid A` and `setoid B`, we write `f : A ~> B`.

Code Extraction for Setoids and Morphisms

We add wrappers on top of Coq's standard Setoids and Proper Morphisms.

Every type must have **exactly one** associated equivalence.

Morphisms are records with a function, and a proof that it respects the relations.

Code Extraction for Setoids and Morphisms

We add wrappers on top of Coq's standard Setoids and Proper Morphisms.

Every type must have **exactly one** associated equivalence.

Morphisms are records with a function, and a proof that it respects the relations.

- We provide automatic coercion to functions.
- Coq's extraction mechanism ignores the `Prop` field.

Code Extraction for Setoids and Morphisms

We add wrappers on top of Coq's standard Setoids and Proper Morphisms.

Every type must have **exactly one** associated equivalence.

Morphisms are records with a function, and a proof that it respects the relations.

- We provide automatic coercion to functions.
- Coq's extraction mechanism ignores the **Prop** field.
- We provide a (very basic!) mechanism to help building morphisms.

Code Extraction for Setoids and Morphisms

We add wrappers on top of Coq's standard Setoids and Proper Morphisms.

Every type must have **exactly one** associated equivalence.

Morphisms are records with a function, and a proof that it respects the relations.

- We provide automatic coercion to functions.
- Coq's extraction mechanism ignores the **Prop** field.
- We provide a (very basic!) mechanism to help building morphisms.
- We allow the use of Coq's **generalised rewriting** on any morphism or morphism input.

Containers

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

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

Containers

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

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

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

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

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

Example

Container for $F X = 1 + X \times X$?

$$S_F = 1 + 1 \quad P_F = \lambda \left\{ \begin{array}{l} \text{inl } \bullet, 0 \\ \text{inr } \bullet, 1 + 1 \end{array} \right\}$$


$$F \mathbb{N} \cong \llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$$

$$\begin{aligned} \text{inl } \bullet &\cong (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) &\cong (\text{inr } \bullet, \lambda \left\{ \begin{array}{l} \text{inl } \bullet, 7 \\ \text{inr } \bullet, 9 \end{array} \right\}) \end{aligned}$$

Example

Two cases (“shapes”)

Container for $F X = 1 + X \times X$?


$$S_F = 1 + 1 \quad P_F = \lambda \left\{ \begin{array}{l} \text{inl } \bullet, 0 \\ \text{inr } \bullet, 1 + 1 \end{array} \right\}$$

$$F \mathbb{N} \cong \llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$$

$$\begin{aligned} \text{inl } \bullet &\cong (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) &\cong (\text{inr } \bullet, \lambda \left\{ \begin{array}{l} \text{inl } \bullet, 7 \\ \text{inr } \bullet, 9 \end{array} \right\}) \end{aligned}$$

Example

No positions on the left shape

Container for $F X = 1 + X \times X$?

$$S_F = 1 + 1 \quad P_F = \lambda \left\{ \begin{array}{l} \text{inl } \bullet, 0 \\ \text{inr } \bullet, 1 + 1 \end{array} \right\}$$

$$F \mathbb{N} \cong \llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$$

$$\begin{aligned} \text{inl } \bullet &\cong (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) &\cong (\text{inr } \bullet, \lambda \left\{ \begin{array}{l} \text{inl } \bullet, 7 \\ \text{inr } \bullet, 9 \end{array} \right\}) \end{aligned}$$

Example

Two positions on the right shape

Container for $F X = 1 + X \times X$?

$$S_F = 1 + 1 \quad P_F = \lambda \left\{ \begin{array}{l} \text{inl } \bullet, 0 \\ \text{inr } \bullet, 1 + 1 \end{array} \right\}$$

$$F \mathbb{N} \cong \llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$$

$$\begin{aligned} \text{inl } \bullet &\cong (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) &\cong (\text{inr } \bullet, \lambda \left\{ \begin{array}{l} \text{inl } \bullet, 7 \\ \text{inr } \bullet, 9 \end{array} \right\}) \end{aligned}$$

Containers in Coq: A Bad Attempt

```
Record Cont := { Shape : Type; Pos : Shape -> Type };
```

```
Record App (C : Cont) (X : Type) :=  
  MkCont { shape : Shape C; contents : Pos shape -> X }.
```

Containers in Coq: A Bad Attempt

```
Record Cont := { Shape : Type; Pos : Shape -> Type };
```

```
Record App (C : Cont) (X : Type) :=  
  MkCont { shape : Shape C; contents : Pos shape -> X }.
```

- The above definition forces us to use dependent equality and UIP/Axiom K/... E.g.: dealing with `eq_dep s1 p1 s2 p2` if `p1 : Pos s1` and `p2 : Pos s2`.
- Type families lead to OCaml code with `Obj.magic`.

Extractable Containers in Coq (I)

Solutions:

1. UIP is **not an axiom** for types with a **decidable equality**.
2. If a type family is defined as a **predicate subtype**, Coq can erase the predicate and extract code that is equivalent to the supertype. E.g. $\{x \mid P\ x\}$ for some $P : X \rightarrow \text{Prop}$.

Extractable Containers in Coq (and II)

```
Class Cont (Sh : Type) {_ : setoid Shape} (Po : Type)
  := { valid : Sh * Po ~> bool };
```

```
Record Pos `{Cont Sh Po} (s : Sh)
  := MkPos { elem : Po; Valid : valid (s, elem) };
```

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

Example: $F X = 1 + X \times X$

```
Inductive ShapeF := Lbranch | Rbranch.
```

```
Inductive PosF := Lpos | Rpos.
```

```
Definition validF : ShapeF * PosF ~> bool.
```

```
|{ x ~> match fst x with | Lbranch => false | Rbranch => true end }|.
```

```
Defined.
```

```
Instance TreeC : Cont ShapeF PosF (* ... validF ... *)
```

```
(** inr (7, 8) **)
```

```
Example e1 : App TreeC nat :=
```

```
  MkCont Rbranch (fun p =>
```

```
    match elem p with
```

```
      | Lpos => 7
```

```
      | Rpos => 8
```

```
    end).
```

Container Equality

We can define least/greatest fixed points of container extensions.

We provide a library of polynomial functors as containers, as well as custom shapes that we use in our examples.

Not discussed:

- Container morphisms and natural transformations
- Container composition $S \triangleleft P = (S_1 \triangleleft P_1) \circ (S_2 \triangleleft P_2)$
- “Nesting” of containers

Part II

Recursive Coalgebras & Coq Hylomorphisms

Container Initial Algebras

Container Terminal Coalgebras

Recursive Coalgebras

Recursive Hylomorphisms

Universal Property of Recursive Hylomorphisms

Proving the Laws of Hylomorphisms

Part III

Code Extraction & Examples

A Tree Container for Divide-and-conquer Computations

Quicksort Definition

Quicksort Extraction

Using Hylo-fusion for Program Optimisation

Optimized Code Extraction

Dynamorphisms

Knapsack

Knapsack Extraction

Wrap-up

