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

University of Kent

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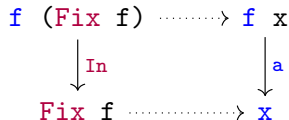
There are many different kinds of Recursion Schemes (e.g. Folds, Paramorphisms, Unfolds, Apomorphisms, ...)

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
```

$$
\begin{array}{ccc}
f\ (Fix\ f) & \dashrightarrow & f\ x \\
\downarrow{\scriptstyle In} & & \downarrow{\scriptstyle a} \\
Fix\ f & \dashrightarrow & x
\end{array}
$$

Folds as Initial Algebras

```
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
```
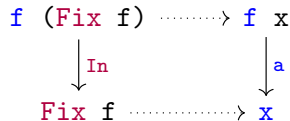
Least Fixed-Point
`Fix f ≅ f (Fix f)`

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
```
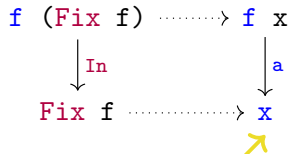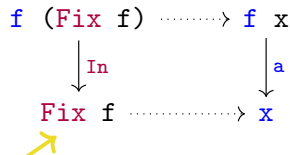
Folds as Initial Algebras

```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
```

```
f (Fix f) ┄┄┄┄> f x
    │                │
 In │                │ a
    ↓                ↓
  Fix f  ┄┄┄┄┄> x
```
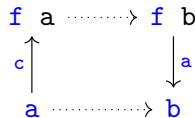
initial f-algebra

Hylomorphisms: Divide-and-conquer Computations
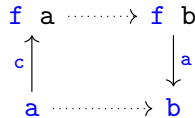
```haskell
hylo :: Functor f =>
          (f b -> b) ->
          (a -> f a) ->
          a -> b
hylo a c = a  . fmap (hylo a c) . c
```

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
```


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 a ⋯⋯⋯⋯> f b

c ↑   ↓ a

a ⋯⋯⋯⋯⋯> b
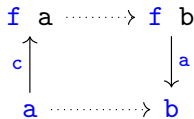
f-algebra
"conquer"

Folds as Hylomorphisms

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

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

Adjoint Folds

Given an adjunction:

$$\mathcal{D} \underset{\mathsf{L}}{\overset{\mathsf{R}}{\rightleftarrows}} \mathcal{C}$$

- There is a correspondence of arrows $\lfloor \cdot \rfloor : \mathsf{Hom}_{\mathcal{D}}(L\,A, B) \cong \mathsf{Hom}_{\mathcal{C}}(A, R\,B) : \lceil \cdot \rceil$.
- An initial algebra on the right corresponds to an universal property on the left:

$$\mathsf{Hom}_{\mathcal{D}}(L\,\mu F, B) \cong \mathsf{Hom}_{\mathcal{C}}(\mu F, R\,B)$$

μ analogous to Haskell's `Fix`

F is an endofunctor in $\mathcal{C}$

L, R are functors between $\mathcal{C}$ & $\mathcal{D}$; the *left* and *right* adjoints.

Conjugate Hylomorphisms

Every recursion scheme is a conjugate hylomorphism

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

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

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

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

Part II

Recursive Coalgebras & Coq Hylomorphisms

Part III

Code Extraction & Examples

Wrap-up