Mechanising Recursion Schemes with Magic-Free Coq Extraction

David Castro-Perez
School of Computing, University of Kent

Marco Paviotti
School of Computing, University of Kent

Michael Vollmer
School of Computing, University of Kent

Abstract

Generic programming with recursion schemes provides a powerful abstraction for structuring recursion while ensuring termination and providing reasoning about program equivalences, as well as deriving optimisations which have been successfully applied to functional programming. Formalising recursion schemes in a type theory offers additional termination guarantees, but it often requires compromises affecting the resulting code, such as imposing performance penalties, requiring the assumption of additional axioms, or introducing unsafe casts into extracted code (e.g. `Obj.magic` in OCaml).

This paper presents the first Coq formalisation to our knowledge of a recursion scheme, called the *hylomorphism*, along with its algebraic laws allowing for the mechanisation of all recognised (terminating) recursive algorithms. The key contribution of this paper is that this formalisation is fully axiom-free allowing for the extraction of safe, idiomatic OCaml code. We exemplify the framework by formalising a series of algorithms based on different recursive paradigms such as divide-and conquer, dynamic programming, and mutual recursion and demonstrate that the extracted OCaml code for the programs formalised in our framework is efficient, resembles code that a human programmer would write, and contains no occurrences of `Obj.magic`. We also present a machine-checked proof of the well-known short-cut fusion optimisation.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases hylomorphisms, Coq, code extraction, verification

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Recursive definitions cannot be proven well-defined automatically due to the halting problem. Modern proof assistants like Coq or Agda provide a sound, but incomplete algorithm which syntactically checks for termination or productivity. For recursion, this is done by automatically inferring which argument in the recursive call gets smaller with respects to the original input argument. For productivity, the algorithm checks that the corecursive call appears directly under a constructor to make sure that this function always produces at least one element after each recursive step. This implies that some functions, though well-defined, cannot be accepted by the proof assistant. Examples of this include common sorting algorithms, such as quicksort:

```
let rec qsort xs = match divide xs with | None -> []
  | Some (pivot, (smaller, larger)) -> qsort smaller @ (pivot::qsort larger)
```

While `qsort` is a well-defined mathematical function it cannot be accepted by a proof assistant. The reason is that the `divide` function *destructuring* the input dives deeper in the input returning two sublists with the head as a pivot.

The main approach for implementing non-structural recursion in Coq is to use *well-founded recursion*, where recursive definitions are coupled with *termination proofs*. Using well-founded recursion, the recursive calls will happen on *structurally smaller termination proofs*[1]. A benefit of this approach is that, extracting verified nonstructural recursive functions to OCaml will erase the termination proofs, and so produce code that will be closer to what a programmer may have written directly in OCaml. However, reasoning about program equivalences requires dealing with such termination proofs, and it is common practice to use custom reduction lemmas that will be used extensively when proving properties about nonstructural recursive functions.

Structured recursion schemes have been successfully employed in functional programming to structure recursive programs, for example, quicksort and mergesort are both instances of a divide-and-conquer algorithm which in terms of recursion schemes can be can be formalised as an hylomorphism [22, 17]. Furthermore, it has been shown by Hinze et al. [16] that hylomorphisms provide the basic building block of every recursion scheme. In particular, any complex recursion scheme can be transmogrified down into an hylomorphisms by means of an adjunction. Furthermore, hylomorphism laws can capture a number of useful equivalences, ranging from common optimisations such as *short-cut fusion* [27], to semi-automatic parallelisations [12, 7].

However, when used in the context of languages with general recursion like Haskell, recursion schemes cannot ensure termination, but while formalising recursion schemes in a type theory does provide stronger termination guarantees, to the best of our knowledge, not many attempts have been made at mechanising structured recursion schemes.

Recently, Abreu et al. [3] encoded an algebraic approach to divide-and-conquer computations in which termination is entirely enforced by the typing discipline. Their approach solves the problem of termination proofs as well as the performance of the code that is run *within Coq*, but it does not allow for extraction of idiomatic OCaml code. This is problematic since code extration has proven useful in a variety of scenarios [25, 21, 23, 26]. However, in Abreu et al.'s approach, extraction (1) do not preserve the recursive structure of common implementations; and (2) lead to unsafe casts like `Obj.magic` in the generated code. This latter is also problematic in that, for higher-order programs, simple interoperations can lead to incorrect behaviour or even segfaults [11] and, moreover, it invalidates the fast-and-loose principle [10].

This work presents the first Coq formalisation to our knowledge of *hylomorphisms* that (1) is *axiom-free* and (2) allows the extraction of idiomatic OCaml code. The full mechanisation can be found on Github[2]. While programmers still need to reason about the termination of their programs, our mechanisation will allow the use of the algebraic laws of hylomorphisms for program reasoning, as well as extracting to idiomatic code (see `qsort` in Section 4).

To summarise, our contributions are as follows. In this paper we provide a framework for *generic programming* with recursion schemes in Coq, with a proof of their algebraic laws and program equivalences for clean code extraction:

- In Section 2 we formalise the type of container functors ensuring the presence of least and greatest fixed-points for functors and suitably adapted for program extraction
- In Section 3 we formalise folds, unfolds and hylomorhisms and their universal properties
- In Section 4 we use the framework to formalise examples of divide-and-conquer, dynamic programming, and mutual recursion algorithms. Furthermore we verify the short-cut fusion optimisation and show the extracted optimised code to OCaml.

[1] In Coq, one approach is using the `Fix` combinator, in which recursion is done on an *accesibility* predicate on the input, which is a proof that there are no infinitely decreasing chains.

[2] https://github.com/dcastrop/coq-hylomorphisms

2 Mechanising Extractable Container Functors

In order to abstract recursion patters we need to be able to abstract away from the particular shape of the data. This is achieved by introducing the concept of *functors* which have the suitable fixed-point properties, i.e. those who have a initial algebras. A common approach to construct such functors is to use *containers* [1] (Section 2.1). However, reasoning about container equality will require us to consider both functional extensionality and heterogeneous equality. We avoid these axioms by introducing a custom equivalence relation on types (Section 2.3).

2.1 Functors and Containers

Functors are functions `F : Type -> Type` which additionally have a map

```
fmap : forall A B, (A -> B) -> F A -> F B
```

witnessing the idea that a functor represents a container for abstract data that can be manipulated without changing the outer structure. For example, the type `List : Type -> Type` is a functor and `List A` is the type of lists over elements of type `A` with the obvious `fmap` function recursively traversing a list and applying the function `A -> B` for each element. Additionally, the type of lists `List A` arises as the least fixed-point of the functor `F X = unit + (A * X)`.

In general not every functor has a fixed-point, and it is not possible to build the fixed-point of a functor such as `F` in Coq due to not being strictly positive. Due to this, we are going to restrict to *polynomial functors* as represented by containers. A container is defined by a type of *shapes* and a *family of position types* indexed by shapes.

```
Context (Shape : Type) (Pos : Shape -> Type).
```

For the functor `F` we introduced in the previous paragraph we would define `Shape` as `unit + A` indicating there are two constructors in the data type and one of these contains a piece of data of type `A`. Moreover, we would define the positions as `Pos (inl tt) = Empty_set` and `Pos (inr a) = unit` indicating that the first constructor does not have any type variables and the second constructor has one type variable.

At this point an *extension* of this container is a functor defined as follows:

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

with the obvious action on morphisms given by post-composition with `contents`.

```
Definition fmap (f : A -> B) (x : App C A) : App C B
  := {| shape := shape x; contents := fun e => f (contents x e) |}
```

In our running example, the type `App X`, for some `X` has two inhabitants. The first is a pair composed by `inl tt : Shape` and a function `Pos (inl tt) -> X`. This latter is in fact a function of type `Empty_set -> X`. This pair is, therefore isomorphic to the type `unit`, the type with only one inhabitant. The second inhabitant is the pair composed by `inr a : Shape`, for some `a : A`, and a function `Pos (inr a) -> X`. This latter type is equal to `unit -> X` which represents the elements of `X` and therefore is isomorphic to `X`. This particular pair of shapes and positions is therefore isomorphic to the type `unit + A * X`.

The correspondence between containers and polynomial functors has been formalised in this work and can be found in the accompanying code (file Container.v).

2.2 Extractable Containers

In the previous section we defined containers using dependent types. However, Ocaml's type system is not equipped to handle these. To solve this problem Coq's code extraction mechanism will insert unsafe casts.

Consider instead representing the positions of a container using a decidable validity predicate (`valid`) which assigns shapes to positions. We use boolean functions and coercions from `bool` to `Prop` to represent decidable predicates, similarly to SSReflect [15]. The family of positions for a given shape in a container can be recovered by defining:

```
Class Cont (Shape : Type) (APos : Type) := { valid : Shape -> APos -> bool }.
Record Pos `(C : Cont Shape APos) (shape : Shape) :=
  ValidPos { val : APos; IsValid : valid shape pos }.
```

Coq's code extraction will now be able to erase the validity predicate, and generate OCaml code that is free of unsafe casts. The OCaml code extracted for `Pos` will now be exactly the code extracted for `APos`. The decidability of the validity predicate is crucial for our purposes of remaining axiom-free. To illustrate this, suppose that we need to show the equality of two container extensions. We will need to show that, for the same positions, they will produce the same result. In Coq, the goal would look as follows:

```
k : Pos C s -> X
P1, P2 : valid s p
---------------------------
k (ValidPos p P1) = k (ValidPos p P2)
```

If `valid` was a regular proposition in `Prop`, it would not possible to prove the equality of `P1` and `P2`. However, by using a decidable predicate, if we know that `P1` and `P2` are of type `valid s p = true`, then we can prove without any axioms that `P1 = P2 = eq_refl`.

2.2.1 Equality of Container Extensions

Reasoning about the equality of container extensions is not entirely solved by using decidable validity predicates to define the families of positions. In general, we want to equate container extensions that have the same shape, and that, for equal shape and position, they return the same element. To avoid the use of the functional extensionality axiom, we capture this relation with the following inductive proposition in Coq:

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

Note that we do not care about the validity proof of the positions, only their value. This is to simplify (slightly) our proofs. This relation is trivially *reflexive*, *transitive*, and *symmetric*.

However, the use of a different equality for container extensions now forces us to deal with the fact that some types have different definitions of equality. In particular, we want to reason about the equality of functions of types such as `App F A -> B` (or `B -> App F A`). Since these types now come with their own equivalence, any function that manipulates them needs to be *respectful*. I.e. given `R : X -> X -> Prop` and `R' : Y -> Y -> Prop`, we want functions (morphisms) that satisfy the following property:

```
forall (x y : X), R x y -> R' (f x) (f y)
```

2.3 Types and Morphisms

We address the different forms of equality by defining a class of *setoids*, types with an associated equivalence relation, and considering only functions that respect the associated equivalences, or *proper* morphisms with respect to the function *respectfulness* relation. We use the type-class mechanism, instead of setoids in Coq's standard library, to help Coq's code extraction mechanism remove any occurence of custom equivalence relations in the extracted OCaml code. We use `=e` to denote the equivalence relation of a setoid. By default, we associate every Coq type with the standard propositional equality, unless a different equivalence is specified (we allow overlapping instances, and Coq's propositional equality takes the lowest priority). Given types `A` and `B`, with their respective equivalence relations `eA : A -> A -> Prop` and `eB : B -> B -> Prop`, the we define the type `A ~> B` to represent *proper* morphisms of the respectfulness relation of `R` to `R'`.

```
Structure morph A {eA : setoid A} B {eB : setoid B} :=
  MkMorph { app :> A -> B; app_eq : forall x y, x =e y -> app x =e app y }.
Notation "A ~> B" := (@morph A _ B _).
```

We rely on Coq's type class mechanism to fill in the necessary equivalence relations. Coq's code extraction mechanism will erase any occurrence of `Prop` in the code, so objects of type `A ~> B` will be extracted to the OCaml equivalent to `A -> B`. Note the implicit coercion from `A ~> B` to `A -> B`. On top of this, we define basic function composition and identity functions:

```
Notation "f \o g" = (comp f g).
Definition comp : (B ~> C) ~> (A ~> B) ~> A ~> C := ...
Definition id : A ~> A := ...
```

Using custom equivalences and proper morphisms, we redefine the definitions of container extensions and container equality. In particular, container extensions require a proper morphism to check the validity of positions in shapes, and container equality now uses equivalences of shapes and contained elements:

```
Class Cont `{setoid Shape} (Pos : Type) := { valid : Shape * Pos ~> bool }.
Inductive AppR `{F : Cont Sf Pf} `{setoid X} (x y : App F X) : Prop :=
  | AppR_ext (Es : shape x =e shape y)
             (Ek : forall e1 e2, val e1 = val e2 -> cont x e1 =e cont y e2).
```

Note the use of `=e` instead of Coq's standard equality. For positions, however, we chose to use Coq's propositional equality, since these would leads to simpler code. We explain why in Section 3.1, and the mechanisation of initial algebras of container extensions.

This definition leads to the well-known "setoid hell", which we mitigate by providing tactics and notations to automatically discharge proofs of `app_eq` for morphisms, whenever the types use the standard propositional equality, or a combination of propositional and extensional equality. However, our compositional approach allows us to build morphisms by plugging in other morphisms to our combinators. In our framework, our expectation is that the user-provided functions remain small, with relatively straightforward proofs of `app_eq`.

However, by using this mechanisation, we gain simplified proofs via the use of Coq's *Generalised Rewriting*. Since every morphism `f : A ~> B` satisfies the property that if `x =e y`, then `f x =e f y`, we can add every morphism as a proper element of Coq's respectfulness relation. In practice, this means that we can use the `rewrite` tactic on proofs of type `A =e B`, for arbitrary `A` and `B`, whenever they are used as arguments of morphisms, as well as Coq's `reflexivity`, `symmmetry`, and `transitivity` tactics. For example:

```
Goal forall `(f : A ~> B) `(g : B ~> C) `(h : C ~> D) (H : h \o g =e id),
  h \o (g \o f) =e f.
Proof.  rewrite compA, H.  reflexivity. Qed.
```

2.3.1 Polynomial Types

We define a number of equivalences for polynomial types.

```
Instance ext_eq (A : Type) `{eq_B : setoid B} : setoid (A -> B).
Instance pair_eq `{eq_A : setoid A} `{eq_B : setoid B} : setoid (A * B).
Instance sum_eq `{eq_A : setoid A} `{eq_B : setoid B} : setoid (A + B).
Instance prop_eq : setoid Prop.
Instance pred_sub `{eA : setoid A} {P : A -> Prop} : setoid {a : A | P a}.
```

Most of the definitions that involve functions and polynomial types are straightworward. Identity and composition are defined as `fun x => x` and `fun f g x => f (g x)` respectively, and the proofs that they are proper morphisms is straightforward, and automatically discharged by Coq. Products are built using function `fun f g x => (f x, g x)`, with the projections being the standard Coq `fst` and `snd` functions. Similarly, sum injections are encoded using Coq's `inl` and `inr` constructors, and pattern matching on them uses the function `fun f g x => match x with | inl y => f y | inr y => g y end`. The proofs that these morphisms are proper are straightforward. Finally, we also provide functions for currying/uncurrying, and flipping the arguments of a proper morphism. We force most of our definitions to be inlined, to help Coq's code extraction mechanism to inline as many of these combinators as possible.

We prove the isomorphisms of polynomial types and the equivalent container extensions. As an example, we discuss (informally) the isomorphisms of pairs with their equivalent container extensions. Suppose that we know that `App F X` is isomorphic to `A`, and `App G X` is isomorphic to `B`. Then we can show that `App (Prod F G) X` is isomorphic to `A * B`. If we have an element of type `App (Prod F G) X`, using the `inl` position, we can obtain `App F X`. Similarly, using `inr`, we can obtain `App G X`. Since these are the only two valid positions in the shape of pairs, we have finished. It is now sufficient to use the isomorphisms of `App F X` and `App G X` to obtain `A * B`. Similarly if we have `A * B`, we can first use the isomorphisms of `A` and `B` to obtain `App F X * App G X`, and then construct the necessary container extension. Given `p_inl : Pos l -> Pos (l * r)` (resp. `p_inr`) that act as `inl` (resp. `inr`) on product positions, and `case_pos : (Pos l -> X) -> (Pos r -> X) -> Pos (l * r) -> X` that pattern matches on the product positions, the functions that witness the isomorphism are:

```
Definition iso_pair (x : App (Prod F G) X) : App F X * App G X :=
  ({| shape := shape (fst x); cont := fun e => cont x (p_inl e) |},
   {| shape := shape (snd x); cont := fun e => cont x (p_inr e) |}).
Definition iso_prod (x : App F X * App G X) : App (Prod F G) X :=
{| shape := (shape (fst x), shape (snd x));
   cont := case_pos (cont (fst x)) (cont (snd x)) |}.
```

Proving that the composition of these functions is the identity is straightforward using the fact that `Prod` containers only have two valid positions.

3 Formalising Recursion Schemes

Recursion schemes provide an abstract way to consume and generate data. We now proceed onto describing how to formalise hylomorphisms in Coq. We first formalise algebras for container extensions (Section 3.1), then we formalise coalgebras (Section 3.2) and then we put together these notions to formalise recursive coalgebras and hylomorphisms (Section 3.3).

3.1 Algebras and Catamorphisms

An algebra is a set `A` (the *carrier* of the algebra) together with some operations on it sometimes subject to some equational laws. For example, given such a type `A` we can define the monoid operations as `u : unit -> A` for unit and `m : A * A -> A` for multiplication. Notice that `F X = unit + X * X` is a functor on `X` which means we can equivalently describe these two maps as a single one of type `F X -> X` which we call the algebra for the functor `F` satisfying the unit and associative laws of the monoid. In this work we will be using algebras to describe the operations of a data type and so we will not be needing additional equations on these operations.

Given a type `A` and a functor `F`, an `F`-algebra is a pair given by a type `A` called the *carrier* of a *structure map* of type `App F A ~> A`:

```
Notation Alg F A := (App F A ~> A).
```

The least fixed-point for a functor `F` is an instance of an `F`-algebra where the structure map is an isomorphism. This is sometimes referred to as the *initial algebra* for a functor `F`. We will explain the reason behind this name shortly. [1] We define the least fixed-point of `F` as an inductive type:

```
Inductive LFix `(F : Cont Sh P) : Type := LFix_in { LFix_out : App F (LFix F) }.
```

where `LFix_in` is the `F`-algebra while `LFix_out` is its inverse. As an example, the initial `F`-algebra for the functor `F X = unit + A * X` is the type of lists with the `F`-algebra being defined by the empty list `Empty : unit -> LFix F` and the cons operation `Cons : A * LFix F -> LFix F`.

We define `LFix` as a setoid, where its equivalence relation can be described as the least fixed point of `AppR` and we define smart constructors for the isomorphism of least fixed points as respectful morphisms:

```
l_in : App F (LFix F) ~> LFix F                  l_out : LFix F ~> App F (LFix F)
```

The least fixed-point for `F` is the initial `F`-algebra in the sense that it gives rise to an inductive recursion scheme. Specifically, for any other `F`-algebra there exists a *unique* map, known as a *fold* or *catamorphism*, such that it structurally deconstructs the data type using `LFix_out`, calls itself recursively and then composes the result of the recursive call using the given an `F`-algebra. In other words, for any give `F`-algebra there exists a unique `F`-algebra homomorphism from the initial one. We define it in Coq as follows:

```
  Definition cata_f `{F : Cont Sh P} (alg : Alg F A) : LFix -> A
:= fix f (x : LFix) := match x with
  | LFix_in ax => let (sx, kx) := ax in alg (MkApp sx (fun e => f (kx e))) end.
```

It is easy to show that this function is a respectful morphism of `F`-algebras. In fact, it is possible to define it as a map of the following type:

```
  cata : forall `{F : Cont Sh P} `{setoid A}, Alg F A ~> LFix ~> A
```

We prove that catamorphisms satisfy the universality property we explained previously:

[1] To be more precise, initial algebras are isomorphisms by Lambek's lemma, but isomorphisms of functors do not necessarily correspond to initial algebras.

```
Lemma cata_univ `{eA : setoid A} (alg : Alg F A) (f : LFix ~> A)
    : f =e cata alg <-> f =e alg \o fmap f \o l_out.
```

In other words, if there is any other `f` with the same structural recursive shape as the catamorphism on the algebra `alg` then it must be equal to that catamorphism.

3.2 Coalgebras and Anamorphisms

Algebras and catamorphisms dualise nicely to the coinductive setting. The dual of an algebra is a coalgebra. A coalgebra can be thought of as an *observation* map. For example, for a type of states `X` and an alphabet type `L` we can define a labelled transition system (LTS) on `X` as a function `c : X -> FX` for a functor `F X = L * X` implementing the *transition* map. In particular, for a state `x1 : X`, `c x1` returns a pair `(l, x2) : L * X` where `l : L` is the observable action and `x2 : X` is the next state.

In general, for a functor `F`, an `F`-coalgebra is a pair of a type `X` called the *carrier* of the coalgebra and a structure map `X -> F X`.

In our development we use the following notation for coalgebras:

```
Notation Coalg F A := (A ~> App F A).
```

Dually to the initial `F`-algebra, a final `F`-coalgebra is the greatest fixed-point for a functor `F`. We define it using a coinductive data type:

```
CoInductive GFix `(F : Cont Sh Po) : Type := GFix_in { GFix_out : App F GFix }.
```

where `GFix_out` is the final `F`-coalgebra and `GFix_in` is its inverse witnessing the isomorphism. Similarly to `LFix`, `GFix` is also defined as a setoid, with an equivalence relation that is the greatest fixpoint of `AppR`. Additionally, we define smart constructors for the isomorphism of greatest fixed points

```
g_in : App F (GFix F) ~> GFix F                g_out : GFix F ~> App F (GFix F)
```

The greatest fixed-point is a terminal `F`-coalgebra in the sense that it yields a coinductive recursion scheme. Specifically, for any other `F`-coalgebra there exists a *unique* map, called the *unfold* or *anamorphism*, such that it applies the observation map, corecursively generates the rest of the computation and composes the result of the corecursive call by using the algebra `GFix_in`. In other words, for any given `F`-coalgebra there exists a unique `F`-coalgebra homomorphism into the terminal `F`-coalgebra.

We define anamorphisms as follows:

```
Definition ana_f_ (c : Coalg F A) :=
  cofix f x :=
    match c x with | MkApp sx kx => GFix_in (MkApp sx (fun e => f (kx e))) end.

Definition ana : forall `{setoid A}, Coalg F A ~> A ~> GFix F := (*... ana_f_ ... *)
```

From this definition the universality property falls out:

```
Lemma ana_univ `{eA : setoid A} (h : Coalg F A) (f : A ~> GFix F)
: f =e ana h <-> f =e g_in \o fmap f \o h.
```

In words, for any `F`-coalgebra, if there is any other function `f` that is a `F`-coalgebra homomorphism then it must be the anamorphism on the same coalgebra.

3.3 Recursive Coalgebras and Hylomorphism

Hylomorphisms capture the concept of *divide-and-conquer* algorithms where the input is first destructured (*divide*) in smaller parts by means of a coalgebra which are computed recursively and then composed back together (*conquer*) by means of an algebra.

In general given an F-algebra and F-coalgebra, the hylomorphism is the unique solution (when it exists) to the equation

$$f = a \circ F\ f \circ c \tag{1}$$

As we stated earlier, a solution to this equation does not exist for an arbitrary algebra and coalgebra pair and, in fact, this definition cannot be accepted by Coq.

In order to find a solution we restrict ourselves to the so-called *recursive coalgebras* [4, 6]. An example of a recursive coalgebra is the `partition` function in quicksort which destructures a list into a pivot and two sublist and as long as the sublists are smaller the partitioning function still yields a unique solution to the recursion scheme.

We mechanise *recursive hylomorphisms* which are guaranteed to have a unique solution to the hylomorphism equation. These are hylomoprhisms where the coalgebra is *recursive*, i.e. coalgebras that terminate on all inputs. The following predicate captures that a coalgebra terminates on an input:

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

A recursive coalgebra of type `RCoalg F A` is a coalgebra `c` such that `forall x, RecF c x`. Recursive hylomorphisms are implemented in Coq recursively on the structure of the proof for the type (`RecF`) as follows:

```
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 h0 return (forall e : Pos (shape h0), RecF c (cont h0 e)) -> B
     with | MkApp s_x c_x => fun H => a (MkApp s_x (fun e => f (c_x e) (H e)))
     end (RecF_inv H).
```

We use `RecF_inv` to obtain the structurally smaller proof to use in the recursive calls. As we did with catamorphisms and anamorphisms, we prove that `hylo_def` is respectful, and use this proof to build the corresponding higher-order proper morphism:

```
  hylo : forall `{F : Cont Sh P} `{setoid A} `{setoid B},
    Alg F B ~> RCoalg F A ~> A ~> B
```

Finally, we show that recursive hylomorphisms are the unique solution to the hylomorphism equation.

```
Lemma hylo_univ (g : Alg F B) (h : RCoalg F A) (f : A ~> B)
    : f =e hylo g h <-> f =e g \o fmap f \o h.
```

Hylomorphisms fusion falls out from the universal property:

```
Lemma hylo_fusion_l (h1 : RCoalg F A) (g1 : Alg F B) (g2 : Alg F C)
  (f2 : B ~> C) (E2 : f2 \o g1 =e g2 \o fmap f2) : f2 \o hylo g1 h1 =e hylo g2 h1.
```

Using the hylo fusion law, we can prove the well-known *deforestation optimisation*. This is when two consecutive recursive computations, one that builds a data structure, and another one that consumes it, can be fused together into a single recursive definition. This, in turn, allows us to prove that a recursive hylomorphism is the composition of a catamorphism and a recursive anamorphism.

```
Lemma deforest (h1 : RCoalg F A) (g2 : Alg F C)
  (g1 : Alg F B) (h2 : RCoalg F B) (INV: h2 \o g1 =e id)
  : hylo g2 h2 \o hylo g1 h1 =e hylo g2 h1.
```

3.3.1 On the subtype of finite elements

In this development we have defined recursive anamorphisms on inductive data types. We might have as well defined them on the subtype of finite elements of coinductive data types using a predicate which states when an element of a coinductive data type is finite:

```
Inductive FinF : GFix F -> Prop :=
| FinF_fold (x : GFix F) : (forall e, FinF (cont (g_out x) e)) -> FinF x.
```

Now the subtype `{x : GFix F | FinF x}` of finite elements for `GFix F` is isomorphic its corresponding the inductive data type `LFix F`. This is easy to see. We first define a catamorphism `ccata_f_` from the subtype `{x : GFix F | FinF x}` of finitary elements of `GFix F` to any `F`-algebra.

```
Definition ccata_f_ `{eA : setoid A} (g : Alg F A)
  : forall x : GFix F, FinF x -> A := fix f x H :=
    let hx := g_out x in
      g (MkCont (shape hx) (fun e => f (cont hx e) (FinF_inv H e))).
```

We now prove this is isomorphic to the least fixed-point of the functor `F`. We take the catamorphism from the finite elements of `GFix F` to the inductive data type `LFix F` using the `F`-algebra `l_in`. Its inverse is the catamorphism on the restriction of `g_in` to the finite elements of `GFix`, which we denote by `lg_in`. The following lemmas prove the isomorphism:

```
Lemma cata_ccata `{setoid A} : cata lg_in \o ccata l_in =e id.
Lemma ccata_cata `{setoid A} : ccata l_in \o cata lg_in =e id.
```

The finite subtype of `GFix F` allows us to compose catamorphisms and anamorphisms, by using the above isomorphism. In our work, however, we use *recursive* anamorphisms, defined as `hylo l_in c` for a recursive coalgebra `c`, which compose easily with catamorphisms.

4 Extraction

We go in this section through a series of case studies of various recursive algorithms. We show how they can be encoded in our framework, how can we do program calculation techniques for optimisation, and how can they be extracted to idiomatic OCaml code. Our examples are the Quicksort and Mergesort algorithms (Section 4.1), dynammic programming and Knapsack (Section 4.2), and examples of the shortcut deforestation optimisation in our framework (Section 4.3).

4.1 Sorting Algorithms

Our first case study is divide-and-conquer sorting algorithms. Encoding them in our framework will require the use of recursive hylomorphisms and termination proofs. We complete the sorting algorithm examples by applying fusion optimisation.

Both mergesort and quicksort are divide-and-conquer algorithms that can be captured by the structure of an hylomorphisms. The structure of the recursion is that of a binary tree. For example, in the case of quicksort, a list is split into a pivot, the label of the node, and two sublists. We define the data functor of trees as follows:

```
Inductive ITreeF L N X := i_leaf (l : L) | i_node (n : N) (l r : X)
```

We define the functor as a container using the following shapes and positions:

```
Inductive Tshape L A := | Leaf (ELEM : L) | Node (ELEM : A).
Inductive Tpos := | Lbranch | Rbranch.
```

These define a container, `TreeF`, in a straightforward way, by making the positions of type `Tpos` only valid in `Node`. We define a series of definitions for tree container constructors and destructors:

```
Definition a_out {L A X : Type} : App (TreeF L A) X ~> ITreeF L A X.
Notation a_leaf x := (MkCont (Leaf _ x) (@dom_leaf _ _ _ x)).
Notation a_node x l r := (MkCont (Node _ x)
  (fun p => match val p with | Lbranch => l | Rbranch => r end)).
```

The container for the Quicksort hylomorphism is `TreeF unit int`, with the following algebra and coalgebra.

```
  Definition merge : App (TreeF unit int) (list int) ~> list int.
|{ x : (App (TreeF unit int) (list int)) ~> (
         match x with
         | MkCont sx kx =>
             match sx return (Container.Pos sx -> _) -> _ with
             | Leaf _ _ => fun _ => nil
             | Node _ h => fun k => List.app (k (posL h)) (h :: k (posR h))
             end kx
         end
  )}|.
Defined.

Definition c_split : Coalg (TreeF unit int) (list int).
|{ x ~> 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}|.
Defined.
```

We prove that the coalgebra `c_split` is recursive by showing that it respects the "less-than" relation on the length of the lists. The code that we extract for `hylo merge c_split` is the following:

```
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))
```

Note that Coq's code extraction is unable to inline `x0`, but the resulting code is similar to a hand-written `qsort`. The mergesort algorithm can be defined analogously and can be found in the formalisation[3].

[3] https://github.com/dcastrop/coq-hylomorphisms

4.1.1 Fusing a divide-and-conquer computation

As an example of how can we use program calculation techniques in our framework, we show how another traversal can be fused into the divide-and-conquer algorithm using the laws of hylomorphisms. Suppose that we map a function to the result of sorting the list. We can use our framework to fuse both computations. In particular, consider the following definition:

```
Definition qsort_times_two := Lmap times_two \o hylo merge c_split.
```

Here, `Lmap times_two` is a list map function defined as a hylomorphism, and `times_two` multiplies every element of the list by two. We can use Coq's generalised rewriting, and `hylo_fusion_l` to fuse `times_two` into the RHS hylomorphism in `qsort_times_two`. After applying hylo fusion and the necessary rewrites, the hylomorphism that we extract is `hylo (merge \o natural times_two) c_split`. In this definition, `natural` defines a natural transformation by applying `times_two` to the shapes, and `times_two` multiplies every pivot in the Quicksort tree by two. Our formalisation contains a proof that `natural` is indeed a natural transformation, which relies on the fact that it preserves the structure of the shapes and, therefore, the validity of the positions. The extracted OCaml code is a single recursive traversal:

```
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))
```

4.2 Knapsack

We focus now on the formalisation and extraction of *dynamorphisms* for dynamic programming, by using their encoding as a hylomorphism. We use the knapsack example from [16]. Dynamorphisms build a memoisation table that stores intermediate results, alongside the current computation. The algebra used to define a dynamorphism can access this memoisation table to speed up computation. First, we define memoisation tables in terms of containers.

```
Definition MemoShape : Type := A * Sg.
Definition MemoPos := Pg.
Instance Memo : Cont MemoShape MemoPos := { valid := valid \o pair (snd \o fst) snd }.

Definition Table := LFix Memo.
```

Memoisation tables are the least fixed point of the container defined by shapes `A * Sg` and positions `Pg`, given a a container `G : Cont Sg Pg`. We define a function to insert elements into the memo tables:

```
  Definition Cons : A * App G Table ~> App Memo Table := (* *)
```

And two functions to inspect the head of a memo table, and remove an element from it:

```
Definition headT : Table ~> A := (* *)
Definition tailT : Table ~> App G Table := (* *)
```

These tables map "paths" in the least fixed point of `Memo` to elements of type `A`. For example, if `G` is a list-generating functor, these paths will be natural numbers. Using these definitions, a dynamorphism is defined as follows:

```
Definition dyna (a : App G Table ~> A) (c : RCoalg G B) : B ~> A
:= headT \o hylo (l_in \o Cons \o pair a id) c.
```

Note how, instead of an algebra `App G A ~> A`, the algebra takes a memo table. The definition of the algebra can use this table to lookup elements, instead of triggering a further recursive call. Elements are inserted into the memoisation table by the use of `Cons` to the result of applying the algebra. The algebra for the dynamorphism looks up the previously computed elements to produce the result, thus saving the corresponding recursive calls:

```
Fixpoint memo_knap table wvs :=
  match wvs with | nil => nil | h :: t =>
      match lookupT (Nat.pred (fst h)) table with
      | Some u => (u + snd h)%sint63 :: memo_knapsack table t
      | None => memo_knapsack table t
      end
  end.

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 kx => 0%sint63
  | inr tt => fun kx => let tbl := kx next_elem in max_int 0 (memo_knap tbl wvs)
  end kx end.
Definition knapsackA wvs : App NatF (Table NatF int) ~> int :=
  (* [knapsack_alg wvs] as a respecful morphism *)
```

The hylomorphism for knapsack is as follows, where `out_nat` is the recursive coalgebra for `nat`.

```
Example knapsack wvs : Ext (dyna (knapsackA wvs) out_nat).
```

Coq's code extraction mechanism is unable to inline several definitions in this case. We have manually inlined the extracted code for simplicity. The reader can check in our artefact that the extracted code can be trivially inlined to produce the following:

```
let knapsack wvs x = let (y, _) =
    (let rec f x0 =
      if x0=0 then Uint63.of_int (0)
      else let fn := f (x0-1) in { lFix_out = {
           shape = (max_int (Uint63.of_int (0)) (memo_knap fn wvs), sx);
           cont = fun _ -> fn } }
     in f x).lFix_out.shape in y
```

Note how the recursive calls of `f` build the memoisation table, and how this memoisation table is used to compute the intermediate results in `memo_knap`, which is finally discarded to produce the final result.

4.3 Shortcut Deforestation

The final case study we consider is shortcut deforestation on lists. Shortcut deforestation can be expressed succintly in terms of hylomorphisms and their laws [27]. In particular, given a function:

```
  s  : forall A. (App F A -> A) -> (App F A -> A)
```

We can conclude, by parametricity, that

```
hylo a l_out \o hylo (sigma l_in) c =e hylo (s a) c
```

This is generally known as the *acid rain* theorem. Unfortunately, this is **not** provable in Coq if we want to remain axiom-free, since we would need to add the necessary parametricity axiom [19]. However, we prove a specific version of this theorem for the list generating container (i.e. the container whose least fixed point is a list), and use it to encode the example from Takano and Meijer [27]:

```
Definition sf1 (f : A ~> B) ys : Ext (length \o Lmap f \o append ys).
```

Here, we are defining function `sf1` as the composition of `length`, `Lmap f` and `append ys`. Functions `length` and `Lmap` are catamorphisms. Function `append ys` is also a catamorphism that appends `ys` to an input list. It is defined by applying an algebra to every cons node of a list, and applying a catamorphism with the input algebra to `ys` in the nil case:

```
Definition tau (l : list A) (a : Alg (ListF A) B) : App (ListF A) B -> B :=
  fun x => match x with | MkCont sx kx => match sx with
  | s_nil => fun _ => (hylo a ilist_coalg) l
  | s_cons h => fun kx => a (MkCont (s_cons h) kx)
  end kx end.
Definition append (l : list A) := hylo (tau l l_in) ilist_coalg.
```

Here, `ilist_coalg` is a recursive coalgebra from Coq lists to the `ListF` container. We apply the hylo fusion law repeatedly, unfold definitions, and simplify in our specification for `sf1`:

```
Definition sf1 (f : A ~> B) ys : Ext (length \o Lmap f \o append ys).
  rewrite hylo_map_fusion, <- acid_rain. simpl; reflexivity.
Defined.
```

From this, we extract the following OCaml code:

```
let rec sf1 f ys = function | [] -> let rec f0 = function
                                     | [] -> (Uint63.of_int (0))
                                     | _ :: t -> add (Uint63.of_int (1)) (f0 t)
                                   in f0 ys
                            | _ :: t -> add (Uint63.of_int (1)) (sf1 f ys t)
```

We then prove that the length function fuses with the naive quadratic *reverse* function:

```
Definition sf2 : Ext (length \o reverse).
  calculate.  unfold length, reverse. rewrite hylo_fusion_l.
  2:{ (* Rewrite into the fused version *) }
  simpl; reflexivity.
Defined.
```

This code extracts to the optimised length function on the input list:

```
let rec ex2 = function | [] -> (Uint63.of_int (0))
                       | _ :: t -> add (Uint63.of_int (1)) (ex2 t)
```

5 Related Work

Encoding recursion schemes in Coq is not new. We compare our work with other encodings of program calculation techniques in Coq (Section 5.1), recursion schemes in Coq (Sections 5.1 and 5.2.1). Finally, we compare our encoding of recursion schemes to other forms of termination checking (Section 5.2, and sized types (Section 5.2.2), as a way to guarantee termination of nonstructural recursion.

5.1 Program Calculation

Within the domain of program optimization, program calculation serves as a well-established programming technique aimed at deriving efficient programs from their naive counterparts through systematic program transformation [13]. This area has been extensively explored over the years. Tesson et al. demonstrated the efficacy of leveraging Coq to establish an approach for implementing a robust system dedicated to verifying the correctness of program transformations for functions that manipulate lists [28]. Murata and Emoto went further and formalised recursion schemes in Coq [24]. Their development does not include hylomorphisms and dynamorphisms, and relies on the functional extensionality axiom, as well as further extensionality axioms for each coinductive datatype that they use. They do not discuss the extracted OCaml code from their formalisation. Larchey-Wendling and Monin encode recursion schemes in Coq, by formalising computational graphs of algorithms [20]. Their work does not focus on encoding generic recursion schemes, and proving their algebraic laws. Castro-Perez et al. [7] encode the laws of hylomorphisms as part of the type system of a functional language to calculate parallel programs from specifications. Their work focuses on parallelism, and they do not formalise their approach in a proof assistant, and the laws of hylomorphisms are axioms in their system.

5.2 Termination Checking

Termination and productivity are non-trivial properties of programs, and various methods have been proposed for checking that these properties hold. A common approach is guardedness checking [14], which is a *syntactic* check that definitions avoid the introduction of non-normalizable terms. This sort of check generally looks for a *structural decrease* of arguments in the recursive calls of a function. Coq uses such a check, and it works for many classic functional programming patterns (like `map` and `foldr`). However, some desirable definitions will not be accepted by such checks.

In particular, the problem of *nonstructural recursion* (including divide-and-conquer algorithms) is well-studied [5]. Certain functions that are not structurally recursive can be reformulated using a nonstandard approach to achieve structural recursion [3]. Take, for instance, division by iterated subtraction, which is inherently non-structurally recursive since it involves recursion based on the result of a subtraction. There is a nonstandard implementation of divivion found on Coq's standard library, which involves a four-argument function that effectively combines subtraction and division. Similarly, the mergesort in Coq's standard library uses an "explicit stack of pending merges" in order to avoid issues with nonstructural definitions. There is a major downside, however; as noted by Abreu et al., the result is "barely recognizable as a form of mergesort" [3].

5.2.1 Divide and Conquer Recursion

Abreu et al. [3] encode divide-and-conquer computations in Coq, using a recursion scheme in which termination is entirely enforced by its typing. This is a significant advance, since it avoids *completely* the need for termination proofs. Their work differs from ours in that they require the functional extensionality axiom, and the use of impredicative `Set`. The authors justify well the use of impredicative `Set` and its compatibility with the functional extensionality axiom. In contrast, our development remains entirely *axiom-free*. Another key difference with our approach is that they do not discuss what the resulting extracted code *looks like* (that is, whether the extracted OCaml code resembles the natural formulation of the recursive function). Through experiments, we found that their formalisation leads to

`Obj.magic`, and code with a complex structure that may be hard to understand or interface with other OCaml code. Due to the great benefit of entirely avoiding termination proofs, it would be interesting to extend their approach to improve code extraction.

5.2.2 Sized Types

Another approach to certifying termination of recursive functions is the use of *sized types*. Sized types were introduced by Hughes et al. as a way to track/verify various properties of recursive programs, including productivity and termination [18]. The core idea of sized types is that types express bounds on the sizes of recursive data structures. With this approach, algorithms are implemented in the standard functional way, following the divide-and-conquer pattern of splitting, recurring, and merging, with the addition of the data types being indexed by a static approximation of the relative *size*. An advantage of this approach is that it allows for code to be written in a natural way. There are costs, however. Programs must be written in a way that accounts for the handling of size indexes, and support for the size types must be added to the language.

This approach has been used to express nonstructurally recursive algorithms in Agda: Copello et al. used sized types in a straightforward formulation of mergesort [9]. Such an approach has also been used in MiniAgda [2]. A proposal exists to add sized types to Coq as well, though it has not yet been adopted [8].

6 Conclusions and Future Work

Hylomorphisms are a general recursion scheme that can encode any other recursion scheme, and that satisfy a number of algebraic laws that can be used to reason about program equivalences. To our knowledge, this is their first formalisation in Coq. This is partly due to the difficulty of dealing with termination, and reasoning about functional extensionality. In this work, we tackle these problems in a *fully axiom-free* way that targets the extraction of idiomatic OCaml code. This formalisation allows the use of program calculation techniques in Coq to derive formally optimised implementations from naive specifications. Furthermore, the rewritings that are applied to specifications are formal, machine-checked proofs that the resulting program is extensionally equal to the input specification.

Remaining axiom-free forces us to deal with the well-known *setoid hell*. As part of the future improvements, we will study how to mitigate this problem. At the moment, we use a short ad-hoc tactic that is able to automatically discharge many of these proofs in simple settings. We will study the more thorough and systematic use of proof automation for respectful morphisms. Generalised rewriting in proofs involving setoids tends to be quite slow, due to the large size of the terms that need to be rewritten. Sometimes, this size is hidden in implicit arguments and coercions. We will study alternative formulations to try to improve the performance of the rewriting tactics (e.g. canonical structures). Currently, Coq is unable to inline a number of trivially inlineable definitions. We will study alternative definitions, or extensions to Coq's code extraction mechanisms to force the full inlining of all container code that is used in hylomorphisms. Finally, proving termination still remains a hurdle. In our framework this reduces to proving that the anamorphism terminates in all inputs, and we provide a convenient connection to well-founded recursion. Furthermore, recursive coalgebras compose with natural transformations, which allows the reuse of a number of core recursive coalgebras. A possible interesting future line of work is the use of the approach by Abreu et al. [3] in combination with ours to improve code extraction from divide-and-conquer computations whose termination does not require an external proof.

References

1 Michael Gordon Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theor. Comput. Sci.*, 342(1):3–27, 2005. URL: https://doi.org/10.1016/j.tcs.2005.06.002, doi:10.1016/J.TCS.2005.06.002.

2 Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In Dale Miller and Zoltán Ésik, editors, *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012*, volume 77 of *EPTCS*, pages 1–11, 2012. doi:10.4204/EPTCS.77.1.

3 Pedro Abreu, Benjamin Delaware, Alex Hubers, Christa Jenkins, J. Garrett Morris, and Aaron Stump. A type-based approach to divide-and-conquer recursion in coq. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. doi:10.1145/3571196.

4 Jirí Adámek, Stefan Milius, and Lawrence S. Moss. On well-founded and recursive coalgebras. *CoRR*, abs/1910.09401, 2019. URL: http://arxiv.org/abs/1910.09401, arXiv:1910.09401.

5 Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers – an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, 2016. doi:10.1017/S0960129514000115.

6 Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Recursive coalgebras from comonads. In Jirí Adámek and Stefan Milius, editors, *Proceedings of the Workshop on Coalgebraic Methods in Computer Science, CMCS 2004, Barcelona, Spain, March 27-29, 2004*, volume 106 of *Electronic Notes in Theoretical Computer Science*, pages 43–61. Elsevier, 2004. URL: https://doi.org/10.1016/j.entcs.2004.02.034, doi:10.1016/J.ENTCS.2004.02.034.

7 David Castro-Perez, Kevin Hammond, and Susmit Sarkar. Farms, pipes, streams and reforestation: reasoning about structured parallel processes using types and hylomorphisms. In *Proc. of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, page 4–17. ACM, 2016. doi:10.1145/2951913.2951920.

8 Jonathan Chan, Yufeng Li, and William J. Bowman. Is sized typing for coq practical? *J. Funct. Program.*, 33:e1, 2023. doi:10.1017/S0956796822000120.

9 Ernesto Copello, Alvaro Tasistro, and Bruno Bianchi. Case of (quite) painless dependently typed programming: Fully certified merge sort in agda. In Fernando Magno Quintão Pereira, editor, *Programming Languages - 18th Brazilian Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings*, volume 8771 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2014. doi:10.1007/978-3-319-11863-5_5.

10 Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. Fast and loose reasoning is morally correct. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 206–217. ACM, 2006. doi:10.1145/1111037.1111056.

11 Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. Verified Extraction from Coq to OCaml. working paper or preprint, November 2023. URL: https://inria.hal.science/hal-04329663.

12 Jeremy Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4):657–665, 1996. Earlier version appeared in C. B. Jay, editor, *Computing: The Australian Theory Seminar*, Sydney, December 1994, p. 62–69. URL: http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/thirdht.ps.gz.

13 Jeremy Gibbons. The school of squiggol. In *Formal Methods. FM 2019 International Workshops*, pages 35–53, Cham, 2020. Springer International Publishing.

14 Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *Selected Papers from the International Workshop on Types for Proofs and Programs*, TYPES '94, page 39–59, Berlin, Heidelberg, 1994. Springer-Verlag.

15 Georges Gonthier and Roux Le. An ssreflect tutorial. 01 2009.

16 Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. Conjugate hylomorphisms - or: The mother of all structured recursion schemes. In Sriram K. Rajamani and David Walker, editors, *Proceedings*

of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, pages 527–538. ACM, 2015. doi:10.1145/2676726.2676989.

17 Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In Robert Harper and Richard L. Wexelblat, editors, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, pages 73–82. ACM, 1996. doi:10.1145/232627.232637.

18 John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 410–423. ACM Press, 1996. doi:10.1145/237721.240882.

19 Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic (CSL'12) - 26th International Workshop/21st Annual Conference of the EACSL*, volume 16 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 381–395, Dagstuhl, Germany, 2012. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2012.381, doi:10.4230/LIPIcs.CSL.2012.381.

20 Dominique Larchey-Wendling and Jean-François Monin. The braga method: Extracting certified algorithms from complex recursive schemes in coq. In *PROOF AND COMPUTATION II: From Proof Theory and Univalent Mathematics to Program Extraction and Verification*, pages 305–386. World Scientific, 2022.

21 Dominique Larchey-Wendling and Jean-François Monin. Proof pearl: Faithful computation and extraction of μ-recursive algorithms in coq. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.ITP.2023.21, doi:10.4230/LIPICS.ITP.2023.21.

22 Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings*, Lecture Notes in Computer Science. Springer, 1991. doi:10.1007/3540543961_7.

23 Marino Miculan and Marco Paviotti. Synthesis of distributed mobile programs using monadic types in coq. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving - Third International Conference, ITP 2012, Princeton, NJ, USA, August 13-15, 2012. Proceedings*, volume 7406 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2012. doi:10.1007/978-3-642-32347-8_13.

24 Kosuke Murata and Kento Emoto. Recursion schemes in coq. In Anthony Widjaja Lin, editor, *Programming Languages and Systems*, pages 202–221, Cham, 2019. Springer International Publishing.

25 Kosuke Ono, Yoichi Hirai, Yoshinori Tanabe, Natsuko Noda, and Masami Hagiya. Using coq in specification and program extraction of hadoop mapreduce applications. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, volume 7041 of *Lecture Notes in Computer Science*, pages 350–365. Springer, 2011. doi:10.1007/978-3-642-24690-6_24.

26 Kazuhiko Sakaguchi. Program extraction for mutable arrays. *Science of Computer Programming*, 191:102372, 2020. URL: https://www.sciencedirect.com/science/article/pii/S0167642319301650, doi:10.1016/j.scico.2019.102372.

27 Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer*

Architecture, FPCA '95, page 306–313, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224164.224221.

28 Julien Tesson, Hideki Hashimoto, Zhenjiang Hu, Frédéric Loulergue, and Masato Takeichi. Program calculation in coq. In Michael Johnson and Dusko Pavlovic, editors, *Algebraic Methodology and Software Technology*, pages 163–179, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.