

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)
```

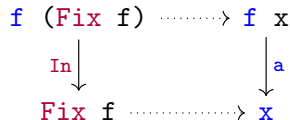
There are many different kinds of Recursion Schemes (e.g. Folds, Paramorphisms, Unfolds, Apomorphisms, ...)

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
```



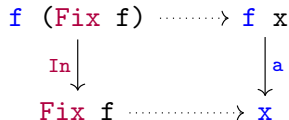
Folds as Initial Algebras

```
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
```

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

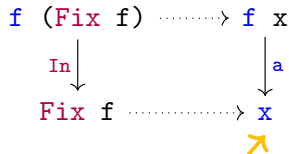


Folds as Initial Algebras

```
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
```



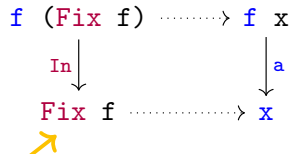
f-algebra

Folds as Initial Algebras

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

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

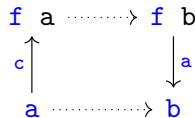
```
fold a = f  
  where f (In x) = (a . fmap f) x
```



initial `f`-algebra

Hylomorphisms: Divide-and-conquer Computations

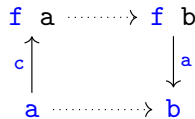
```
hylo :: Functor f =>  
    (f b -> b) ->  
    (a -> f a) ->  
    a -> b  
hylo a c = a . fmap (hylo a c) . c
```



Hylomorphisms: Divide-and-conquer Computations

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

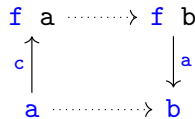
```
hylo a c = a . fmap (hylo a c) . c
```



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

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

f-algebra

Example: Nonstructural Recursion

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

```
split [] = Leaf
```

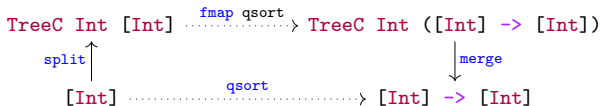
```
split (h : t) = Node l h r
```

```
  where
```

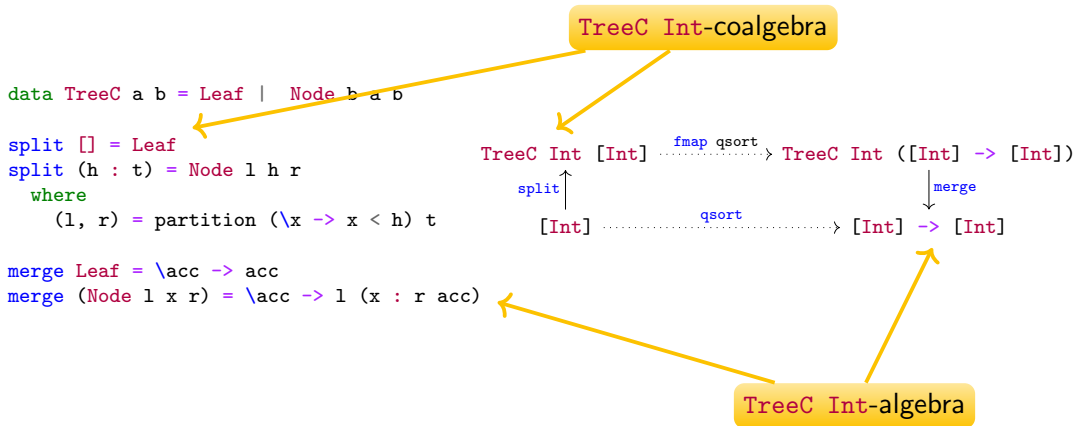
```
    (l, r) = partition (\x -> x < h) t
```

```
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.

Conjugate Hylomorphisms

- Every complex recursion scheme is an hylomorphism via its associated adjunction/conjugate pair
- (e.g) folds with parameters (accumulators) use the curry/uncurry adjunction
- A recursion scheme from comonads (RSFCs, Uustalu, Vene, Pardo, 2001) is an conjugate hylomorphism via the coEilenberg-Moore category for the cofree comonad

recursio

(hylo-sh

mutual

accumul

course-of-values (§5.6)	$\mathbf{U}_D \dashv \mathbf{Cofree}_D$	ccf	$x = a \cdot (id \triangle D (D_\infty x \cdot [c]) \cdot c) : A \leftarrow C$	$a : C \times D (D_\infty A) \rightarrow A$
finite memo-table (§5.6)	$\mathbf{U}_* \dashv \mathbf{Cofree}_*$	ccf	$x = a \cdot (id \triangle D (D_* x \cdot [c]_*) \cdot c) : A \leftarrow C$	$a : C \times D (D_* A) \rightarrow A$

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

Consider the functor $F X = 1 + X \times X$

S_F and P_F define a container that is isomorphic to F

$$S_F = 1 + 1 \quad \begin{array}{l} P_F (\text{inl } \bullet) = 0 \\ P_F (\text{inl } \bullet) = 1 + 1 \end{array}$$

Examples of objects of types $F \mathbb{N}$ (left) and $\llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$ (right):

$$\begin{array}{lcl} \text{inl } \bullet & \cong & (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) & \cong & (\text{inr } \bullet, \lambda x, \text{case } x \{ \text{inl } \bullet \Rightarrow 7; \text{inr } \bullet \Rightarrow 9 \}) \end{array}$$

Example

Consider the functor $F X = 1 + X \times X$

Two cases (“shapes”)

S_F and P_F define a container that is isomorphic to F

$$S_F = 1 + 1 \quad \begin{array}{l} P_F (\text{inl } \bullet) = 0 \\ P_F (\text{inr } \bullet) = 1 + 1 \end{array}$$

Examples of objects of types $F \mathbb{N}$ (left) and $\llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$ (right):

$$\begin{array}{lcl} \text{inl } \bullet & \cong & (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) & \cong & (\text{inr } \bullet, \lambda x, \text{case } x \{ \text{inl } \bullet \Rightarrow 7; \text{inr } \bullet \Rightarrow 9 \}) \end{array}$$

Example

No positions on the left shape

Consider the functor $F X = 1 + X \times X$

S_F and P_F define a container that is isomorphic to F

$$S_F = 1 + 1 \quad \begin{array}{l} P_F (\text{inl } \bullet) = 0 \\ P_F (\text{inl } \bullet) = 1 + 1 \end{array}$$

Examples of objects of types $F \mathbb{N}$ (left) and $\llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$ (right):

$$\begin{array}{ll} \text{inl } \bullet & \cong (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) & \cong (\text{inr } \bullet, \lambda x, \text{case } x \{ \text{inl } \bullet \Rightarrow 7; \text{ inr } \bullet \Rightarrow 9 \}) \end{array}$$

Example

Two positions on the right shape

Consider the functor $F X = 1 + X \times X$

S_F and P_F define a container that is isomorphic to F

$$S_F = 1 + 1 \quad \begin{array}{l} P_F (\text{inl } \bullet) = 0 \\ P_F (\text{inl } \bullet) = 1 + 1 \end{array}$$

Examples of objects of types $F \mathbb{N}$ (left) and $\llbracket S_F \triangleleft P_F \rrbracket \mathbb{N}$ (right):

$$\begin{array}{lcl} \text{inl } \bullet & \cong & (\text{inl } \bullet, !_{\mathbb{N}}) \\ \text{inr } (7, 9) & \cong & (\text{inr } \bullet, \lambda x, \text{case } x \{ \text{inl } \bullet \Rightarrow 7; \text{inr } \bullet \Rightarrow 9 \}) \end{array}$$

Containers in Coq: A Bad Attempt

Assume a `Shape : Type` and `Pos : Shape -> Type`.

We can define a container extension in the straightforward way:

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

Containers in Coq: A Bad Attempt

Assume a `Shape : Type` and `Pos : Shape -> Type`.

We can define a container extension in the straightforward way:

```
Record App (X : Type) :=  
  MkCont { shape : Shape; 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** in Coq 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)

Our containers are defined by:

- `Sh` : `Type`: type of shapes
- `Po` : `Type`: type of **all** positions
- `valid` : `Sh * Po -> bool`
decidable predicate stating when a pair shape/position is valid

Container extensions that lead to “clean” code extraction:

```
Record App (X : Type)
:= MkCont { shape : Sh;
            contents : {p | valid (shape, p)} -> X
          }.
```

Extractable Containers in Coq (and II)

Our container

- `Sh` :
- `Po` :
- `valid` :
 - All proofs of the form `V1 V2 : valid(s,p) = true` are provably equal in Coq to `eq_refl`.
 - Given `p1 p2 : {p | valid(s, p)}`, `p1 = p2` iff `proj1_sig p1 = proj1_sig p2`.
 - Extraction will treat the contents of container extensions equivalently to contents : `Po -> X`

Container (no unsafe coercions).

```
Record App (X : Type)
```

```
:= MkCont { shape : Sh;  
            contents : {p | valid (shape, p)} -> X  
          }.
```

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

Container definition:

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

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

```
Definition validF (x : ShapeF * PosF) : bool
```

```
:= match fst x with | Lbranch => false | Rbranch => true end.
```

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

Example object equivalent to `inr (7,8)`

```
Example e1 : App nat :=  
  MkCont Rbranch (fun p => match elem p with  
    | Lpos => 7 | Rpos => 8  
  end).
```


The argument of container extensions occurs in strictly positive positions:

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

We provide a library of polynomial functors as containers, as well as custom shapes (e.g. binary trees) that we use in our examples.

Not discussed:

- Container morphisms and natural transformations
- Container composition $S \triangleleft P = (S_1 \triangleleft P_1) \circ (S_2 \triangleleft P_2)$
- Container equality

Part II

Recursive Coalgebras & Coq Hylomorphisms

Container Initial Algebras

The least fixed-point of a container extension `App C` is:

```
Inductive LFix C := Lin { lin_op : App C (LFix C) }.
```

Algebras are of type `Alg C X = App C X -> X`.

Cartamorphisms:

```
cata : Alg C X -> LFix C -> X
```

```
cata_univ : forall (a : Alg C X) (f : LFix C -> X),  
  f \o Lin =e a \o fmap f <-> f =e cata a
```

Container Terminal Coalgebras

The greatest fixed-point of a container extension `App C` is:

```
CoInductive GFix C := Gin { gin_op : App C (GFix C) }.
```

Coalgebras are of type `CoAlg C X = X ~> App C X`.

Anamorphisms:

```
ana : CoAlg C X ~> X ~> GFix C
```

```
ana_univ : forall (c : CoAlg C X) (f : X ~> GFix C),  
  gin_op \o f =e fmap f \o c <-> f =e ana c
```

Recursive Coalgebras (I)

We cannot compose `cata` and any arbitrary `ana...`

Recursive Coalgebras (I)

We cannot compose `cata` and any arbitrary `ana`...

But we can, if `ana` is applied to a **recursive coalgebra**.

Recursive Coalgebras (I)

We cannot compose `cata` and any arbitrary `ana`...

But we can, if `ana` is applied to a **recursive coalgebra**.

Recursive coalgebras: `coalgebras (c : CoAlg C X)` that terminate in all inputs.

- i.e. their anamorphisms only produce finite trees.

Recursive Coalgebras (I)

We cannot compose `cata` and any arbitrary `ana`...

But we can, if `ana` is applied to a **recursive coalgebra**.

Recursive coalgebras: coalgebras ($c : \text{CoAlg } C \ X$) that terminate in all inputs.

- i.e. their anamorphisms only produce finite trees.
- i.e. they decompose inputs into “smaller” values of type X

Recursive Coalgebras (and II)

We define a predicate $\text{RecF } c \ x$ that states that $c : \text{CoAlg } C \ X$ terminates on $x : X$.

Using RecF , we define:

- Recursive coalgebras:

$$\text{RCoAlg } C \ X = \{c \mid \text{forall } x, \text{RecF } c \ x\}$$

- Given a well-founded relation R , well-founded coalgebras

$$\text{WfCoAlg } C \ X = \{c \mid \text{forall } x \ p, R \ x \ (\text{contents } (c \ x) \ p)\}$$

Recursive Coalgebras (and II)

We define a predicate $\text{RecF } c \ x$ that states that $c : \text{CoAlg } C \ X$ terminates on $x : X$.

Using RecF , we define:

- Recursive coalgebras:

$$\text{RCoAlg } C \ X = \{c \mid \text{forall } x, \text{RecF } c \ x\}$$

- Given a well-founded relation R , well-founded coalgebras

$$\text{WfCoAlg } C \ X = \{c \mid \text{forall } x \ p, R \ x \ (\text{contents } (c \ x) \ p)\}$$

- Definitions (1) and (2) are equivalent

Recursive Coalgebras (and II)

We define a predicate $\text{RecF } c \ x$ that states that $c : \text{CoAlg } C \ X$ terminates on $x : X$.

Using RecF , we define:

- Recursive coalgebras:

$$\text{RCoAlg } C \ X = \{c \mid \text{forall } x, \text{RecF } c \ x\}$$

- Given a well-founded relation R , well-founded coalgebras

$$\text{WfCoAlg } C \ X = \{c \mid \text{forall } x \ p, R \ x \ (\text{contents } (c \ x) \ p)\}$$

- Definitions (1) and (2) are equivalent
- Our mechanisation represents (2) in terms of (1)

Recursive Coalgebras (and II)

We define a predicate $\text{RecF } c \ x$ that states that $c : \text{CoAlg } C \ X$ terminates on $x : X$.

Using RecF , we define:

- Recursive coalgebras:

$$\text{RCoAlg } C \ X = \{c \mid \text{forall } x, \text{RecF } c \ x\}$$

- Given a well-founded relation R , well-founded coalgebras

$$\text{WfCoAlg } C \ X = \{c \mid \text{forall } x \ p, R \ x \ (\text{contents } (c \ x) \ p)\}$$

- Definitions (1) and (2) are equivalent
- Our mechanisation represents (2) in terms of (1)
- Termination proofs may be easier using (1) or (2), depending on the use case

Recursive Hylomorphisms

Recall: hylomorphisms are solutions to the equation $f = a \circ \text{fmap } f \circ c$.

But, due to termination, this solution may not exist, or may not be unique.

However, if c is recursive, then the solution **is unique, and guaranteed to exist**.

Recursive Hylomorphisms

Recall: hylomorphisms are solutions to the equation $f = a \circ \text{fmap } f \circ c$.

But, due to termination, this solution may not exist, or may not be unique.

However, if c is recursive, then the solution **is unique, and guaranteed to exist**.

```
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 (shape cx), RecF c (cont cx e)) -> B
    with
    | MkCont sx cx => fun H => a (MkCont sx (fun e => f (cx e) (H e)))
    end (RecF_inv H).
```

Universal Property of Recursive Hylomorphisms

We define wrappers over `hylo_def`:

```
hylo : Alg C B ~> RCoAlg C A ~> A ~> B
```

From this definition, we can prove the universal property of hylomorphisms.

Given `a : Alg C B` and `c : RCoAlg C A`:

```
hylo_univ : forall f : A ~> B,  
  f =e a \o fmap f \o c <-> f = hylo a c
```

A Note on Recursive Anamorphisms

For simplicity, we define recursive anamorphisms as $\text{rana } c = \text{hylo } \text{Lin } c$.

- This way we avoid the need to convert GFix to LFix .
- We prove (straightforward) that $\text{rana } c$ is equal to $\text{ana } c$, followed by converting the result to LFix .

Proving the Laws of Hylomorphisms

The following `hylo_fusion` laws are straightforward consequences of `hylo_univ`.

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

Proving the Laws of Hyломorphisms

The following `hylo_fusion` laws are straightforward consequences of `hylo_univ`.

```
Lemma hylo_fusion_l
  : h \o a =e b \o fmap h -> h \o hylo a c =e hylo b c.
```

The proofs in Coq are almost direct copies from pen-and-paper proofs: By `hylo_univ`, `hylo b c` is the only arrow making the outer square commute.

$$\begin{array}{ccccc} tb & \xleftarrow{h} & ta & \xleftarrow{x} & tc \\ b \uparrow & & a \uparrow & & \downarrow c \\ f\ tb & \xleftarrow{\text{fmap } h} & f\ ta & \xleftarrow{\text{fmap } x} & f\ tc \end{array}$$

- Our formalisation allows to do equational reasoning that closely mirrors pen-and-paper proofs.
- `hylo_fusion` can be applied to *calculate* optimised programs by fusing simpler specifications in Coq.
- This leads to more modular development and proofs, without affecting the performance of the extracted code.

Part III

Code Extraction & Examples

A Tree Container for Divide & Conquer

Our divide-and-conquer examples use a tree container `TreeC A B` that is isomorphic to:

$$T\ A\ B\ X = A + B \times X \times X$$

Given two setoids `A` and `B`, we define the following wrappers in Coq:

```
a_node : B ~> X ~> X ~> App (TreeC A B) X
a_leaf  : A ~> App (TreeC A B) X
a_out   : App (TreeC A B) X ~> A + B * X * X
```

Quicksort Definition

```
Definition mergeF (x : App (TreeC unit int) (list int)) : list int :=  
  match a_out x with  
  | inl _ => nil  
  | inr (p, l, r) => List.app l (h :: r)  
end.
```

```
Definition splitF (l : list int) : App (TreeC unit int) (list int) :=  
  match x with  
  | nil => a_leaf tt  
  | cons h t => let (l, r) := List.partition (fun x => x <=? h) t in  
    a_node h l r  
end.
```

Quicksort Extraction

```
Definition qsort := hylo merge split.  
Extraction qsort.
```

Quicksort Extraction

Definition qsort := hylo merge split.
Extraction qsort.

```
let rec qsort = function
| [] -> []
| h :: t ->
  let (l, r) = partition (fun x0 -> leb x0 h) t in
  let x0 = fun e -> qsort (match e with
                           | Lbranch -> l
                           | Rbranch -> r) in
  app (x0 Lbranch) (h :: (x0 Rbranch))
```


Using Hylo-fusion for Program Optimisation

```
Definition qsort_times_two
  : {f | f =e map times_two \o hylo merge split}.
  eapply exist.
  (* ... *)
  rewrite (hylo_fusion_1 H); reflexivity.
Defined.

Extraction qsort_times_two.
```

Using Hylo-fusion for Program Optimisation

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

A Recursion Scheme for Dynamic Programming

Given a functor G , we can construct a memoisation table $G_*A = \mu X.A \times GX$.
We can index the memoisation table, extract its head, and insert a new element:

$$\text{look} : \mathbb{N} \times G_*A \rightarrow 1 + A \quad \text{head} : G_*A \rightarrow A \quad \text{Cons} : A \times G(G_*A) \rightarrow G_*A$$

Given an algebra $a : G(G_*A) \rightarrow A$, we can construct

$$a' = \text{in} \circ \text{Cons} \circ \text{pair } a \text{ id} : G(G_*A) \rightarrow G_*A$$

a' computes the current value, as well as storing it in the memoisation table.

A Recursion Scheme for Dynamic Programming

Given a functor G , we can construct a memoisation table $G_*A = \mu X. A \times GX$.
We can index the memoisation table, extract its head, and insert a new element:

$$\text{look} : \mathbb{N} \times G_*A \rightarrow 1 + A \quad \text{head} : G_*A \rightarrow A \quad \text{Cons} : A \times G(G_*A) \rightarrow G_*A$$

Given an algebra $a : G(G_*A) \rightarrow A$, we can construct

$$a' = \text{in} \circ \text{Cons} \circ \text{pair } a \text{ id} : G(G_*A) \rightarrow G_*A$$

a' computes the current value, as well as storing it in the memoisation table.

Dynamorphisms: $\text{dyna } a \text{ } c = \text{head} \circ \text{hylo } a' \text{ } c$

Knapsack

```
Definition knapsack_alg (wvs : list (nat * int))
  (x : App NatF (Table NatF int)) : int
:= match x with
  | MkCont sx kx =>
    match sx with
    | inl tt => fun _ => 0
    | inr tt => fun kx => let table := kx posR in
                          max_int 0 (memo_knap table wvs)
    end kx
end.
```

Knapsack

```
let knapsack wvs x =  
  ((let rec f n =  
    if n=0 then  
      { lFix_out = { shape = UInt63.of_int 0;  
                    cont  = fun _ -> f 0 } }  
    else  
      let fn = f (n-1) in  
      { lFix_out = { shape = max_int (UInt63.of_int 0)  
                    (memo_knapsack fn wvs);  
                    cont = fun e -> fn } }  
  ) in f x).lFix_out.shape
```

Wrap-up

Summary

Hylomorphisms in Coq

- Modular specification of functions, without sacrificing performance thanks to `hylo_fusion`.
- Modular treatment of divide-and-conquer and termination proofs using recursive coalgebras.
- Clean OCaml code extraction.

Summary

Hylomorphisms in Coq

- Modular specification of functions, without sacrificing performance thanks to `hylo_fusion`.
- Modular treatment of divide-and-conquer and termination proofs using recursive coalgebras.
- Clean OCaml code extraction.

Future work:

- Improve extraction & inlining.
- Effects.
- Dealing with setoids & equalities.