Program Optimisations via Hylomorphisms for Extraction of Executable Code

David Castro Perez
University Of Kent, Canterbury, CT2 7NZ, United Kingdom

Marco Paviotti
University Of Kent, Canterbury, CT2 7NZ, United Kingdom

Michael Vollmer
University Of Kent, Canterbury, CT2 7NZ, United Kingdom

Abstract

Generic programming with recursion schemes provides a powerful abstraction for structuring recursion and provides a rigorous reasoning principle for program optimisations which have been successfully applied to compilers for functional languages. Formalising recursion schemes in a type theory offers additional termination guarantees, but it often requires compromising on performance, requiring the assumption of additional axioms, and/or introducing unsafe casts into extracted code.

This paper presents the first Rocq formalisation of *hylomorphisms* allowing for the mechanisation of all recognised recursive algorithms. The key contribution of this paper is that this formalisation is fully *axiom-free*, and it allows the extraction of safe, idiomatic functional 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 functional code for the programs formalised in our framework is efficient, humanly readable, and runnable. Furthermore, we show the use of the machine-checked proofs of the laws of hylomorphisms to do program optimisations.

2012 ACM Subject Classification Theory of computation → Functional constructs; Theory of computation → Type theory; Theory of computation → Program verification

Keywords and phrases hylomorphisms, program calculation, divide and conquer, fusion

Digital Object Identifier 10.4230/LIPIcs.ITP.2025.32

Supplementary Material *Software (Source Code)*: https://github.com/dcastrop/coq-hylomorphisms

Funding *David Castro Perez*: EP/Y00339X/1, EP/T014512/1

1 Introduction

Over the years, extensive research has been conducted on program calculation techniques, a well-established approach for deriving efficient programs from simpler specifications through systematic program transformation [12, 4]. A key aspect of this approach is the use of structured recursion schemes, which serve as powerful abstractions that capture common patterns of recursion. By structuring computation in this way, programs benefit from well-established *algebraic properties* that provide a solid foundation for reasoning about program equivalences, transformations, and optimisations – such as *fusion* laws or semi-automatic parallelisations [28, 11, 20, 7]. In the context of *program calculation*, these algebraic properties allow programmers to describe code using simple, inefficient specifications within an *algebra of programming* [4]. They can then apply algebraic laws to systematically *calculate* more efficient versions of the same algorithms.

Suppose, for example, that we want to write a program that sorts a list of integers and multiplies them by 2 at the same time. In OCaml, we may write this function directly:

```
let rec sort_times_two = function
  | [] -> []
  | h :: t -> let (l, r) = partition (fun x -> x < h) t in
  sort_times_two l @ (h * 2) :: sort_times_two r
```

This program is the same as composing a *quicksort* OCaml implementation, with the function `map (fun x -> x * 2)` which is a known program optimisation called *fusion*. A lot of program calculations have this pattern: start from a simple specification, e.g. $\mathsf{map}\ (\lambda x.\ x \times 2) \circ \mathsf{sort}$, and use program equivalences and algebraic laws to rewrite it to an optimised version. We will revisit a similar example in Section 4.1.

This program calculation stems from the more general theory of hylomorphisms. A hylomorphism is a structured recursion scheme representing the idea of a divide-and-conquer algorithm where, first, the problem is split into smaller sub-problems (divide), then sub-solutions are computed recursively, and, finally, these sub-solutions are put together to compute the solution to the original problem (conquer). This general pattern of recursion induces a reasoning principle, called the *fusion law*, which has been exemplified in the previous paragraph. By leveraging this principle, we can systematically transform recursive definitions into more efficient versions while preserving correctness.

Hylomorphisms are highly versatile, non-structural recursive algorithms that are capable of implementing structural recursion as well as other known recursion schemes like mutual recursion, accumulators, primitive recursion, and course-of-values iteration (dynamic programming) [15]. However, this versatility comes with a trade-off: because hylomorphisms are not inherently structurally recursive, special attention is required when encoding them in a type theory. Specifically, the "divide" phase of the algorithm must be recursive, meaning the input needs to be broken down into smaller parts, which do not necessarily align with the structure of the input data. On the other hand, a key benefit is the reusability of termination proofs: once the correctness of the divide phase is established, the same proof can be reused for any other conquer phase we choose. Moreover, in the case of structural recursion, the divide phase naturally follows the structure of the data, making it correct by definition and eliminating the need for additional proofs.

Implementations of recursion schemes generally focus on non-terminating languages (e.g. Haskell) and they would benefit from a type-theoretical formalisation which leverages the underlying prover's logic to ensure correctness of definitions. While existing mechanisations do exist they either do not cover a large subset of the recursion schemes or are not suitable for program calculation or code extraction (see Section 5, Related Work).

In this work, we mechanise hylomorphisms, reaping the benefits of their generality and algebraic properties. Our approach enables users to write high-level specifications for their programs, reason about program optimisations, and ultimately extract human-readable, executable functional code. The extracted code is free from unsafe casts, a priority in our design. We focus strongly on avoiding unsafe casts because, even if the extracted code has been verified, simple interoperations with them can lead to incorrect behaviour or even segmentation faults [10] and, moreover, it invalidates the fast-and-loose principle [8].

To preserve the generality of hylomorphisms, we encode data structures using polynomial functors. To ensure that the extracted code is idiomatic and does not contain unsafe casts, we avoid using indexed types in the definition of the recursion schemes, and make sure that the definitions can be inlined by Rocq's extraction mechanism. To strengthen the results of this paper, we avoid all axioms.

We list the contributions of this paper. First, in Section 2, we provide the theoretical background on recursion schemes and hylomorphisms. We then provide a Rocq formalisation

of *hylomorphisms* (Section 3) that (1) is *fully axiom-free*; (2) allows the extraction of idiomatic and efficient code; and (3) can use regular Rocq equalities to do program calculation, derive correct implementations, and apply optimisations. In Section 4, we apply this framework to formalise practical examples of divide-and-conquer, dynamic programming, and mutual recursion algorithms. Additionally, we verify the short-cut fusion optimisation and present the extracted optimised code.

2 Recursion Schemes

The structure of data is very similar to the structure of an algorithm which processes that data. This relationship manifests in the form of structured recursion schemes, which are widely used in functional languages such Haskell. Canonical examples are folds (catamorphisms), which consume data, and unfolds (anamorphisms), which produce it. While some implementations of recursion schemes like foldr/unfoldr in Haskell are specific to a particular data structure, in this case Lists, we can generalise these ideas further to account for generic (co)inductive data types and generic algorithms operating on them.

Furthermore, folds and unfolds can be shown to capture a wide range of recursion schemes, such as primitive recursion, mutual recursion, dynamic programming algorithms, polymorphic recursion, and recursion with accumulators. However, for divide-and-conquer algorithms, folds need to be generalised to hylomorphisms, which provide the ultimate basic building block for any other recursion scheme. To do this, we look at recursion schemes from the point of view of category theory.

2.1 Elements of Category Theory

A *category* is a collection of objects A, B, C, denoted by $\mathrm{Obj}(\mathcal{C})$ and a collection of arrows f, g, h between these objects, denoted by $\mathrm{Arr}(\mathcal{C})$, such that there always exists an identity arrow $id_A : A \to A$ for each object A and for two arrows $A \xrightarrow{f} B$ and $B \xrightarrow{g} C$ there always exists an arrow $A \xrightarrow{g \circ f} C$ obeying the identity and associativity law. We denote $\mathrm{Hom}_{\mathcal{C}}(A, B)$ the set of arrows from A to B and we use the letters $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$ ... for categories[1]. The initial object (when it exists), denoted by 0, is the object such that for any other object A there is a unique arrow $0 \xrightarrow{!} A$. Dually, the terminal object (when it exists), denoted by 1, is the object such that for any other object A there is a unique arrow $A \xrightarrow{!} 1$. As a result of the uniqueness properties, initial and terminal objects are unique up-to isomorphism.

For example, the category of sets, denoted by **Set**, is the category where objects are sets and arrows are functions between sets. The initial object 0 in **Set** is the empty set $\emptyset$ and the terminal object 1 is any singleton set. The reader who is not accustomed with category theory can assume for simplicity that types may be viewed as sets and programs as functions between sets, giving the intuition that the category **Set** can also be viewed as the category of (simple) types and programs between them.

A *functor* $F : \mathcal{C} \to \mathcal{D}$ is a map between categories mapping both objects and arrows from one category to another. Hence a functor has two components, one which maps objects into objects $F : \mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}(\mathcal{D})$ and one which maps arrows into arrows $F : \mathrm{Hom}_{\mathcal{C}}(A, B) \to$

[1] For presentation purposes we shall not deal with size issues and assume all the categories are locally small.

$\mathrm{Hom}_{\mathcal{D}}(FA, FB)$ such that identity and composition of arrows are preserved:

$$F(id_A) = id_{FA} \qquad\qquad F(g \circ f) = F(g) \circ F(f)$$

This latter component is also called the *functorial action* and, if types are **Set**s, this can be thought of as the fmap higher-order function in functional programming.

In **Set**, we can define the set of lists

$$\mathsf{List}(A) \cong 1 + A \times \mathsf{List}(A)$$

as the set inductively generated by the constructors $\mathsf{nil} : 1 \to \mathsf{List}(A)$ and $\mathsf{cons} : A \times \mathsf{List}(A) \to \mathsf{List}(A)$. The $\mathsf{List}(-)$ type is a *functor* **Set** $\to$ **Set**, in particular, an *endofunctor*, mapping objects and arrows in **Set** to **Set** itself. Its functorial action $\mathsf{List}(f) : \mathsf{List}(A) \to \mathsf{List}(B)$ is given by $\mathsf{List}(f)(\mathsf{nil}(*)) = \mathsf{nil}(*)$ and $\mathsf{List}(f)(\mathsf{cons}(a, xs)) = \mathsf{cons}(f(a), \mathsf{List}(f)(xs))$. Notice that the definition $\mathsf{List}(f)$ is well-defined as it recursively calls on a smaller argument. Similarly, the set of streams $\mathsf{Str}(A) \cong A \times \mathsf{Str}(A)$ is the greatest set generated by the constructor $\mathsf{cons} : A \times \mathsf{Str}(A) \to \mathsf{Str}(A)$. The maps $\mathsf{head} : \mathsf{Str}(A) \to A$ and $\mathsf{tail} : \mathsf{Str}(A) \to \mathsf{Str}(A)$ can be easily constructed from the isomorphism.

Given two functors $F, G : \mathcal{C} \to \mathcal{D}$, a *natural transformation* is is a family of morphisms $\phi_X : FX \to GX$ indexed by the objects $X \in \mathrm{Obj}(\mathcal{C})$ and such that it is *natural* in X, that is $G(f) \cdot \phi_X = \phi_Y \cdot F(f)$, for all morphisms $f : X \to Y$. Intuitively, a natural transformation is akin to a polymorphic function transforming the structure of a functor into the structure of another functor without assuming what is the type of data contained in them. For example, a program $\phi_X : \mathtt{List}\ X \to \mathtt{Maybe} X$, which returns nothing if the list is empty and the head of the list otherwise, does not need to know the content of the list to perform its task, in other words, it can work for all types X *uniformly*.

2.2 Algebras and Catamorphisms

For an endofunctor $F : \mathcal{C} \to \mathcal{C}$ an *F-algebra* is a pair (X, a_X) where X is an object of the category called the *carrier* of the algebra and a_X is an arrow of type $FX \to X$ called the *structure map*. The category of F-algebras, denoted by $F\text{-Alg}(\mathcal{C})$, is the category where objects are F-algebras and arrows $f : (X, a_X) \to (Y, a_Y)$ are *F-algebra homomorphisms*. These are arrows $f : X \to Y$ in the underlying category $\mathcal{C}$ such that they respect the structure of the algebra, that is $f \circ a_X = a_Y \circ F(f)$. The initial object in this category is called the *initial F-algebra*, that is the F-algebra which has a unique F-algebra homomorphism into any other F-algebra. By Lambek's lemma if F has an initial F-algebra then this is the *least fixed-point* for the functor F which we denote by $(\mu F, \mathsf{in})$ where $\mathsf{in} : F\mu F \to \mu F$ is the F-algebra witnessing the isomorphism $F\mu F \cong \mu F$, furthermore $\mathsf{in}^\circ : \mu F \to F\mu F$ is the inverse of in. The uniqueness property of initial F-algebras states that for any other F-algebra (X, a_X) there exists a unique F-algebra homomorphism, denoted by $(\!|\, a_X \,|\!) : (\mu F, \mathsf{in}) \to (X, a_X)$ and pronounced "*catamorphism*" or "*fold*". This is formally stated as follows:

$$f = (\!|\, a_X \,|\!) \iff f = a_X \circ F(f) \circ \mathsf{in}^\circ \tag{1}$$

As a result of the uniqueness property we can derive the fusion law. For all F-algebra homomorphisms $f : (X, a_X) \to (Y, a_Y)$ we have

$$f \circ (\!|\, a_X \,|\!) = (\!|\, a_Y \,|\!) \tag{2}$$

which means that the composition of a program f with a catamorphism recursing once over the data structure is the same as performing that recursion once using the algebra a_Y instead of a_X. This is a useful result for program optimisation as we shall see.

For example, for a set A we define the functor $F : \mathbf{Set} \to \mathbf{Set}$ mapping $X \mapsto 1 + A \times X$. An F-algebra is a set B together with a structure map $[base, step] : 1 + A \times B \to B$. The initial F-algebra is clearly the set of lists $\mathsf{List}(A)$ and the catamorphism associated with the type of lists is the unique arrow which recursively translates the initial algebra $[\mathsf{nil}, \mathsf{cons}]$ into the algebra $[base, step]$ while turning the operation nil into $base$ and cons into $step$. In functional programming this is commonly referred to as $\mathsf{foldr} : (1 \to B) \to (A \times B \to B) \to \mathsf{List}(A) \to B$. We can in fact set $\mathsf{foldr}\ base\ step = (\!|\, [base, step] \,|\!)$.

2.3 Coalgebras and Anamorphisms

The dual of an algebra is a coalgebra. For an endofunctor $B : \mathcal{C} \to \mathcal{C}$, a *$B$-coalgebra* is a pair (X, c_X) where $X \in \mathrm{Obj}(\mathcal{C})$ is the carrier of the coalgebra and $c_X : X \to BX$ is a morphism.

For example, for a set of states X and a finite set of labels L we can define a labelled transition system (LTS) [31] on X as a function $X \to BX$ implementing the *transition system* with $BX = L \times X$. In particular, for a state $x_1 \in X$, $c(x_1)$ returns a pair (l, x_2) where $l \in L$ is the observable action and $x_2 \in X$ is the next state.

The category of B-coalgebras, denoted B-CoAlg, is the category where objects are B-coalgebras and morphisms $f : (X, c_X) \to (Y, c_Y)$ are B-coalgebra homomorphisms $f : A \to B$, that is $c_Y \circ f = F(f) \circ c_X$. The terminal object in this category is called the terminal, or final, B-coalgebra. The carrier of this coalgebra corresponds to the greatest fixed-point for the functor B, denoted by $(\nu B, \mathsf{out})$ with out being the final B-coalgebra witnessing the isomorphism and $\mathsf{out}^\circ : B\nu B \to \nu B$ being its inverse.

For example, the terminal coalgebra for the functor $BX = A \times X$ is the set of infinite streams over the set A, that is the greatest solution to the equation $\mathsf{Str}(A) \cong A \times \mathsf{Str}(A)$. The uniqueness property of terminal B-coalgebras states that for any other B-coalgebra (X, c_X) there exists a unique B-coalgebra homomorphism into the terminal coalgebra $(\nu B, \mathsf{out})$ which is denoted by $[\![c_X]\!]$ and pronounced "*anamorphism*" or "*unfold*". We spell out the uniqueness property of unfolds:

$$f = [\![c_X]\!] \iff f = \mathsf{out}^\circ \circ B(f) \circ c_X \tag{3}$$

From the uniqueness property we can derive the fusion law for unfolds:

$$[\![c_Y]\!] \circ f = [\![c_X]\!] \tag{4}$$

for all B-coalgebra homomorphisms $f : (X, c_X) \to (Y, c_Y)$.

2.4 Recursive Coalgebras and Hylomorphism

Recursion schemes provide an abstract way to consume and generate data capturing *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.

Let (A, a) be an F-algebra and (C, c) be an F-coalgebra. An arrow $C \to A$ is an *hylomorphism*, written $h : (C, c) \rightarrowtail (A, a)$ if it satisfies

$$h = a \circ F(h) \circ c \tag{5}$$

A solution to this equation does not exist for an arbitrary algebra and coalgebra pair and, in fact, a recursive function definition like this is not accepted by Rocq.

A coalgebra (C, c) is *recursive* if for every algebra (A, a) there is a *unique* hylo $(C, c) \rightarrowtail (A, a)$ [6, 3]. We denote these type of hylos by $(\!|\, c \to a \,|\!)$.

An example of a recursive coalgebra is the `partition` function used in quicksort, which has the type List $A \to B(\text{List } A)$, where the functor B is defined as $BX = 1 + X \times A \times X$. This function deconstructs a list by selecting a pivot element of type A and splitting the remaining elements into two sublists.

Notice that the initial algebra for lists corresponds to the functor $FX = 1 + A \times X$ and its associated algebra map (or its inverse, viewed as a coalgebra), in° has the type List $A \to F(\text{List } A)$. Importantly, `partition` is not this map, which means that it does not arise from the standard initial algebra structure, and thus catamorphisms here cannot be used to define this recursive function.

Nevertheless, since `partition` always produces sublists that are strictly smaller than the input list, it still supports a well-founded recursion scheme, ensuring the existence of a unique solution. The uniqueness property of the hylomorphisms yields the following fusion laws:

$$f \circ (\!|\, c \to a \,|\!) = (\!|\, c \to a' \,|\!) \quad \Longleftarrow \quad f \circ a = a' \circ F(f) \tag{6}$$

$$(\!|\, c \to a \,|\!) \circ f = (\!|\, c' \to a \,|\!) \quad \Longleftarrow \quad c \circ f = F(f) \circ c' \tag{7}$$

Using the hylomorphism fusion laws, we can prove the well-known *deforestation optimisation* [30], also known as the *composition law* [14]. 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.

Recursive Anamorphisms

Anamorphisms applied to recursive coalgebras specialise to hylomorphisms into an inductive data type in the following way. A recursive coalgebra can be applied only finitely many times, therefore when this is applied to an anamorphism the only possible types it can produce from the seed are the finite ones. We call this special kind of recursion scheme *recursive anamorphism*. We can show that recursive anamorphisms of type $X \to \nu F$ can also be given the type $X \to \mu F$. Moreover, these anamorphisms are exactly hylomorphisms on the recursive F-coalgebra and the algebra in for the inductive data type μF. This fact falls out from the uniqueness property of the hylomorphism and the fact that recursive anamorphisms satisfy the same equation.

2.5 Polynomial Functors

Recursion schemes are formulated using generic functors that encapsulate the structure of the recursive operator. However, since not all functors have suitable fixed points, it is necessary to restrict our focus to *strictly positive functors*.

For example, in the category **Set**, consider the polynomial functor corresponding to

$$F(X) = A + B \times X + C \times X^2.$$

We can model this functor by using a *container* [1]. This is a set S of *shapes*, for example, $S = A + B + C$ for the example above, together with a function $P(s)$ denoting the exponent of X for each shape. Then a container extension is defined as the sum of all maps $P(s) \to X$

$$F(X) = \sum_{s \in S} (P(s) \to X)$$

The functorial action of F is given by post-composition, mapping an element (s, g) in $F(A)$ to $(s, f \circ g)$ in $F(B)$, for $f : A \to B$.

3 Formalising Recursion Schemes

We now consider the formalisation of hylomorphisms in Rocq. We first formalise container functors as a tool to represent polynomial functors (Section 3.1). Then we formalise algebras for container extensions (Section 3.2). In Section 3.3 we formalise coalgebras and put together these notions to formalise recursive coalgebras and hylomorphisms (Section 3.4).

3.1 Mechanising Extractable Container Functors

A direct encoding of containers as presented in Section 2 requires the use of the functional extensionality axiom, and axiom K to deal with equalities of objects of type $P(s) \to X$. Axiom K states that a predicate on equality proofs holds for all such proofs if it holds for reflexivity. This would be needed to reason about the equality of any two $f : P(s_1) \to X$ and $g : P(s_2) \to X$, when $s_1 = s_2$. In Rocq, if we know $(s_1, f) = (s_2, g)$, we cannot extract a proof that $f = g$ unless we assume axiom K, or the equality on the type of s_1 and s_2 is decidable. Furthermore, upon extraction Rocq will need to insert unsafe casts for anything of type $P(s)$, because OCaml requires the input value to have a type that is identical to the function parameter, and in the general case this cannot be done in OCaml. To avoid these problems, we use *setoids*, and encode families of positions using *decidable validity predicates*.

In our formalisation, we require that every type is a setoid, where `x =e y` denotes setoid equality and that all morphisms need to be *proper* morphisms that respect setoid equalities. We use special notation `A ~> B` for proper morphisms and we add an implicit coercion from `A ~> B` to `A -> B`. We also provide tactics to automatically discharge proofs that morphisms are proper, for some simple cases.

We define containers in terms of shapes, *base* positions (`APos : Set`), and *decidable validity predicates* that specify when a base position is valid in a shape. Since validity predicates are in `Prop`, they will be erased during extraction, and since they are decidable, they do not require axiom K to deal with heterogeneous equalities. Container extensions are defined as

```
Record App F X := {shape : Shape F; cont : {p : APos F | valid s p = true} -> X}.
```

We define the setoid equality of two container extensions `x y : App F X` as

```
shape x =e shape y /\ (forall p p', projT1 p = projT1 p' -> cont x p =e cont y p')
```

3.2 Algebras and Catamorphisms for Containers

Recall that an algebra is a set A together with a morphism that defines the operations of the algebra $F\ A \to A$. Given a type `A` and a container `F`, an '`App F`'-algebra (or '`F`-algebra' for short) is a pair given by the carrier `A`, and the *structure map* of type:

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

We define the initial `F`-algebra as the inductive type which is constructed by applying `App F` finitely many times.

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

`LFix_in` is the initial algebra in, while `LFix_out` is its inverse in$^\circ$ (see Section 2). As an example, the initial `F`-algebra for the container that is isomorphic to 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`.

The `LFix F` setoid equality is the least fixed point of the `App F` setoid equality, i.e. if `LFixR F` represents the setoid equality of `LFix F`. Let `x y : LFix F`, then `LFixR F x y` iff

```
shape x =e shape y /\ (forall p p', projT1 p = projT1 p'
   -> LFixR F (cont x p) (cont y p'))
```

We define smart constructors for the isomorphism of least fixed points as proper morphisms:

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

Catamorphisms are constructed so that they structurally deconstruct the datatype, call themselves recursively, and then compose the result using an `F`-algebra.

```
Definition cata_f (a : Alg F A) : LFix F -> A
  := fix f (x : LFix) := match x with
        | LFix_in x => a {shape := shape x; cont := fun e => f (cont x e)}
       end.
```

It is easy to prove that catamorphisms respect setoid equalities. We define `cata` as a setoid morphism between the setoid of `F`-algebras and the setoid of functions from `LFix F` to `A`:

```
cata : forall `{setoid A}, Alg F A ~> (LFix F ~> A)
```

Finally, we prove that catamorphisms satisfy their universal property (see Section 2):

```
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.

3.3 Coalgebras and Anamorphisms

In general, for a container `F`, an `F`-coalgebra is a pair of a carrier `X` and a structure map `X -> App 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 of `App F`. We define it using a coinductive data type:

```
CoInductive GFix F : Type := GFix_in { GFix_out : App F GFix }.
```

`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 the `App F` setoid equality. 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: the *anamorphism*.

```
Definition ana_f_ (c : Coalg F A) :=
  cofix f x := let cx := c x in
   GFix_in { shape := shape cx; cont := fun e => f (cont cx e) }.

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

From this definition the universal 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 an `F`-coalgebra homomorphism then it must be the anamorphism on the same coalgebra.

3.4 Mechanising Hylomorphisms

Recall that 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.

As we mentioned in Section 2, given an F-algebra a and F-coalgebra c, f is a hylomorphism if it satisfies

$$f = a \circ F\ f \circ c.$$

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

In order to find the unique solution we restrict ourselves to the so-called *recursive coalgebras* [3, 6]. We mechanise *recursive hylomorphisms* which are guaranteed to have a unique solution to the hylomorphism equation. These are hylomorphisms where the coalgebra is *recursive*, i.e. coalgebras that terminate on all inputs. We represent recursive coalgebras using a predicate that states 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.
```

`RecF` represents that a coalgebra will eventually terminate on an input. The base case takes place when the set of valid positions in the container extension returned by `h x` is empty. For convenience, we equip recursive coalgebras with an additional proof of termination:

```
Notation RCoalg F A := ({ c : Coalg F A | forall x, RecF c x }).
```

Recursive hylomorphisms are implemented by structural recursion on the proof that coalgebra `c` eventually terminates for some input `x` (`RecF c x`).

```
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
        | cx => fun H =>
            a { shape := shape cx ; cont := fun e => f (cont cx 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 a proper morphism, and use this proof to build the corresponding higher-order proper morphism:

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

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

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

Fusing a hylomorphism with any algebra or coalgebra homomorphism (in the code below, `f2` and `f1` respectively) falls out from this uniqueness property:

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

Lemma hylo_fusion_r (h1 : RCoalg F B) (g1 : Alg F C) (h2 : RCoalg F A) (f1 : A ~> B)
  :  h1 \o f1 =e fmap f1 \o h2 -> hylo g1 h1 \o f1 =e hylo g1 h2.
```

```
(* E2 : f2 \o g1 =e g2 \o fmap f2 *)
(* ------------------------------------ *)
(* Goal : f2 \o hylo g1 h1 =e hylo g2 h1 *)
apply hylo_uniq.
       (* f2 \o hylo g1 h1 =e (g2 \o fmap (f2 \o hylo g1 h1)) \o h1        *)
rewrite fmap_comp.
rewrite ... (* rearranging by associativity *)
       (* f2 \o hylo g1 h1 =e ((g2 \o fmap f2) \o fmap (hylo g1 h1)) \o h1 *)
rewrite <- E2.
rewrite ... (* rearranging by associativity *)
       (* f2 \o hylo g1 h1 =e f2 \o ((g1 \o fmap (hylo g1 h1)) \o h1)      *)
rewrite <- hylo_unroll.
       (* f2 \o hylo g1 h1 =e f2 \o hylo g1 h1                             *)
```

Figure 1 Rewrite steps to prove `hylo_fusion_l` in our mechanisation. The steps are exactly the same that would be required in a manual pen-and-paper proof.

It is important to highlight that, in our mechanisation, these proofs follow *exactly* the steps that one would do in a pen-and-paper proof. We show the series of rewrite steps for `hylo_fusion_l` in Figure 1. The steps of this proof are: (1) apply the uniqueness law of recursive hylomorphisms; (2) rewrite using that functors preserve composition; (3) rewrite using the condition of `hylo_fusion_l`; (4) use the uniqueness law of hylomorphisms to fold $a \circ F\ f \circ c$ into f.

Proving the deforestation optimisation is a straightforward application of `hylo_fusion_l` (or `hylo_fusion_r`).,

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

3.4.1 On the subtype of finite elements

As we mention in Section 2, we define recursive anamorphisms as hylomorphisms built by using a recursive coalgebra, and the respective initial `F`-algebra. In other words, in this development we have defined recursive anamorphisms on inductive data types. We might as well have 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:

```
Definition FinF : GFix F -> Prop := RecF g_out.
```

`FinF` represents finiteness since it must contain a base case with no positions, after a finite number of applications of `g_out`. Now the subtype `{x : GFix F | FinF x}` of finite elements for `GFix F` is isomorphic to its corresponding inductive data type `LFix F`. We show this by defining 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) (RecF_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 hylomorphisms on the recursive coalgebra `c`, and the initial `F`-algebra: `hylo l_in c`. The main advantage of this definition is that recursive anamorphisms compose with catamorphisms without the need to reason about termination and finiteness of values.

3.4.2 Proving Correctness

Suppose that we have an algebra `a : Alg F B`, a recursive coalgebra `c : RCoalg F A`, and a property `P : A -> B -> Prop`, and we want to guarantee that the hylomorphism satisfies this property: `forall x, P x (hylo a c x)`. Such proofs in our framework are done by induction on the proof that the recursive coalgebra terminates. In particular, the induction hypothesis would be that `P` is satisfied by any recursive call of the hylomorphism

```
forall e, P (cont (c x) e) (hylo a c (cont (c x) e))
```

Then, by the uniqueness property of hylomorphisms, it is enough to prove that `P` is satisfied by the unfolding of the hylomorphism:

```
P x ((a \o fmap (hylo a c) \o c) x)
```

We will see in Section 4 one example where we will prove that the output of a sorting algorithm is a sorted permutation of the input, and show that these proofs are comparable to other techniques for non-structural recursion, with the additional advantage that any result that is proven on a hylomorphism will also hold for any result of doing program calculation.

4 Extraction

In this section we show how programs and their optimisations can be encoded in our framework and how can they be extracted to idiomatic functional code. Although our examples use OCaml as target language, extraction works equally well to other target languages (e.g. Haskell or Scheme). In particular, we show how to formalise sorting algorithms by looking at quicksort (Section 4.1), we show how to formalise dynamic programming algorithms by looking at knapsack (Section 4.2), and, finally, we show an example of the shortcut deforestation optimisation (Section 4.3).

4.1 Sorting Algorithms

We now focus on divide-and-conquer sorting algorithms and demonstrate how to use program calculation to apply a fusion optimisation. In the supplemental material accompanying this paper (https://github.com/dcastrop/coq-hylomorphisms), we formalise both mergesort and quicksort, but in this section, we focus solely on quicksort. The recursive structure of quicksort is given by the following functor

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

The idea is that a list is either empty or is split into a pivot which is represented by node of type `N` and the two sublists of type `X`.

The container encoding `ITreeF` has two shapes, one for leaves and one for nodes, and nodes have two positions, one for each sublist `l r : X` in `i_node`.

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

The validity predicate, `valid_f` below, simply specifies that positions are only valid in `Node`. Our tactics automatically discharge the necessary proofs to construct the respective setoid morphism.

```
Definition valid {A} (x : Tshape A * Tp) : bool :=
  match x with | (Node _ _, _)  => true | _ => false end.
```

The container for `ITreeF` is denoted by `TreeF`. For convenience, we define the wrappers `a_leaf` and `a_node` to construct values of the extension of this container. Now, the container for quicksort is given by `TreeF int`. In the definition below, we use `posL` and `posR` as wrappers for `Lbranch` and `Rbranch` with validity proofs.

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

Definition merge_f (x : App (TreeF int) (list int)) : list int :=
  let (sx, kx) := x in
  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.
```

The proofs that these are proper morphisms (called `merge` and `qsplit`) are automatically discharged by our tactics, but we still have to prove that the coalgebra `qsplit` is recursive. In general, we know that if the input is related to the data inside the structure functor returned by the coalgebra by some *well-founded relation* `R`, then the coalgebra is recursive. In this specific case, it is sufficient to show that the two sublists are smaller than the input list.

```
  forall x (p : Pos (shape (qsplit x))), length (cont (qsplit x) p) < length x.
```

We can directly write mergesort as `hylo merge qsplit`, or derive it by program calculation from the composition of `cata merge`, and the recursive anamorphism `hylo l_in qsplit`. The resulting extracted code is similar to a hand-written implementation:

```
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))
```

The correctness proof is done by proving the following statement:

```
forall (l : list int), Sorted (hylo merge qsplit l) /\ Perm l (hylo merge qsplit l).
```

that is any list produced by quicksort is sorted and is some permutation of the original one.

The proof is by induction on the fact that `qsplit` is recursive. This means that `qsplit` terminates on `l`. In practice this is done by using the tactic `induction (recP qsplit l)`. Then we can unfold `hylo merge qsplit` to `merge \o fmap (hylo merge qsplit) \o qsplit`

using hylomorphism uniqueness (Section 3.4). The rest of the proof is standard. Overall, our proof is shorter than alternative implementations, and of comparable complexity[1].

4.1.1 Fusing a divide-and-conquer computation

We now show how we can use program calculation techniques to fuse a traversal with a divide-and-conquer algorithm to obtain a new program that only performs recursion once instead of twice. In particular, we can map a function to the result of quicksort and obtain a new program which orders the elements and computes the function on every element by traversing the list only once. The composition of these two programs is given by the following:

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

where `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 Rocq's generalised rewriting [26] and `hylo_fusion_l` to fuse `times_two` into `hylo merge qsplit` which gives the program `hylo (merge \o natural times_two) qsplit`. In this definition, `natural` defines a natural transformation by applying `times_two` to the shapes thus multiplying every pivot by two.

```
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))
```

The extracted OCaml code is a single recursive traversal. Note the similarity to a hand-written implementation, and that it has been derived by fusing two different recursive programs using regular Rocq rewrite tactics.

It would be straightforward to prove the correctness of the fused version, by simply proving that `Lmap times_two` preserves the order of the elements.

4.2 Knapsack

Dynamorphisms are hylomorphisms where the algebra has access to the memory table of the result of all previous recursive calls. To implement this we need to formalise the idea of a memory table which is in our case simply represented by the cofree comonad. For a functor $G : \mathcal{C} \to \mathcal{C}$ the cofree comonad is defined as the greatest solution to the following equation

$$G_\infty A \cong A \times G(G_\infty A)$$

In words, these are the (possibly) infinite trees which branch out with shape G. For example, for the identity functor $\mathrm{Id}_\infty A$ is the type of infinite streams. For $GX = 1 + X$, the type $G_\infty A$ is isomorphic to $A \times (1 + G_\infty A)$. This is the type representing our memory table with the operation $\mathsf{head}_A : G_\infty A$ being the result of the previous recursive call. Notice that in the case of a recursive G-coalgebra this type only contains finite G-trees and so we can view $G_\infty A$ as the finite G-trees thus being able to inspect all the previous recursive calls. This has been expounded thoroughly by Hinze et al. [15].

In our framework, we formalise G as a container with shape `Sg` and position `Pg`, then `Memo` is the least fixed-point of the functor $FX = A \times GX$:

[1] The QSort definition and correctness proof in the equations package is under 200LOC – https://github.com/mattam82/Rocq-Equations/blob/8.20/examples/quicksort.v, and other online proofs of correctness are of similar complexity https://gist.github.com/RyanGlScott/ff36cd6f6479b33becca83379a36ce49

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

The operations of the cofree comonad `Memo` are the `Cons`, the `headT` and the `tailT`:

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

Finally, we define a dynamorphism as a hylomorphism from a type `B` into a type `Table` post-composed with the `headT` map which outputs the last result from the memory table:

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

Note the map `App G (Table A) ~> A` corresponding to a map $GG_\infty A \to A$ is not a G-algebra. To recover the G-algebra we need to transform it by pairing it with the identity map and post-composing it with `consT`. This algebra uses this table to lookup elements, instead of triggering a further recursive call, then elements are inserted into the memoisation table by the use of `consT` to the result of applying the algebra. In our example this algebra is called `knapsackA` (see accompanying code). Finally, given the recursive coalgebra `out_nat` on the natural numbers, we can define knapsack, given a list of pairs of weights and values `wvs`, using the recursion scheme for dynamorphisms:

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

The full extracted code coming from this specification is available in the artefact accompanying this paper, and an interested reader can check that simple inlining would produce the following:

```
let knapsack wvs x = let (y, _) = (let rec f x0 =
      if x0=0 then
        { lFix_out = {shape = Uint63.of_int (0); cont  = fun _ -> f 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 that in the extracted code, the recursive calls to `f` build the memoisation table, and that 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

We now showcase the shortcut deforestation optimisation on lists. This is used to remove the creation of intermediate data structures by fusing multiple recursive traversals into one. For example, we can use it to show that the following code from Takano and Meijer [28], which uses three different hylomorphisms, can be fused into one:

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

Here `sf1` is the composition of `length`, `Lmap f` and `append ys` which are, in particular, catamorphisms. To prove our goal we use the so called *acid rain* theorem [28] which states that given a parametric function of the following type:

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

we can conclude, by parametricity, that the following equation holds:

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

Unfortunately, this is not provable in Rocq without adding the parametricity axiom [17], but we can prove it for specific functors, e.g. the functor of lists.

We prove a specialised version of the acid rain theorem that fuses hylomorphisms defined in terms of the following function `tau` that maps list algebras to list algebras. This essentially is a function on lists which uses the algebra for both the recursive step and for the base case where it is used in the hylomorphism to continue the recursion:

```
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.
```

Our acid rain theorem is stated as follows:

```
  hylo a l_out \o hylo (tau l l_in) c =e hylo (tau l a) c
```

Now we use `tau` to define the `append` function as follows:

```
Definition append (l : list A) := hylo (tau l l_in) ilist_coalg.
```

Here, `ilist_coalg` is a recursive coalgebra from Rocq lists to the `ListF` container. The acid rain theorem allows us to fuse this definition of append with the maps `Lmap` and `length` which is the optimisation we wanted in the first place. The following is the extracted OCaml code from the optimised program:

```
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)
```

Notice how the `Lmap` is optimised away and the function `length` is applied to both the arguments of the append function.

As a second example, we prove that `length` fuses with the naive quadratic `reverse` function:

```
Definition sf2 : Ext (length \o reverse).
```

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)
```

Here by fusing the hylomorphism for the reverse function with the length function we obtain the original length function.

5 Related Work

There is existing prior work on formalising recursion schemes in a proof assistant. Tesson et al. demonstrated the efficacy of leveraging Rocq to establish an approach for implementing a robust system dedicated to verifying the correctness of program transformations for functions that manipulate lists [29]. Murata and Emoto went further and formalised recursion schemes

in Rocq [22]. 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 code from their formalisation. Mu et al. formalise hylomorphisms in Agda, and can do relational program transformation [21]. Their paper does not discuss extracting runnable code from their encodings, and they do not seem to formalise hylomorphisms in terms of generic functors and datatypes.

Larchey-Wendling and Monin encode recursion schemes in Rocq, by formalising computational graphs of algorithms [18]. Their work does not focus on encoding higher-order generic recursion schemes, and proving their algebraic laws. Castro-Perez et al. [7] encode the laws of hylomorphisms as part of a type system to calculate parallel programs from specifications. Their work focuses on parallelism, and they do not formalise their approach in a proof assistant, and they treat the laws of hylomorphisms as axioms.

Abreu et al. [2] encode divide-and-conquer computations in Rocq, using a recursion scheme in which termination is entirely enforced by its typing. This is a significant advance, since it *completely* avoids 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*, and we retain the ability to extract natural-looking OCaml code. Through experiments, we found that code extracted from their formalisation makes heavy use of unsafe casts and has a generally complex structure that may be hard to understand and integrate with other external code. Due to the great benefit of entirely avoiding termination proofs, it would be interesting to extend their approach to improve code extraction.

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. For example, the mergesort in Rocq'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 recognisable as a form of mergesort" [2]. There are common approaches in Rocq to deal with termination [27, 25], but none of these approaches address program calculation techniques, and the mechanisation of fusion laws.

The fusion laws that we formalise in this paper are applied in the more general context of compilers for functional programming languages. Fusion has been applied to recursive tree traversals, in order to reduce the number of times each section of the tree must be visited. Research has demonstrated that this sort of fusion can be performed automatically by a compiler [23]. The correctness of tree traversal fusion has been verified mechanically in Rocq [9], but the authors do not formalise unfolds and hylomorphisms.

Another related family of transformations are worker/wrapper transformations [13], which have been used in the context of optimising compilers, and have been mechanised in Agda [24]. This technique has been applied to improving the performance of programs defined using folds via fusion [16]. Their work does not focus on the use of an interactive proof assistant, but rather on the manual use of equational reasoning to re-write programs and on the automatic transformation of functional programs by a compiler.

6 Conclusions and Future Work

In this work we mechanise hylomorphisms and their algebraic laws and use them to perform program calculation and optimisations. The resulting programs can be later extracted into idiomatic and runnable code. Our mechanisation is *fully axiom-free*. This formalisation allows us to use Rocq as a framework for program calculation in a way that is close to common practice in pen and paper program calculation proofs. This implies that every rewriting step is the result of applying a formal, machine-checked proof that the resulting program is extensionally equal to the input specification.

As part of the future improvements, we will study how to mitigate the problem of setoids. 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 proper 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, Rocq is unable to inline a number of trivially inlineable definitions. We will study alternative definitions, or extensions to Rocq's code extraction mechanisms to force the full inlining of all container code that is used in hylomorphisms. Following work by Miculan and Paviotti [19] an interesting next step would be to embed the notion of hylomorphism into a type theory with a notion of distributed computations (e.g. ML5), the resulting extracted code would be then implemented in a distributed programming language (e.g. Erlang). 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. [2] 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. doi:10.1016/J.TCS.2005.06.002.

2 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.

3 Jiří Adámek, Stefan Milius, and Lawrence S. Moss. On well-founded and recursive coalgebras. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures*, pages 17–36, Cham, 2020. Springer International Publishing.

4 Richard S. Bird and Oege de Moor. The algebra of programming. In Manfred Broy, editor, *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, pages 167–203, 1996.

5 Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers an overview. *Mathematical Structures in Computer Science*, 26(1):3888, 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. 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 417. ACM, 2016. doi:10.1145/2951913.2951920.

8 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.

9 Eleanor Davies and Sara Kalvala. Postcondition-preserving fusion of postorder tree transformations. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 191200, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377555.3377884.

10 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.

11 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.

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

13 Andy Gill and Graham Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19, 03 2009. doi:10.1017/S0956796809007175.

14 Ralf Hinze, Thomas Harper, and Daniel W. H. James. Theory and practice of fusion. In Jurriaan Hage and Marco T. Morazán, editors, *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*, volume 6647 of *Lecture Notes in Computer Science*, pages 19–37. Springer, 2010. doi:10.1007/978-3-642-24276-2_2.

15 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.

16 Graham Hutton, Mauro Jaskelioff, and Andy Gill. Factorising folds for faster functions. *J. Funct. Program.*, 20(34):353373, July 2010. doi:10.1017/S0956796810000122.

17 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. doi:10.4230/LIPIcs.CSL.2012.381.

18 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.

19 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.

20 Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. The third homomorphism theorem on trees: downward & upward lead to divide-and-conquer. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 177–185. ACM, 2009. doi:10.1145/1480881.1480905.

21 Shin-Cheng Mu, Hsiang-Shang Ko, and Patrik Jansson. Algebra of programming in agda: Dependent types for relational program derivation. *J. Funct. Program.*, 19(5):545–579, 2009. doi:10.1017/S0956796809007345.

22 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.

23 Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. Treefuser: a framework for analyzing and fusing general recursive tree traversals. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133900.

24 Neil Sculthorpe and Graham Hutton. Work it, wrap it, fix it, fold it. *Journal of Functional Programming*, 24(1):113127, 2014. doi:10.1017/S0956796814000045.

25 Matthieu Sozeau. Program-ing finger trees in Coq. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 13–24. ACM, 2007. doi:10.1145/1291151.1291156.

26 Matthieu Sozeau. A new look at generalized rewriting in type theory. *J. Formaliz. Reason.*, 2(1):41–62, 2009. doi:10.6092/ISSN.1972-5787/1574.

27 Matthieu Sozeau and Cyprien Mangin. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.*, 3(ICFP):86:1–86:29, 2019. doi:10.1145/3341690.

28 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 306313, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224164.224221.

29 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.

30 Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990. doi:10.1016/0304-3975(90)90147-A.

31 Glynn Winskel and Mogens Nielsen. Models for concurrency. In *Handbook of Logic in Computer Science*. Oxford University Press, 05 1995. doi:10.1093/oso/9780198537809.003.0001.