Mechanising Hylomorphisms for Extraction of Executable Code

David Castro Perez[1 [0000−0002−6939−4189]], Marco Paviotti[2,3 [0000−0002−1513−0807]], and Michael Vollmer[3 [0000−0002−0496−8268]]

University Of Kent, Canterbury, CT2 7NZ, United Kingdom
{D.Castro-Perez, M.Paviotti, M.Vollmer}@kent.ac.uk

Abstract. Generic programming with recursion schemes provides a powerful abstraction for structuring recursion, in part due to the rigorous set of algebraic laws that they satisfy. These laws are the basis for reasoning about program equivalences and, therefore , they can be used for reasoning about program correctness and optimisations. Some of these optimisations are successfully applied by compilers of (functional) languages. 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 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.

1 Introduction

Structured recursion schemes [19,20] are powerful abstractions that capture common patterns of recursion. The main benefit of structuring computation using recursion schemes, is that they enjoy well-estabished *algebraic properties* that can serve as a foundation for reasoning about program equivalences, transformations, and optimisations (e.g. *fusion* laws, or semi-automatic parallelisations [34,13,29,7]). These algebraic properties led to their use in the context of *program calculation*, where programmers would describe their code using simple, inefficient specifications in an *algebra of programming* [4], and then use the algebraic laws of this algebra of programming to *calculate* an efficient version of

the same algorithm. Suppose, for example, that we want to write a program that sorts a list of integers, and then multiplies by 2 all its elements. 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 t
```

Instead of writing this function directly, we may observe that we can derive it by *fusing* a regular *quicksort* OCaml implementation, with `map (fun x -> x * 2)`. This is the main idea behind program calculation: start from a simple specification, e.g. `map` $(\lambda x.\ x \times 2) \circ$ `sort`, and use program equivalences and algebraic laws to rewrite it to an optimised version (e.g. the fused OCaml implementation above). We will revisit a similar example in Section 5.1.

Despite the rigorous set of algebraic laws that are satisfied by recursion schemes, there is a lack of tool support for their use in the context of program calculation. In fact, most of the work on applying program calculation is done by performing pen-and-paper proofs, and then translating the result to specific instances of recursion schemes, implemented in a programming language (generally Haskell). There are some examples of implementations of program calculation techniques, but these implementations are scarce, not up to date, and not verified in a proof assistant. For example, Cunha et al. [9] automated program calculation techniques by a custom Haskell implementation.

Several authors build tools for applying program calculation techniques by *mechanising* them as part of a proof assistant. However, most of the work in mechanising recursion schemes focuses on a narrow subset of the known recursion schemes due to termination issues, or do not focus on proving algebraic laws of generic recursion schemes [35,30,24]. Indeed, most of the mechanisations of generic recursion schemes focus on *maps* and *folds*, and few authors focus on the *unfolds*. Omitting *unfolds* severely limits the expressivity of the resulting mechanisations, and prevent them from being able to mechanise the most general recursion scheme: a divide-and-conquer algorithm which goes under the name of *hylomorphism* [26,21]. This generality has been proven by Hinze et al. [20], by showing that *all known recursion schemes are instances of hylomorphisms*. To date, no work has mechanised hylomorphisms in the Coq proof assistant, together with their algebraic laws.

Recently, Abreu et al. [2] 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, and it is not well-suited for program calculation. This is unfortunate since code extration is what allows the execution of code that has been verified in Coq [31,25,28,32]. In Abreu et al's approach, extraction (1) does not preserve the recursive structure of common implementations; and (2) leads 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 [12] and, moreover, it invalidates the fast-and-loose principle [10].

The contributions of this paper are as follows. This work presents the first Coq formalisation *hylomorphisms* that (1) is *fully axiom-free*; (2) allows the extraction of idiomatic OCaml code; and (3) can use regular Coq equalities to do program calculation, derive correct implementations, and apply optimisations. The full mechanisation is open source. While programmers still need to reason about the termination of their programs, the use *program calculation* can ease this task by allowing the use of fusion on programs that have already been proven to terminate.

The remainder of the paper is structured as follows:

- In Section 2 we give an overview about recursion schemes.
- In Section 3 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 4 we mechanise folds, unfolds, and hylomorphisms, as well as proving their uniqueness properties in Coq.
- In Section 5 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.

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 in Haskell are specific to a particular data structure, in this case Lists, we can generalise further these ideas 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, recursion with accumulators and so on. 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. This is done by looking 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 associativity law. We denote $\mathrm{Hom}_{\mathcal{C}}(A, B)$ the set of arrows from A to B and we use the letters $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E} \ldots$ for categories[1]. The initial object in a category, 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, denoted by 1, is the object such that for any other object A there is a unique arrow $A \xrightarrow{!} 1$. Initial and terminal object 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 types are 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 $F : \mathrm{Hom}_{\mathcal{C}}(A, B) \to \mathrm{Hom}_{\mathcal{D}}(FA, FB)$ 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 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 terminal object in **Set** is the *unit* type, that only has one inhabitant, $* : 1$. The $\mathsf{List}(-)$ type is a functor. Its action on objects is to take any set A to $\mathsf{List}(A)$, and its functorial action $\mathsf{List}(f) : \mathsf{List}(A) \to \mathsf{List}(B)$ is given by $\mathsf{List}(f)(*) = *$ 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. However, there is a more general way of proving these type of definitions are well-defined. Similarly, the set of streams $\mathsf{Str}(A) \cong A \times \mathsf{Str}(A)$ is the greatest set generated by the constructor $\mathsf{cons} : A \times \mathsf{Str}(A) \to \mathsf{Str}(A)$.

Functors in Haskell It is often helpful for functional programmers to think of categorical constructions in terms of Haskell thereby assuming that Haskell types are sets and programs are functions [10].

We can declare the class of types which are functors. This is done by imposing that a function `f : * -> *` comes with a functorial action given by function `fmap`:

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

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

We now define the type of lists as usual and then show that it is an instance of the class **Functor** by implementing the fmap program:

```
data List a = Nil | Cons a (List a)

instance Functor List where
 fmap f Nil = Nil
 fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

2.2 Algebras and Catamorphisms

For a functor $F : \mathcal{C} \to \mathcal{C}$ 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* $f : X \to Y$ in $\mathcal{C}$ 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 the initial F-algebra is precisely the *least fixed-point* for the functor F which we denote by $(\mu F, \mathsf{in})$ with $\mathsf{in} : F\mu F \to \mu F$ and $\mathsf{in}^\circ : \mu F \to F\mu F$ witnessing the isomorphism $F\mu F \cong \mu F$. By the property of initial F-algebras, for any other F-algebra (X, a_X) there exists a unique F-algebra homomorphism, denoted by $(\!|\beta|\!) : (\mu F, \mathsf{in}) \to (X, a_X)$ and pronounced "*catamorphism*" satisfying:

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

In words, any F-algebra homomorphism from the initial F-algebra is a catamorphism. 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 : **Set** $\to$ **Set** mapping $X \mapsto 1 + A \times X$. An F-algebra is a set B together with a structure map $[b, i] : 1 + A \times B \to B$. The initial F-algebra is clearly the set of lists $\mathsf{List}(A)$. The catamorphism associated with the type of lists is the unique arrow recursively translating the initial algebra $[\mathsf{nil}, \mathsf{cons}]$ into the algebra $[b, i]$. In functional programming this is commonly referred to as $\mathsf{foldr} : (1 \to B) \to (A \times B \to B) \to B$. We can in fact set $\mathsf{foldr}\ base\ ind = (\!|[base, ind]|\!)$.

Inductive Types and Catamorphisms in Haskell We now implment the least fixed-point of a functor `f`. Inuitively this is the type of `f`-branching trees of finite depth:

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

Note that since Haskell is a lazy language with general recursion this type can also represent infinite trees. However, the point of using recursion schemes is precisely that of avoid using general recursion in the first place.

A catamorphism can be readily implemented as follows

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata alg = alg . fmap (cata alg) . In
```

As an example of how to construct inductive types we define the functor $X \mapsto 1 + A \times X$ as data type parameterized by a type:

```
data ListF a x = Nil | Cons a deriving Functor
newtype List a = (Fix ListF) a
```

It should be easy now to see that a foldr is precisely a catamorphism over the type `List` which we have just defined. For example, summing up the elements of a list can be implemented using a catamorphism as follows

```
sum :: Fix (ListF Int) -> Int
sum xs = cata (\case { Nil -> 0; Cons n m -> n + m } )  xs
```

In the program above, the code inside the lambda case is precisely the `ListF Int`-algebra for the catamorphism.

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) 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. In this case, we can even set some notation for the transition map:

$$x \xrightarrow{l} y \triangleq c_X(x) = (l, y)$$

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 $c_Y \circ f = F(f) \circ c_X$. Terminal object in this category B-coalgebra corresponds to the greatest fixed-point for the functor B, denoted by $(\nu B, \mathsf{out})$ where out denotes the final B-coalgebra $\mathsf{out} : \nu B \to B\nu B$.

For example, the terminal coalgebra for the functor $BX = L \times X$ is the set of of infinite streams, that is the greatest solution to the domain equation $\mathsf{Str}(A) \cong A \times \mathsf{Str}(A)$. As a result, for any 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*". We spell out the uniqueness property:

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

In words, for any B-coalgebra, if there is any other B-coalgebra homomorphism f into the terminal object then it must be the anamorphism on the same coalgebra. As a result of the uniqueness property we can derive the fusion laws. For all B-coalgebra homomorphisms $f : (X, c_X) \to (Y, c_Y)$ we have

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

Coinductive Types and Anamorphisms in Haskell Similarly to the inductive case, we can implement coinductive types and catamorphisms as follows:

```
newtype CoFix f = OutOp {out :: f (Fix f )}
```

As mentioned above, this type is similar to the data type `Fix` we already defined. However, this time we are going to use this assuming that it is the coinductive fixed-point by defining an anamorphism on it:

```
ana :: Functor b => (a -> b a) -> a -> CoFix b
ana coalg = OutOp . fmap (ana coalg) . coalg
```

For example, the stream of natural numbers can now be defined using an anamorphism

```
nat :: CoFix (ListF Int)
nat = ana (\x -> Cons x (x+1)) 0
```

where the argument to the anamorphism is the `ListF Int`-coalgebra.

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

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

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.

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

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. The uniqueness property of the hylo 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 hylo fusion laws, we can prove the well-known *deforestation optimisation*, also known as the *composition law* [18]. 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.

Haskell Implementation of Hylomorphisms We implement the hylomorphism tightly following the theory explained above:

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

A classic example of a divide-and-conquer algorithm that can be written as an hylomorphism is the quicksort program below:

```
partition x xs = ([l | l <- xs, l < x], x, [r | r <- xs, r >= x])

quicksort :: [Int] -> [Int]
quicksort [] = []
quicksort (x:xs) = let (xs, x, ys) = partition x xs
                   in (quicksort xs) ++ [x] ++ (quicksort ys)
```

The partition program takes an element `x` and a list and returns a triple consisting of a list with all the elements in the list smaller than `x`, the element itself and a list with all the elements in the list greater or equal than `x`. The quicksort program first runs the partition program, which divides the list in three parts, calls itself recursively on the sublists and then puts the results together using the concatenation function.

To implement this as an hylo we define the functor representing trees where leaves are just empty and each node contains a value:

```
data TreeF a k = Leaf | Branch k a k deriving Functor
```

Now the partition program can be written as a `TreeF`-coalgebra

```
divide :: [Int] -> TreeF Int [Int]
divide [] = Leaf
divide (p:xs) = Branch [l | l <- xs, l < p] p [r | r <- xs, r > p]
```

and the concatenation function can be written as `TreeF`-algebra

```
conquer :: TreeF Int [Int] -> [Int]
conquer Leaf = []
conquer (Branch ls x rs) = ls ++ [x] ++ rs
```

Finally, quicksort can be rewritten as an hylomorphism

```
quicksort = hylo divide conquer
```

As an important remark we have used the in-built list type for keeping the code as tidy as possible, but it would have been possible to use the inductive `List` type we defined above and define both `divide` and `conquer` via catamorphisms.

Finally, it is important to note that both `divide` and `conquer` are not initial algebras or terminal coalgebras on `ListF`. These are rather (co)algebras on `TreeF`. Hence the terminating argument relies solely on the fact that `divide` is a recursive coalgebra thus there is an unique solution to `hylo divide conquer`.

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 traces it can produce from the seed function are the finite ones. We denote 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 νF. This fact falls out from the uniqueness property of the hylomorphism and the fact that recursive anamorphisms satisfy the same equation.

3 Mechanising Extractable Container Functors

In this section, we focus on mechanising *container functors* in a way that is suitable for code extraction. Recall from Section 2 that we need to be able to abstract away from particular shape of the data, and we can do this using *functors* that have suitable fixed-point properties; i.e. those which have a initial algebras. A common approach to construct such functors is to use *containers* [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 3.3).

3.1 Functors and Containers

A straightforward mechanisation of functors as defined in Section 2 does not work in Coq due to the strict positivity condition. In fact, if we simply require that functors are functions `F : Type -> Type` satisfying the necessary identity and

composition properties, then we will not be able to construct least/greatest fixed points of this type, and we will not be able to mechanise recursion schemes that follow this structure:

```
(* REJECTED due to negative occurrence of (LFix F) *)
Inductive LFix (F : Type -> Type) := LFix_in (lfix_out : F (LFix F)).
```

One common workaround for this is the use of Mendler-style recursion [27]. Roughly, the idea is to "pack" an abstract type together with the functor that captures the structure of the recursion. From this abstract type, we will be able to "extract" the contents of the functor. This is, for example, the solution used by the PaCo library [22], and a simplified version of it is the following:

```
Inductive MendlerLFix (F : Type -> Type) :=
  In Y (next : Y -> MendlerLFix F) (in_op : F Y).
```

While this definition solves the strict positivity problem, it does not solve the problems of: (1) reasoning about data/program equalities; and (2) extraction to idiomatic OCaml code. To reason about program equalities, we would need to reason about equalities of functions with *some abstract* `Y` as their domain. Furthermore, the extraction of `F Y` will lead to unsafe casts in the extracted OCaml code.

Another common solution is to the strict positivity problem is the use *containers* [1]. A container $S \triangleright P$ is defined by a type of *shapes* $S : \mathbf{Type}$ and a *family of position types* indexed by shapes $P : S \to \mathbf{Type}$. An *extension* of this container is a functor $[\![S \triangleright P]\!]$, whose action on objects is given by $[\![S \triangleright P]\!]\ X = \Sigma s : S.\ P\ s \to X$, and on morphisms is given by post-composition, keeping the shape the same $[\![S \triangleright P]\!]\ f = \lambda(s,\ g).\ (s,\ f \circ g)$.

To explain the role of shapes and positions in containers, consider the functor $F\ X = 1 + A \times X$. An equivalent container, i.e. a container with an extension that is isomorphic to F, requires a shape that can distinguish two cases, indicating that there are two constructors to build an object of type $F\ X$. For example, we could define $S_F = 1 + A$. We need to consider two cases for the family of positions: $P_F(\mathsf{inj}_1\ *) = 0$, because there are no occurrences of X in the left case; and $P_F(\mathsf{inj}_2\ a) = 1$, because there is one occurrence of X in the right hand side. It is easy to see that $[\![S_F \triangleright P_F]\!] \cong F$. In general, every polynomial functor has an equivalent container, and we have mechanised this result.

3.2 Extractable Containers

A straightforward encoding of containers in Coq would use dependent types to represent families of positions. However, Ocaml's type system is not equipped to handle these, which will lead to extracting OCaml code with unsafe casts.

Consider instead representing the positions of a container using a decidable validity predicate which assigns shapes to positions. In Coq, we use a boolean function and a coercion from `bool` to `Prop` to represent the decidable predicate `valid`, similarly to SSReflect [17]. Given a `Shape : Type`, and a type of *all possible positions* `APos : Type`, the family of positions for a given shape can be defined as a dependent pair:

```
Definition Pos (s : Shape) = { p : APos | valid s p }.
```

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 s -> X
P1, P2 : valid s p
--------------------------
k (existT _ p P1) = k (existT _ 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`.

Suppose that `F` is a container represented in Coq. One can think of these `F` as pairs containing the shapes and families of positions (which are, in turn, defined in terms of decidable validity predicates). An extension of a container `F`, `App F` in Coq, is then defined as follows:

```
Record App F (X : Type)
  := { shape : Shape F; cont : {p | valid shape p} -> X }.
```

This approach solves the problem of the unsafe casts, since the code extracted for `{p | valid shape p}` will be an OCaml singleton type defined as the OCaml equivalent to `Pos`.

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:

```
Inductive AppR (x y : App F X) : Prop :=
| AppR_ext (shape x = shape y)
           (forall p1 p2, projT1 e1 = projT1 e2 -> cont p1 = cont p2).
```

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)
```

3.3 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'`.

```
Record 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:

```
Inductive AppR F `{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 this leads to simpler code. We explain why in Section 4.1, and the mechanisation of initial algebras of container extensions.

Our definition of morphisms, and the use of different equivalences leads to the well-known "setoid hell". We mitigate this problem 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 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.
```

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 will consider 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.

4 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 4.1), then we formalise coalgebras (Section 4.2) and then we put together these notions to formalise recursive coalgebras and hylomorphisms (Section 4.3).

4.1 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 is a pair given by the carrier `A`, and the *structure map* of type:

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

For simplicity, we will abuse terminology, and call these `F`-algebras, instead of '`App F`'-algebras.

The initial `F`-algebra can be defined as follows, thanks to the use of containers:

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

where `LFix_in` is the initial algebra `in`, while `LFix_out` is its inverse in° (see Section 2). As an example, the initial `F`-algebra for the container that is isomorpic 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`.

We define `LFix` as a setoid, where its equivalence relation can be described as the least fixed point of `AppR` (see Section 3.3). 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)
```

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

```
Definition cata_f (alg : Alg F A) : LFix -> A
  := fix f (x : LFix)
     := match x with
        | LFix_in ax =>
            alg {shape := shape ax; cont := fun e => f (cont ax 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 `{setoid A}, Alg F A ~> LFix ~> A
```

We prove that catamorphisms satisfy the universality property we explained in 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.

4.2 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 for `App F`. We define it using a coinductive data type:

```
CoInductive GFix F : 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: 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 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.

4.3 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 and F-coalgebra, the hylomorphism is the unique solution (when it exists) to the equation

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

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

In order to find a 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 hylomoprhisms 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.
```

For convenience, we package recursive coalgebras with their proofs that they terminate for all inputs:

```
Notation RCoalg F A := ({ c : Coalg F A | 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 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).
```

```
(* 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                             *)
```

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

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 `{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_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.
```

Hylomorphisms fusion 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.
```

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 the goal using the property that functors preserve composition; (3) rewrite the goal 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) (INV: h2 \o g1 =e id)
  : hylo g2 h2 \o hylo g1 h1 =e hylo g2 h1.
```

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 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 follows for a recursive coalgebra c:

```
Definition rana c := 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 of coinductive types.

5 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 5.1), dynammic programming and Knapsack (Section 5.2), and examples of the shortcut deforestation optimisation in our framework (Section 5.3).

5.1 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.

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))
```

5.2 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 [20]. 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 table, and remove an element:

```
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 :=  (* ... *)
```

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
        { 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 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.

5.3 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 [34]. 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 [23]. 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 [34]:

```
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 `length` 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)
```

6 Related Work

Encoding recursion schemes in Coq is not new. We compare our work with other encodings of program calculation techniques and recursion schemes in Coq, and discuss other approaches to fusion and to termination of nonstructural recursion.

6.1 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 [14]. Programmers may specify algorithms abstractly at a high level, and then derive an efficient implementation using the process of program calculation [4]. 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 [35]. Murata and Emoto went further and formalised recursion schemes in Coq [30]. 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 [24]. 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.

6.2 Divide and Conquer Recursion

Abreu et al. [2] 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*. 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.

6.3 Termination Checking of Nonstructural Recursion

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 [2]. 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" [2]. There are common approaches in Coq to deal with termination, but none of these approaches address program calculation techniques, and the mechanisation of fusion laws.

6.4 Fusion

Fusion in functional programming is a general technique for combining multiple computations into a single pass to eliminate intermediate data structures and improve efficiency. The fusion we have discussed in this paper relating to program calculation is one form of fusion, but there are other approaches.

Deforestation, introduced by Wadler [36], is a specific approach to fusion that aims to remove intermediate trees (such as lists) created between chained operations. A well-known method of deforestation is `foldr`/`build` fusion [15], which optimizes list-processing by fusing list-producing functions (like `map` or

`filter`) with list-consuming ones (like `foldr`), avoiding the creation of temporary lists. Extending this idea, stream fusion generalizes the concept to handle other data types, using a lazy stream abstraction to represent sequences and fuse transformations like `map` and `fold` [8].

In the context of Coq, Danvy explains the use of fold/unfold lemmas to mechanize equational reasoning about recursive programs [11]. Fold-unfold lemmas enable optimizations like fusion by allowing programs to be systematically simplified, including eliminating intermediate structures. Additionally, an approach to fusion called the worker/wrapper transformation [16] was formalised and mechanised in Agda [33].

7 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. [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. URL: https://doi.org/10.1016/j.tcs.2005.06.002, 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. 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.
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):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. Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, page 315–326, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1291151.1291199.
9. Alcino Cunha, Jorge Sousa Pinto, and José Proença. A framework for point-free program transformation. In Andrew Butterfield, Clemens Grelck, and Frank Huch, editors, *Implementation and Application of Functional Languages, 17th International Workshop, IFL 2005, Dublin, Ireland, September 19-21, 2005, Revised Selected Papers*, volume 4015 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2005. doi:10.1007/11964681_1.
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. OLIVIER DANVY. Fold–unfold lemmas for reasoning about recursive programs using the coq proof assistant. *Journal of Functional Programming*, 32:e13, 2022. doi:10.1017/S0956796822000107.
12. 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.

13. 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.
14. Jeremy Gibbons. The school of squiggol. In *Formal Methods. FM 2019 International Workshops*, pages 35–53, Cham, 2020. Springer International Publishing.
15. Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, page 223–232, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/165180.165214.
16. Andy Gill and Graham Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19, 03 2009. doi:10.1017/S0956796809007175.
17. Georges Gonthier and Roux Le. An ssreflect tutorial. 01 2009.
18. 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.
19. Ralf Hinze and Nicolas Wu. Unifying structured recursion schemes - an extended study. *J. Funct. Program.*, 26:e1, 2016. doi:10.1017/S0956796815000258.
20. 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.
21. 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.
22. Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. The power of parameterization in coinductive proof. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 193–206. ACM, 2013. doi:10.1145/2429069.2429093.
23. 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.
24. 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.
25. 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.

26. 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.
27. Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Ann. Pure Appl. Log.*, 51(1-2):159–172, 1991. doi:10.1016/0168-0072(91)90069-X.
28. 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.
29. 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.
30. 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.
31. 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.
32. 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.
33. Neil Sculthorpe and Graham Hutton. Work it, wrap it, fix it, fold it. *Journal of Functional Programming*, 24(1):113–127, 2014. doi:10.1017/S0956796814000045.
34. 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.
35. 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.
36. Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, January 1988. doi:10.1016/0304-3975(90)90147-A.