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

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

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

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

Folds as Initial Algebras

Least Fixed-Point

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

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

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

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

Folds as Initial Algebras

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

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

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

f-algebra

Folds as Initial Algebras

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

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

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

initial f-algebra

Hylomorphisms: Divide-and-conquer Computations

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

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

Hylomorphisms: Divide-and-conquer Computations

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

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

f-coalgebra
“divide”

Hylomorphisms: Divide-and-conquer Computations

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

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

f-algebra
“conquer”

Folds as Hylomorphisms

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

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

f-coalgebra

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

f-algebra

Example: Nonstructural Recursion

```haskell
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)
```

$$\begin{tikzcd}
\texttt{TreeF Int [Int]} \arrow[r, dotted, "\texttt{fmap qsort}"] & \texttt{TreeF Int ([Int] -> [Int])} \arrow[d, "\texttt{merge}"] \\
\texttt{[Int]} \arrow[u, "\texttt{split}"] \arrow[r, dotted, "\texttt{qsort}"] & \texttt{[Int] -> [Int]}
\end{tikzcd}$$

Example: Nonstructural Recursion

TreeF Int-coalgebra

```haskell
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)
```

$$
\begin{tikzcd}
\texttt{TreeF Int [Int]} \arrow[r, dotted, "\texttt{fmap qsort}"] & \texttt{TreeF Int ([Int] -> [Int])} \arrow[d, "\texttt{merge}"] \\
\texttt{[Int]} \arrow[u, "\texttt{split}"] \arrow[r, dotted, "\texttt{qsort}"] & \texttt{[Int] -> [Int]}
\end{tikzcd}
$$

TreeF Int-algebra

Conjugate Hylomorphisms

Every recursion scheme is a conjugate hylomorphism

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

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

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

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 $S \lhd P$:

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

Containers

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

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

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

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

Example

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

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

$$\frac{F\ \mathbb{N} \quad \cong \quad [\![S_F \lhd P_F]\!]\ \mathbb{N}}{\begin{array}{rcl} \mathsf{inl}\ \bullet & \cong & (\mathsf{inl}\ \bullet, !_{\mathbb{N}}) \\ \mathsf{inr}\ (7, 9) & \cong & (\mathsf{inr}\ \bullet, \lambda \left\{ \begin{array}{l} \mathsf{inl}\ \bullet, 7 \\ \mathsf{inr}\ \bullet, 9 \end{array} \right\}) \end{array}}$$

Example

Two cases ("shapes")

Container for $F\ X = \boxed{1} + \boxed{X \times X}$?

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

$$\frac{F\ \mathbb{N} \quad \cong \quad [\![S_F \triangleleft P_F]\!]\ \mathbb{N}}{}$$

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

Example

No positions on the left shape

Container for $F\ X = \boxed{1} + X \times X$?

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

$$\frac{F\ \mathbb{N} \quad \cong \quad [\![S_F \triangleleft P_F]\!]\ \mathbb{N}}{\begin{array}{rcl} \mathsf{inl}\ \bullet & \cong & (\mathsf{inl}\ \bullet, !_{\mathbb{N}}) \\ \mathsf{inr}\ (7, 9) & \cong & (\mathsf{inr}\ \bullet, \lambda \left\{ \begin{array}{l} \mathsf{inl}\ \bullet, 7 \\ \mathsf{inr}\ \bullet, 9 \end{array} \right\}) \end{array}}$$

Example

Two positions on the right shape

Container for $F\ X = 1 + \boxed{X \times X}$?

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

$$\frac{F\ \mathbb{N} \quad \cong \quad [\![S_F \lhd P_F]\!]\ \mathbb{N}}{\begin{array}{rcl} \mathsf{inl}\ \bullet & \cong & (\mathsf{inl}\ \bullet, !_{\mathbb{N}}) \\ \mathsf{inr}\ (7, 9) & \cong & (\mathsf{inr}\ \bullet, \lambda \left\{ \begin{array}{l} \mathsf{inl}\ \bullet, 7 \\ \mathsf{inr}\ \bullet, 9 \end{array} \right\}) \end{array}}$$

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 | P x}` for some `P : X -> 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 $S \lhd P = (S_1 \lhd P_1) \circ (S_2 \lhd P_2)$
- "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