# Fantastic Morphisms and Where to Find Them
# A Guide to Recursion Schemes

Zhixuan Yang Nicolas Wu

Imperial College London

Manuscript 2019

**Abstract**

This document is a 'dictionary' of *structured recursion schemes* in functional programming. Each recursion scheme is motivated and explained with concrete programming examples in Haskell, and the dual of each recursion scheme is also presented.

## Contents

# 1 Introduction

Work on catamorphisms by Malcolm (1990b), the form of recursion that derives from initial algebra, motivated a whole research agenda concerned with capturing the pattern of many other recursive functions that did not quite fit the scheme. Just as with catamorphisms, these recursion schemes attracted attention since they make termination or progress manifest, and enjoy many useful calculational properties which would otherwise have to be established afresh for each new application.

## 1.1 Diversification

The first variation on the catamorphism was the paramorphism (Meertens, 1992), about which Meertens talked at the 41st 2.1 meeting in Burton, UK (1990), which describe recursive functions in which the body of structural recursion has access to immediate subterms as well as to their images under the recursion.

Then came a whole zoo of morphisms. Mutumorphisms (Fokkinga, 1990), which are pairs of mutually recursive functions; zygomorphisms (Malcolm, 1990a), which consist of a main recursive function and an auxiliary one on which it depends; histomorphisms (Uustalu & Vene, 1999), in which the body has access to the recursive images of all subterms, not just the immediate ones; so-called generalised folds (Bird & Paterson, 1999), which use polymorphic recursion to handle nested datatypes; and then there were generic accumulations (Pardo, 2002), which keep intermediate results in additional parameters for later stages in the computation.

While catamorphisms focused on terminating programs based on initial algebra, the theory also generalized in the dual direction: *anamorphisms*. These describe productive programs based on final coalgebras, that is, programs that progressively output structure, perhaps indefinitely. As variations on anamorphisms, there are apomorphisms (Vene & Uustalu, 1998), which may generate

subterms monolithically rather than step by step; futumorphisms (Uustalu & Vene, 1999), which may generate multiple levels of a subterm in a single step, rather than just one; and many other anonymous schemes that dualize better known inductive patterns of recursion.

Recursion schemes that combined the features of inductive and coinductive datatypes were also considered. The *hylomorphism* arises when an anamorphism is followed by a catamorphism, and the *metamorphism* is when they are the other way around. A more sophisticated recursion scheme is the *dynamorphism* which encodes dynamic programming schemes, where a lookup table is coinductively constructed in an inductive computation over the input.

### 1.2 Unification

The many divergent generalisations of catamorphisms can be bewildering to the uninitiated, and there have been attempts to unify them. One approach is the identification of recursion schemes from comonads by Uustalu *et al.* (2001). Comonads capture the general idea of 'evaluation in context' (Uustalu & Vene, 2008), and this scheme makes contextual information available to the body of the recursion. It was used to subsume both zygomorphisms and histomorphisms.

Another attempt by Hinze (2013) used adjunctions as the common thread. Adjoint folds arise by inserting a left adjoint functor into the recursive characterisation, thereby adapting the form of the recursion; they subsume accumulating folds, mutumorphisms (and hence zygomorphisms), and generalised folds. Later, it was observed that adjoint folds could be used to subsume rsfcs (Hinze *et al.*, 2013), which in turn draws on material from (Hinze, 2013).

Thus far, the unifications had dealt largely with generalisations of catamorphisms and anamorphisms separately. The job of putting combinations of these together and covering complex beasts such as dynamorphisms was achieved by Hinze *et al.* (2015) in a scheme that the Group dubbed the *mamamorphism*. This worked by using conjugates distributive laws to structure hylomorphisms

## 2 Background

This paper assumes familiarity with basic Haskell as all examples and recursion schemes are presented in Haskell. No knowledge of category theory is needed to understand most of the paper except Section 9, where higher-order functors and natural transformations play a central role. In the rest of this section, we introduce the basis of recursive schemes — recursive datatypes, viewed as fixed points of functors.

**Functors and Algebras** Endofunctors, or simply functors, in Haskell are type constructors `f` instantiating the following type class:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

Additionally `fmap` is expected to satisfy two functorial laws:

$$\texttt{fmap id} = \texttt{id}$$
$$\texttt{fmap}\ (\texttt{a} \circ \texttt{b}) = \texttt{fmap a} \circ \texttt{fmap b}$$

for any `a` and `b`.

Two terms from category theory that we will use are `f`-*algebras*, which are simply functions of type $\texttt{f a} \to \texttt{a}$, and `f`-*coalgebras*, which are functions of type $\texttt{a} \to \texttt{f a}$ for some functor `f`. In both cases, type `a` is called the *carrier* of the (co)algebra.

**Fixed Points** Given a functor `f`, a fixed point is a type `p` such that `p` is in bijection with `f p`. Under some mild conditions, functor `f` has a least fixed point $\mu\ \texttt{f}$, which is implemented as follows in Haskell,

$$\textbf{data}\ \mu\ \texttt{f} = \texttt{In}\ (\texttt{f}\ (\mu\ \texttt{f}))$$

and a greatest fixed point $\nu\ \texttt{f}$ implemented as follows in Haskell:

$$\textbf{data}\ \nu\ \texttt{f} = \texttt{Out}^{\circ}\ (\texttt{f}\ (\nu\ \texttt{f}))$$

In general, elements of $\mu\ \texttt{f}$ are *finite* trees whose branching structure is characterised by `f`, and elements of $\nu\ \texttt{f}$ are *possibly infinite* trees whose branching structure is characterised by `f`. In Haskell, least and greatest fixed points are conflated, as we can see above that they have isomorphic Haskell implementation. However, we carefully maintain the distinction as it brings more safety: a function taking a $\mu\ \texttt{f}$ as argument is only expected to work on finite input, and a function returning a $\nu\ \texttt{f}$ indicates that its output may be output. Elements of $\nu\ \texttt{f}$ cannot be cast into $\mu\ \texttt{f}$ without arguing correctness. In this way, all recursion schemes defined in this paper is *total* — they guarantee to terminate and return a value for all possible values of its input.

**Example 1.** Let `ListF` and `TreeF` be two functors (with the evident `fmap` that can be derived by `GHC` automatically)

$$\textbf{data}\ \texttt{ListF a x} = \texttt{Nil} \mid \texttt{Cons a x}\quad \textbf{deriving}\ \texttt{Functor}$$
$$\textbf{data}\ \texttt{TreeF a x} = \texttt{Empty} \mid \texttt{Node x a x}\ \textbf{deriving}\ \texttt{Functor}$$

The type $\mu$ (`ListF a`) represents finite lists of `a` elements and $\mu$ (`TreeF a`) represents finite binary trees carrying `a` elements. Correspondingly, $\nu$ (`ListF a`) and $\nu$ (`TreeF a`) are possibly finite lists and trees respectively.

As an example, the correspondence between $\mu$ (`ListF a`) and finite elements of [`a`] is evidenced by the following bijection.

$$\begin{aligned}
&\texttt{conv}_{\mu} :: [\texttt{a}] \to \mu\ (\texttt{ListF a}) \\
&\texttt{conv}_{\mu}\ [\,] = \texttt{In Nil} \\
&\texttt{conv}_{\mu}\ (\texttt{a} : \texttt{as}) = \texttt{In}\ (\texttt{Cons a}\ (\texttt{conv}_{\mu}\ \texttt{as})) \\
&\texttt{conv}_{\mu}^{\circ} :: \mu\ (\texttt{ListF a}) \to [\texttt{a}]
\end{aligned}$$

$$\begin{aligned}\mathtt{conv}^{\circ}_{\mu}\ (\mathtt{In}\ \mathtt{Nil}) &= [\,]\\ \mathtt{conv}^{\circ}_{\mu}\ (\mathtt{In}\ (\mathtt{Cons}\ \mathtt{a}\ \mathtt{as})) &= \mathtt{a} : \mathtt{conv}^{\circ}_{\mu}\ \mathtt{as}\end{aligned}$$

Supposing that there is a function computing the length of a list,

$$\mathtt{length} :: \mu\ (\mathtt{ListF}\ \mathtt{a}) \to \mathtt{Integer}$$

it is illegal to pass a value of $\nu\ (\mathtt{ListF}\ \mathtt{a})$ to this function.

**Initial and Final (Co)Algebra** The constructor $\mathtt{In} :: \mathtt{f}\ (\mu\ \mathtt{f}) \to \mu\ \mathtt{f}$ is an $\mathtt{f}$-algebra with carrier $\mu\ \mathtt{f}$, and the inverse of $\mathtt{In}$

$$\begin{aligned}&\mathtt{in}^{\circ} :: \mu\ \mathtt{f} \to \mathtt{f}\ (\mu\ \mathtt{f})\\ &\mathtt{in}^{\circ}\ (\mathtt{In}\ \mathtt{x}) = \mathtt{x}\end{aligned}$$

is an $\mathtt{f}$-coalgebra. Conversely, the constructor $\mathtt{Out}^{\circ} :: \mathtt{f}\ (\nu\ \mathtt{f}) \to \nu\ \mathtt{f}$ is an $\mathtt{f}$-algebra with carrier $\nu\ \mathtt{f}$, and its inverse

$$\begin{aligned}&\mathtt{out} :: \nu\ \mathtt{f} \to \mathtt{f}\ (\nu\ \mathtt{f})\\ &\mathtt{out}\ (\mathtt{Out}^{\circ}\ \mathtt{x}) = \mathtt{x}\end{aligned}$$

is an $\mathtt{f}$-coalgebra with carrier $\nu\ \mathtt{f}$. Furthermore, $\mathtt{In}$ is called the *initial algebra* of $\mathtt{f}$ because it satisfies that for any $\mathtt{f}$-algebra $\mathtt{alg} :: \mathtt{f}\ \mathtt{a} \to \mathtt{a}$, there is exactly one function $\mathtt{h} :: \mu\ \mathtt{f} \to \mathtt{a}$ such that

$$\mathtt{h} \circ \mathtt{In} = \mathtt{alg} \circ \mathtt{fmap}\ \mathtt{h} \tag{1}$$

which was first proved by Adámek (1974) in a categorical setting. Conversely, $\mathtt{out}$ is called the *final coalgebra* of $\mathtt{f}$ because it satisfies that for any $\mathtt{f}$-coalgebra $\mathtt{coalg} :: \mathtt{c} \to \mathtt{f}\ \mathtt{c}$, there is exactly one function $\mathtt{h} :: \mathtt{c} \to \nu\ \mathtt{f}$ such that

$$\mathtt{out} \circ \mathtt{h} = \mathtt{fmap}\ \mathtt{h} \circ \mathtt{coalg} \tag{2}$$

These two properties underlie the two fundamental recursion schemes, the catamorphism and the anamorphism, that we will introduce in the next section.

# 3 Fundamental Recursion Schemes

Most if not all programs are about processing data, and as Hoare (1972) noted, 'there are certain close analogies between the methods used for structuring data and the methods for structuring a program which processes that data.' In essence, *data structure determines program structure*. The determination is abstracted as recursion schemes for programs processing recursive datatypes.

In this section, we investigate the three fundamental recursion schemes — the catamorphism, in which the program is structured by its input, the anamorphisms, in which the program is structured by its output, and the hylomorphism, in which the program is structured by an internal recursive call structure.

## 3.1 Catamorphisms

We start our journey with programs whose structure follows their input. As a motivating example, consider the program computing the length of a list:

```
length :: [a]     → Integer
length []         = 0
length (x : xs)   = 1 + length xs
```

In Haskell, a list is either the empty list `[]` or `x : xs`, an element `x` prepended to list `xs`. This structure of lists is closely followed by the program `length`, which is defined by two cases too, one for the empty list `[]` and one for the recursive case `x:xs`. Additionally, in the recursive case `length (x:xs)` is solely determined by `length xs` without further information about `xs`.

**List Folds** The pattern in `length` is called *structural recursion* and is expressed by the function `foldr` in Haskell:

```
foldr :: (a → b → b) → b → [a] → b
foldr f e []       = e
foldr f e (x : xs) = f x (foldr f e xs)
```

which is very familiar to functional programmers and is ubiquitous in list processing. As a fold, `length = foldr (λ_ l → 1 + l) 0`. The frequently used function `map` is also a fold:

```
map :: (a → b) → [a] → [b]
map f = foldr (λx xs → f x : xs) []
```

Another example is the function flattening a list of lists into a list:

```
concat :: [[a]] → [a]
concat = foldr (++) []
```

By expressing functions as folds, their structures that are obscured by general recursion become clearer, similar in spirit to the well accepted practice of structuring programs with if-conditionals and for-/while-loops in imperative languages.

**Definition 1** (`cata`)**.** Folds on lists can be readily generalised to the generic setting, where the shape of the datatype is determined by some functor (Malcolm, 1990b; Fokkinga, 1992; Hagino, 1987). The resulting recursion scheme is called the *catamorphism*:

```
cata :: Functor f ⇒ (f a → a) → μ f → a
cata alg = alg ∘ fmap (cata alg) ∘ in°
```

Another popular notation for `cata alg` is the so-called banana bracket `(| alg |)` introduced by Meijer *et al.* (1991), but we will not use this style of notations in this paper as we have too many recursion schemes to present with various squiggly brackets.

Informally, the catamorphism gradually breaks down its inductively defined argument, replacing constructors with the given algebra `alg`. The name `cata` dubbed by Meertens (1988) is from Greek κατά meaning 'downwards along' or 'according to'. Catamorphisms enjoy an important *universal property* as follows.

**Proposition 1.** *For any* $\texttt{alg} :: \texttt{f a} \to \texttt{a}$, $\texttt{cata alg}$ *is the unique function that satisfies the equation*

$$\texttt{x} \circ \texttt{In} = \texttt{alg} \circ \texttt{fmap x} \tag{3}$$

*where both sides have type* $\texttt{f}\ (\mu\ \texttt{f}) \to \texttt{a}$ *and the unknown* $\texttt{x}$ *has type* $\mu\ \texttt{f} \to \texttt{a}$.

The function `cata` satisfies Equation (3) by definition, and by the property (1) of initial algebra, it is the unique solution. This property is vital for equational reasoning about catamorphisms, which we discuss in Section 11.

**Example 2.** We can recover `foldr` from `cata` by

```
foldr' :: (a → b → b) → b → [a] → b
foldr' f e = cata alg ∘ conv_μ where
  alg Nil       = e
  alg (Cons a x) = f a x
```

and now we can fold more datatypes, such as binary trees:

```
data TreeF e a = Empty | Node a e a deriving Functor
size :: μ (TreeF e) → Integer
size = cata alg where
  alg Empty = 0
  alg (Node l e r) = l + 1 + r
```

**Example 3** (Interpreting Languages)**.** More evidence of the expressiveness of `cata` comes from its application in giving semantics to programming languages. The semantics of a language is *compositional* if the meaning of a composite program is solely determined the meaning of its constitutes — exactly the pattern of a `cata`. As an example, consider a mini language of mutable memory consisting of three language constructs: `Put (i, x) k` writes value `x` to memory cell of address `i` and then executes program `k`; `Get i k` reads memory cell `i`, letting the result be `s`, and then executes program `k s`; and `Ret a` terminates the execution with return value `a`. The syntax of the language can be modelled as $\mu$ (`ProgF s a`):

```
data ProgF s a x = Ret a | Put (Int, s) x
                 | Get Int (s → x) deriving Functor
```

where `s` is the type of values stored by memory cells and `a` is the type of values finally returned. An example of a program in this language is

```
p1 :: μ (ProgF Int Int)
p1 = In (Get 0 (λs → (In (Put (0, s + 1) (In (Ret s))))))
```

which reads the 0-th cell, increments it, and returns the old value. The syntax is admittedly clumsy because of the repeating `In` constructors, but they can be eliminated if 'smart constructors' such as `ret = In ∘ Ret` are defined.

The semantics of a program in this mini language can be given as a value of type `Map Int s → a`, and the interpretation is a catamorphisms:

```
interp :: μ (ProgF s a) → (Map Int s → a)
interp = cata handle where
  handle (Ret a)       = λ_ → a
  handle (Put (i, x) k) = λm → k (update m i x)
  handle (Get i k)     = λm → k (m ! i) m
```

where `update m i x` is the map `m` with the value at `i` changed to `x`, and `m ! i` looks up `i` in `m`. Then we can use it to interpret programs:

```
*> interp p1 (fromList [(0, 100)])
100
```

## 3.2 Anamorphisms

In catamorphisms, the structure of a program mimics the structure of the input. Needless to say, this pattern is insufficient to cover all programs in the wild. Imagine a program returning a record:

```
data Person = Person
  {name :: String, addr :: String, phone :: [Int]}
mkEntry :: StaffInfo → Person
mkEntry i = Person n a p where
  n = ...; a = ...; p = ...
```

The structure of the program more resembles the structure of its output — each field of the output is computed by a corresponding part of the program. Similarly, when the output is a recursive datatype, a natural pattern is that the program generates the output recursively, sometimes called *(structural) corecursion*. Consider the following program generating evenly spaced numbers over an interval.

```
linspace :: RealFrac a ⇒ a → a → Integer → [a]
linspace s e n = gen s where
  step = (e − s) / fromIntegral (n + 1)
  gen i
    | i < e       = i : gen (i + step)
    | otherwise = []
```

The program `gen` does not mirror the structure of its numeric input at all, but it follows the structure of its output, which is a list: for the two cases of a list, `[]` and `(:)`, `gen` has a corresponding branch generating it.

**List Unfolds** The pattern of generating a list in the example above is abstracted as the Haskell function `unfoldr`:

```
unfoldr :: (b → Maybe (a, b)) → b → [a]
unfoldr g s = case g s of
  (Just (a, s′)) → a : unfoldr g s′
  Nothing        → []
```

in which `g` either produces `Nothing` indicating the end of the output or produces from a seed `s` the next element `a` of the output together with a new seed `s′` for generating the rest of the output. Thus we can rewrite `linspace` as

```
linspace s e n = unfoldr gen s where
  step = (e − s) / fromIntegral (n + 1)
  gen i = if i < e then Just (i, i + step) else Nothing
```

Note that the list produced by `unfoldr` is not necessarily finite. For example,

```
from :: Integer → [Integer]
from = unfoldr (λn → Just (n, n + 1))
```

generates the infinite list of all integers from `n`.

In the same way that `cata` generalises `foldr`, `unfoldr` can be generalised from lists to arbitrary recursive datatypes whose shape is characterised by a functor:

**Definition 2** (`ana`)**.** The *anamorphism* is the following recursion scheme:

```
ana :: Functor f ⇒ (c → f c) → c → ν f
ana coalg = Out° ∘ fmap (ana coalg) ∘ coalg
```

The name is due to Meijer *et al.* (1991) (*ana* from the Greek preposition ανά meaning 'upwards', dual to cata meaning 'downwards'). The return type is the final coalgebra since the generation may never end. Anamorphisms also enjoy a universal property dual to the one for `cata`:

**Proposition 2.** *For any* `coalg :: c → f c`, `ana coalg` *is the unique function that satisfies the equation*

$$\texttt{out} \circ \texttt{x} = \texttt{fmap x} \circ \texttt{coalg} \tag{4}$$

*where both sides have type* `c → f (ν f)` *and the unknown* `x` *has type* `c → ν f`.

By definition, `ana coalg` satisfies and it is the unique solution following the property (2) of the final algebra `ν f`.

**Example 4.** Modulo the isomorphism between `[a]` and `ν (ListF a)`, `unfoldr` is an anamorphism:

```
unfoldr′ :: (b → Maybe (a, b)) → b → [a]
unfoldr′ g = conv°_μ ∘ ana coalg where
  coalg b = case g b of
    Nothing      → Nil
    (Just (a, b)) → Cons a b
```

**Example 5.** An instructive example of anamorphisms is merging a pair of ordered lists:

```
merge :: Ord a ⇒ (ν (ListF a), ν (ListF a)) → ν (ListF a)
merge = ana c where
  c (x, y)
    | null_ν x ∧ null_ν y = Nil
    | null_ν y ∨ head_ν x < head_ν y
                = Cons (head_ν x) (tail_ν x, y)
    | otherwise = Cons (head_ν y) (x, tail_ν y)
```

where $\texttt{null}_\nu$, $\texttt{head}_\nu$ and $\texttt{tail}_\nu$ are the corresponding list processing functions for $\nu$ (ListF a).

## 3.3 Hylomorphisms

Catamorphisms consume data and anamorphisms produce data, but some algorithms are more complex than playing a single role — they produce and consume data at the same time. Taking the quicksort algorithm for example, a (not-in-place, worst complexity $\mathcal{O}(n^2)$) implementation is:

```
qsort :: Ord a ⇒ [a] → [a]
qsort []       = []
qsort (a : as) = qsort l ++ [a] ++ qsort r where
  l = [b | b ← as, b < a]
  r = [b | b ← as, b ⩾ a]
```

Although the input type [a] is a recursive datatype, qsort is not a catamorphism as the recursion is not performed on the immediate substructure as. Neither is it an anamorphism, because the output is not produced in the head-and-recursion manner.

Felleisen *et al.* (2018) referred to this form of recursive programs as *generative recursion* because the input a : as generates a set of subproblems, namely l and r, which are recursively solved and their solutions are combined to solve the overall problem a : as. This structure of computing qsort is made manifest in the following rewrite of qsort:

```
qsort′ :: Ord a ⇒ [a] → [a]
qsort′ = combine ∘ fmap qsort′ ∘ partition
partition :: Ord a ⇒ [a] → TreeF a [a]
```

```
partition []       = Empty
partition (a : as) = Node [b | b ← as, b < a]
                          a
                          [b | b ← as, b ⩾ a]
combine :: TreeF a [a] → [a]
combine Empty        = []
combine (Node l x r) = l ++ [x] ++ r
```

The functor `TreeF e a = Empty | Node a e a` governs the recursive call structure, which is a binary tree. The (`TreeF a`)-coalgebra `partition` divides a problem (if not trivial) into two subproblems, and the (`TreeF a`)-algebra `combine` concatenates the results of subproblems to form a solution to the whole problem.

**Definition 3** (`hylo`)**.** We can abstract the pattern of divide-and-conquer algorithms like `qsort` as a recursion scheme called the *hylomorphism*:

```
hylo :: Functor f ⇒ (f a → a) → (c → f c) → c → a
hylo a c = a ∘ fmap (hylo a c) ∘ c
```

The name is due to Meijer *et al.* (1991) and is a term from the Aristotelian philosophy that objects are compounded of matter and form, where hylo- (Greek ὕλη-) means 'matter'.

The hylomorphism is an expressive recursion pattern. It subsumes catamorphisms because for all `alg :: f a → a`,

$$\texttt{cata alg} = \texttt{hylo alg in}^{\circ}$$

and it also subsumes anamorphisms: for all `coalg :: c → f c`,

$$\texttt{ana coalg} = \texttt{hylo Out}^{\circ}\ \texttt{coalg}$$

However, the expressiveness of `hylo` comes at a cost: even when both `alg::f a → a` and `coalg :: c → f c` are total functions, `hylo alg coalg` may not be total (in contrast, `cata alg` and `ana coalg` are always total whenever `alg` and `coalg` are), let alone have a universal property. Informally, it is because the coalgebra `coalg` may infinitely generate subproblems while the algebra `alg` may require all subproblems solved to solve the whole problem.

**Example 6.** As an instance of the problematic situation, consider a coalgebra

```
geo :: Integer → ListF Double Integer
geo n = Cons (1 / fromIntegral n) (2 ∗ n)
```

which generates the geometric sequence $[\frac{1}{n},\ \frac{1}{2n},\ \frac{1}{4n},\ \frac{1}{8n},\ ..]$, and an algebra

```
sum :: ListF Double Double → Double
sum Nil        = 0
sum (Cons n p) = n + p
```

which sums a sequence. Both `geo` and `sum` are total Haskell functions, but the function `zeno = hylo sum geo` is not total, as it diverges for all input `i :: Integer`. (It does not mean that Achilles can never overtake the tortoise — `zeno` diverges because it really tries to add up an infinite sequence rather than taking the limit.)

**Recursive Coalgebras** The loss of totality is compensated by *recursive coalgebras*, which are coalgebras $\texttt{coalg} :: \texttt{c} \to \texttt{f c}$ that for any algebra $\texttt{alg} :: \texttt{f a} \to \texttt{a}$, there is a unique function $\texttt{x} :: \texttt{c} \to \texttt{a}$ satisfying the hylo equation

$$\texttt{x} = \texttt{alg} \circ \texttt{fmap x} \circ \texttt{coalg}.$$

A canonical recursive coalgebra is $\texttt{in}^\circ :: \mu\ \texttt{f} \to \texttt{f}\ (\mu\ \texttt{f})$ since the universal property of catamorphisms (Eq. 3) guarantees the unique existence of a total function `cata alg` satisfying $\texttt{x} = \texttt{alg} \circ \texttt{fmap x} \circ \texttt{in}^\circ$. Dually, a *corecursive algebra* is an algebra that leads to a unique solution to the hylo equation for any coalgebra, and $\texttt{Out}^\circ$ is corecursive by the universal property of `ana` (Eq. 3.2).

More recursive coalgebras can be obtained from comonads (Capretta *et al.*, 2006) and more generally adjunctions with conjugate natural transformations (Hinze *et al.*, 2015). Theoretically, all recursion schemes presented in this paper are a hylomorphism in some category and can be obtained in the framework of Hinze *et al.* (2015), but the details of the unification is beyond this paper.

**Example 7.** The aforementioned coalgebra `partition` is recursive when restricting its domain to finite lists. This can be proved by an easy inductive argument: For any total $\texttt{alg} :: \texttt{f a} \to \texttt{a}$, suppose that total function `x` satisfies

$$\texttt{x} = \texttt{alg} \circ \texttt{fmap x} \circ \texttt{partition},$$

Given any finite list `l`, we show `x l` is uniquely determined by `alg`. For the base case $\texttt{l} = [\,]$,

$$\begin{aligned}\texttt{x}\ [\,] &= \texttt{alg}\ (\texttt{fmap x}\ (\texttt{partition}\ [\,])) \\ &= \texttt{alg}\ (\texttt{fmap x}\ \texttt{Empty}) \\ &= \texttt{alg Empty}\end{aligned}$$

For the inductive case $\texttt{l} = \texttt{y} : \texttt{ys}$, by the definition of `partition`,

$$\begin{aligned}\texttt{x}\ (\texttt{y} : \texttt{ys}) &= \texttt{alg}\ (\texttt{fmap x}\ (\texttt{partition}\ (\texttt{y} : \texttt{ys}))) \\ &= \texttt{alg}\ (\texttt{fmap x}\ (\texttt{Node ls a rs})) \\ &= \texttt{alg}\ (\texttt{Node}\ (\texttt{x ls})\ \texttt{a}\ (\texttt{x rs}))\end{aligned}$$

where $\texttt{ls} = [\texttt{b} \mid \texttt{b} \leftarrow \texttt{as}, \texttt{b} < \texttt{a}]$ and $\texttt{rs} = [\texttt{b} \mid \texttt{b} \leftarrow \texttt{as}, \texttt{b} \geqslant \texttt{a}]$ are strictly smaller than $\texttt{l} = (\texttt{a} : \texttt{as})$, and thus `x ls` and `x rs` are uniquely determined by `alg`. Consequently, $\texttt{x}\ (\texttt{y} : \texttt{ys})$ is uniquely determined by `alg`. Thus we conclude that `x` satisfying the hylo equation is unique, and the existence of `x` can be argued in the same way.

**Metamorphisms** If we separate the producing and consuming phases of a hylomorphism `hylo alg coalg` where `coalg` is recursive, we have the following equations (both follow the uniqueness of the solution to hylo equations with *recursive* coalgebra `coalg`)

$$\begin{aligned}\texttt{hylo alg coalg} &= \texttt{cata alg} \circ \texttt{hylo In coalg} \\ &= \texttt{cast alg} \circ \nu 2\mu \circ \texttt{ana coalg}\end{aligned}$$

where $\nu 2\mu = \texttt{hylo In out} :: \nu\ \texttt{f} \to \mu\ \texttt{f}$ is the *partial* function that converts the finite subset of a coinductive datatype into its inductive counterpart. Thus loosely speaking (or precisely speaking in a category where $\nu$ and $\mu$ coincide), a `hylo` is a `cata` after an `ana`. The opposite direction of composition can also be considered:

```
meta :: (Functor f, Functor g) ⇒ (c → g c) → (f c → c) → μ f → ν g
meta coalg alg = ana coalg ∘ cata alg
```

which is called *metamorphisms* by Gibbons (2007) because it *metamorphoses* data represented by functor `f` to `g`. Unlike in hylomorphisms, the producing and consuming phases in a metamorphism cannot be straightforwardly fused into a single recursive process. Gibbons (2007, 2019) gave conditions for doing this when `f` is `ListF`, but it is beyond the scope of this paper.

# 4 Accumulations

Accumulating parameters are a well known technique for optimising recursive functions. The canonical example is optimising the following `reverse` function that runs in quadratic time

```
reverse :: [a]      → [a]
reverse  []         = []
reverse  (x : xs) = reverse xs ++ [x]
```

by generalising the function to a version with an additional parameter — an accumulating parameter:

```
revCat :: [a] → [a] → [a]
revCat ys []       = ys
revCat ys (x : xs) = revCat (x : ys) xs
```

which specialises to `reverse` by `recCat []` and it runs in linear time. This pattern of scanning a list from left to right and accumulating an argument is abstracted as the Haskell function `foldl`:

```
foldl :: (b → a → b) → b → [a] → b
foldl f e []       = e
foldl f e (x : xs) = foldl f (f e x) xs
```

which instantiates to `revCat` for $\mathtt{f} = \lambda \mathtt{ys}\ \mathtt{x} \to \mathtt{x} : \mathtt{ys}$.

Similar to `cata`, computing `foldl` follows the structure of the list — a base case for empty list $[\,]$ is defined and an inductive case for $\mathtt{x} : \mathtt{xs}$ is defined. What differs is that `foldl` has an argument `e` varied during the recursion. However, if we reorder the arguments of `foldl` and view `foldl` as computing a $\mathtt{b} \to \mathtt{b}$ function from a function of $\mathtt{b} \to \mathtt{a} \to \mathtt{b}$ and a list of $[\mathtt{a}]$, then it becomes a catamorphism of functions:

```
foldl' :: ∀a b.(b → a → b) → [a] → (b → b)
foldl' f = foldr alg id where
  alg :: a → (b → b) → (b → b)
  alg a g = λb → g (f b a)
```

(For `GHC`, the `ScopedTypeVariables` language option is needed here to let the type `a` and `b` in the signature of `alg` refer to the `a` and `b` in the signature of `foldl'`.) It can be shown that `foldl' f xs e` = `foldl f e xs` for any `f`, `e` and `xs`.

**Definition 4** (`accu`)**.** We can further generalise this pattern to the generic setting and allow the parameter type and result type to differ. This is called the recursion scheme of *accumulations* (Hu *et al.*, 1999; Pardo, 2002; Gibbons, 2000):

```
accu :: Functor f ⇒ (f (b → a) → b → a) → (μ f, b) → a
accu alg = uncurry (cata alg)
```

In `accu`, the algebra `alg` takes two arguments: the second one of type `b` is clearly the accumulating parameter, and the first one of type $\mathtt{f}\ (\mathtt{b} \to \mathtt{a})$ can be understood as 'continuations' of accumulating along branches of `f`. Although an `accu` is simply a curried `cata` for function type $\mathtt{b} \to \mathtt{a}$, by giving it a name, the intention that the result is a function taking an accumulating parameter is explicitly expressed.

**Example 8.** An example of `accu` is the program that accumulates from root to leaves in a binary tree labelled with integers, and results in a tree in which each node is labelled with the sum of the path from the root to the node in the original tree:

```
sumPath :: μ (TreeF Integer) → μ (TreeF Integer)
sumPath t = accu alg (t, 0) where
  alg Empty        _ = In Empty
  alg (Node l e r) s = let s' = s + e
                       in In (Node (l s') s' (r s'))
```

In the unifying theory of recursion schemes by means of adjunction (Hinze, 2013; Hinze & Wu, 2016; Hinze *et al.*, 2015), `accu` is an adjoint fold from the

curry adjunction $- \times b \dashv (-)^b$. That is to say, `accu alg`$::(\mu\ \mathtt{f}, \mathtt{b}) \to \mathtt{a}$ is the (left-)adjunct of a catamorphism of type $\mu\ \mathtt{f} \to (\mathtt{b} \to \mathtt{a})$ via the curry adjunction. The same adjunction gives rise to a dual recursion scheme — unfolding a seed that is a pair type:

```
coaccu :: Functor f ⇒ ((c, p) → f (c, p)) → c → p → ν f
coaccu coalg = curry (ana coalg)
```

of which an example is the `merge` function already appeared in Section 3.2.

# 5 Mutual Recursion

This section is about *mutual recursion* in two forms: mutually recursive functions and mutually recursive datatypes. The former is abstracted as a recursion scheme called mutumorphisms, and we will discuss its categorical dual, which turns out to be corecursion generating mutually recursive datatypes.

## 5.1 Mutumorphisms

In Haskell, function definitions can not only be recursive but also be *mutually* recursive — two or more functions are defined in terms of each other. The simplest example is `isOdd` and `isEven` determining the parity of a natural number:

```
data NatF a = Zero | Succ a deriving Functor
type Nat = μ NatF

isEven :: Nat → Bool
isEven (In Zero) = True
isEven (In (Succ n)) = isOdd n

isOdd :: Nat → Bool
isOdd (In Zero) = False
isOdd (In (Succ n)) = isEven n
```

Here we are using an inductive definition of natural numbers: `Zero` is a natural number and `Succ n` is a natural number whenever `n` is. Both `isEven` and `isOdd` are very much like a catamorphism: they have a non-recursive definition for the base case `Zero`, and a recursive definition for the inductive case `Succ n` in terms of the substructure `n`, except that their recursive definitions depend on the recursive result for `n` of the other function, instead of their own, making them not a catamorphism.

A more interesting example exploiting mutual recursion is the following way computing Fibonacci number $F_i$ (i.e. $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geqslant 2$):

```
fib :: Nat → Integer
fib (In Zero) = 0
fib (In (Succ n)) = fib n + aux n
```

$$
\begin{aligned}
&\texttt{aux} :: \texttt{Nat} \to \texttt{Integer} \\
&\texttt{aux}\ (\texttt{In}\ \texttt{Zero}) = 1 \\
&\texttt{aux}\ (\texttt{In}\ (\texttt{Succ}\ \texttt{n})) = \texttt{fib}\ \texttt{n}
\end{aligned}
$$

The function $\texttt{aux}\ \texttt{n}$ is defined to be equal to the $(\texttt{n}-1)$-th Fibonacci number $F_{n-1}$ for $\texttt{n} \geqslant 1$, and $\texttt{aux}\ 0$ is chosen to be $F_1 - F_0 = 1$. Consequently, $\texttt{fib}\ 0 = F_0$,

$$\texttt{fib}\ 1 = \texttt{fib}\ 0 + \texttt{aux}\ 1 = F_0 + (F_1 - F_0) = F_1,$$

and $\texttt{fib}\ \texttt{n} = \texttt{fib}\ (\texttt{n}-1) + \texttt{fib}\ (\texttt{n}-2)$ for $n >= 2$, which matches the definition of Fibonacci sequence.

**Well-definedness** The recursive definitions of the examples above are well-defined, in the sense that there is a unique solution to each group of recursive definitions regarded as a system of equations. Imagine a story of creating functions: In the first day, the images of `Zero` are uniquely determined for both functions:

$$\langle \texttt{fib}\ 0, \texttt{aux}\ 0 \rangle = \langle 0, 1 \rangle$$

In the second day, the images of `Succ Zero` are uniquely determined for both too according to the recursive cases of the defining equations,

$$
\begin{aligned}
\langle \texttt{fib}\ 0, \texttt{aux}\ 1 \rangle &= \langle 0, 1 \rangle \\
\langle \texttt{fib}\ 1, \texttt{aux}\ 1 \rangle &= \langle 1, 0 \rangle
\end{aligned}
$$

and so on for all natural numbers:

$$
\begin{aligned}
\langle \texttt{fib}\ 0, \texttt{aux}\ 1 \rangle &= \langle 0, 1 \rangle \\
\langle \texttt{fib}\ 1, \texttt{aux}\ 1 \rangle &= \langle 1, 0 \rangle \\
\langle \texttt{fib}\ 2, \texttt{aux}\ 2 \rangle &= \langle 1, 1 \rangle \\
&\cdots
\end{aligned}
$$

The same line of reasoning applies too when we generalise this pattern to a recursion scheme capturing mutual structural recursion:

**Definition 5.** The *mutumorphism* (Fokkinga, 1992) is the following recursion scheme:

$$
\begin{aligned}
&\texttt{mutu} :: \texttt{Functor}\ \texttt{f} \Rightarrow (\texttt{f}\ (\texttt{a}, \texttt{b}) \to \texttt{a}) \to (\texttt{f}\ (\texttt{a}, \texttt{b}) \to \texttt{b}) \\
&\qquad\qquad \to (\mu\ \texttt{f} \to \texttt{a}, \mu\ \texttt{f} \to \texttt{b}) \\
&\texttt{mutu}\ \texttt{alg}_1\ \texttt{alg}_2 = (\texttt{fst} \circ \texttt{cata}\ \texttt{alg}, \texttt{snd} \circ \texttt{cata}\ \texttt{alg}) \\
&\quad \textbf{where}\ \texttt{alg}\ \texttt{x} = (\texttt{alg}_1\ \texttt{x}, \texttt{alg}_2\ \texttt{x})
\end{aligned}
$$

in which $\texttt{alg}_1$ and $\texttt{alg}_2$ are provided with the images of substructures under both functions being defined and are respectively responsible for computing the images of the whole structure under the two functions being defined.

The name *mutumorphism* is a bit special in the dictionary of Greek morphisms, as the prefix mutu- is from Latin rather than Greek.

**Example 9.** With `mutu`, our Fibonacci number example can be concisely expressed as:

$$
\begin{aligned}
&\mathtt{fib}' = \mathtt{fst}\ (\mathtt{mutu}\ \mathtt{f}\ \mathtt{g})\ \mathbf{where}\\
&\quad \mathtt{f}\ \mathtt{Zero} = 0\\
&\quad \mathtt{f}\ (\mathtt{Succ}\ (\mathtt{n}, \mathtt{m})) = \mathtt{n} + \mathtt{m}\\
&\quad \mathtt{g}\ \mathtt{Zero} = 1\\
&\quad \mathtt{g}\ (\mathtt{Succ}\ (\mathtt{n}, \_)) = \mathtt{n}
\end{aligned}
$$

In the unifying theory of recursion schemes by means of adjunctions, a mutumorphism $\mathtt{mutu}\ \mathtt{alg}_1\ \mathtt{alg}_2 :: (\mu\ \mathtt{f} \to \mathtt{a}, \mu\ \mathtt{f} \to \mathtt{b})$ is the left-adjunct of a catamorphism of type $\mu\ \mathtt{f} \to (\mathtt{a}, \mathtt{b})$ via the adjunction $\Delta \dashv \times$ between the product category $C \times C$ and some base category $C$ (Hinze & Wu, 2016) (In the setting of this paper, $C = \mathbf{Set}$). The same adjunction also underlies a dual scheme discussed in the next subsection.

## 5.2 Dual of Mutumorphisms

As a mutumorphism is two or more mutually recursive functions folding one inductive datatype, we can consider its dual — unfolding a seed to two or more mutually-defined coinductive datatypes. An instructive example is recovering an expression from a Gödel number that encodes the expression. Consider the grammar of a simple family of arithmetic expressions:

$$
\begin{aligned}
&\mathbf{data}\ \mathtt{Expr} = \mathtt{Add}\ \mathtt{Expr}\ \mathtt{Term} \mid \mathtt{Minus}\ \mathtt{Expr}\ \mathtt{Term} \mid \mathtt{FromT}\ \mathtt{Term}\\
&\mathbf{data}\ \mathtt{Term} = \mathtt{Lit}\ \mathtt{Integer} \mid \mathtt{Neg}\ \mathtt{Term} \mid \mathtt{Paren}\ \mathtt{Expr}
\end{aligned}
$$

which is a pair of mutually-recursive datatypes. A Gödel numbering of this grammar *invertibly* maps an `Expr` or a `Term` to a natural number, for example:

$$
\begin{aligned}
\mathtt{g}\ (\mathtt{Add}\ \mathtt{e}\ \mathtt{t}) &= 2^{\mathtt{g}\ \mathtt{e}} * 3^{\mathtt{h}\ \mathtt{t}} & \mathtt{h}\ (\mathtt{Lit}\ \mathtt{n}) &= 2^{\mathtt{encLit}\ \mathtt{n}}\\
\mathtt{g}\ (\mathtt{Minus}\ \mathtt{e}\ \mathtt{t}) &= 5^{\mathtt{g}\ \mathtt{e}} * 7^{\mathtt{h}\ \mathtt{t}} & \mathtt{h}\ (\mathtt{Neg}\ \mathtt{t}) &= 3^{\mathtt{h}\ \mathtt{t}}\\
\mathtt{g}\ (\mathtt{FromT}\ \mathtt{t}) &= 11^{\mathtt{h}\ \mathtt{t}} & \mathtt{h}\ (\mathtt{Paren}\ \mathtt{e}) &= 5^{\mathtt{g}\ \mathtt{e}}
\end{aligned}
$$

where $\mathtt{encLit}\ \mathtt{n} = \mathbf{if}\ \mathtt{n} \geqslant 0\ \mathbf{then}\ 2 * \mathtt{n} + 1\ \mathbf{else}\ 2 * (-\mathtt{n})$ invertibly maps any integer to a positive integer. Although the encoding functions $g$ and $h$ clearly hint at a recursion scheme (of folding mutually-recursive datatypes to the same type), in this section we pay our attention to the opposite decoding direction:

$$
\begin{aligned}
&\mathtt{decE} :: \mathtt{Integer} \to \mathtt{Expr}\\
&\mathtt{decE}\ \mathtt{n} = \mathbf{let}\ (\mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_5, \mathtt{e}_1, \mathtt{e}_{11}) = \mathtt{factorise11}\ \mathtt{n}\\
&\quad \mathbf{in\ if}\ \mathtt{e}_2 > 0 \vee \mathtt{e}_3 > 0\ \mathbf{then}\ \mathtt{Add}\ (\mathtt{decE}\ \mathtt{e}_2)\ (\mathtt{decT}\ \mathtt{e}_3)\\
&\qquad \mathbf{else\ if}\ \mathtt{e}_5 > 0 \vee \mathtt{e}_1 > 0\\
&\qquad\quad \mathbf{then}\ \mathtt{Minus}\ (\mathtt{decE}\ \mathtt{e}_5)\ (\mathtt{decT}\ \mathtt{e}_1)\\
&\qquad\quad\ \mathbf{else}\ \mathtt{FromT}\ (\mathtt{decT}\ \mathtt{e}_{11})\\
&\mathtt{decT} :: \mathtt{Integer} \to \mathtt{Term}
\end{aligned}
$$

```
decT n = let (e2, e3, e5, _, _) = factorise11 n
  in if e2 > 0 then Lit (decLit e2)
    else if e3 > 0 then Neg (decT e3)
         else Paren (decE e5)
```

where `factorise11 n` computes the exponents for $2, 3, 5, 7$ and 11 in the prime factorisation of $n$, and `decLit` is the inverse of `encLit`. Functions `decT` and `decE` can correctly recover the encoded expression/term because of the fundamental theorem of arithmetic (i.e. the unique-prime-factorization theorem).

In the definitions of `decE` and `decT`, the choice of `decE` or `decT` when making a recursive call must match the type of the substructure at that position. It would be convenient, if the correct choice (of `decE` or `decT`) can be automatically made based on the types — we let a recursion scheme do the job.

For the generality of our recursion scheme, let us first generalise `Expr` and `Term` to an arbitrary pair of mutually recursive datatypes, which we model as fixed points of two bifunctors `f` and `g`. Likewise, the least fixed point models finite inductive data, and the greatest fixed point models possibly infinite coinductive data. Here we are interested in the latter:

```
newtype ν1 f g where
  Out1° :: f (ν1 f g) (ν2 f g) → ν1 f g
newtype ν2 f g where
  Out2° :: g (ν1 f g) (ν2 f g) → ν2 f g
```

For instance, let

```
data ExprF e t = Add' e t | Minus' e t | FromT' t
data TermF e t = Lit' Int | Neg' t | Paren' e
```

then `Expr` is isomorphic to $\nu_1$ `ExprF TermF` and `Term` is isomorphic to $\nu_2$ `ExprF TermF`.

**Definition 6** (`comutu`)**.** Now we can define a recursion scheme that generates a pair of mutually recursive datatypes from a single seed:

```
comutu :: (Bifunctor f, Bifunctor g)
        ⇒ (c → f c c) → (c → g c c) → c → (ν1 f g, ν2 f g)
comutu c1 c2 s = (x s, y s) where
  x = Out1° ∘ bimap x y ∘ c1
  y = Out2° ∘ bimap x y ∘ c2
```

which remains unnamed in the literature.

**Example 10.** The `comutu` scheme renders our decoding example to become

```
decExprTerm :: Integer → (ν1 ExprF TermF, ν2 ExprF TermF)
decExprTerm = comutu genExpr genTerm

genExpr :: Integer → ExprF Integer Integer
genExpr n =
```

```
  let (e2, e3, e5, e1, e11) = factorise11 n
  in if e2 > 0 ∨ e3 > 0 then Add′ e2 e3
    else if e5 > 0 ∨ e1 > 0
           then Minus′ e5 e1 else FromT′ e11
genTerm :: Integer → TermF Integer Integer
genTerm n =
  let (e2, e3, e5, _, _) = factorise11 n
  in if e2 > 0 then Lit′ (decLit e2)
    else if e3 > 0 then Neg′ e3 else Paren′ e5
```

Comparing to the direct definitions of `decE` and `decT`, `genTerm` and `genExpr` are simpler as they just generate a new seed for each recursive position and recursive calls of the correct type is invoked by the recursion scheme `comutu`.

Theoretically, `comutu` is the adjoint unfold from the adjunction $\Delta \dashv \times$: `comutu c1 c2 :: c → (ν1 f g, ν2 f g)` is the right-adjunct of an anamorphism of type `(c → ν1 f g, c → ν2 f g)` in the product category $C \times C$. A closely related adjunction $+ \dashv \Delta$ also gives two recursion schemes for mutual recursion. One is an adjoint fold that consumes mutually recursive datatypes, of which an example is the encoding function of Gödel numbering discussed above, and dually an adjoint unfold that generates $\nu$ `f` from seed `Either c1 c2`, which captures *mutual corecursion*. Although attractive and practically important, we forgo an exhibition of these two recursion schemes here.

# 6 Primitive (Co)Recursion

In this section, we investigate the pattern in recursive programs in which the original input is directly involved besides the recursively computed results, resulting in a generalisation of the catamorphism — the *paramorphism*. We also discuss its generalisation the *zygomorphism* and its categorical dual the *apomorphism*.

## 6.1 Paramorphisms

A wide family of recursive functions that are not directly covered by catamorphisms are those in whose definitions the original substructures are directly used in addition to their images under the function being defined. An example is one of the most frequently demonstrated recursive function `factorial`:

```
factorial :: Nat → Nat
factorial (In Zero)     = 1
factorial (In (Succ n)) = In (Succ n) ∗ factorial n
```

In the second case, besides the recursively computed result `factorial n`, the substructure `n` itself is also used, but it is not directly provided by `cata`. A slightly more practical example is counting the number of words (more accurately, maximal sub-sequences of non-space characters) in a list of characters:

$$
\begin{aligned}
&\texttt{wc} :: \mu\ (\texttt{ListF Char}) \to \texttt{Integer}\\
&\texttt{wc}\ (\texttt{In Nil}) &&= 0\\
&\texttt{wc}\ (\texttt{In}\ (\texttt{Cons c cs})) &&= \textbf{if}\ \texttt{isNewWord}\ \textbf{then}\ \texttt{wc cs} + 1\ \textbf{else}\ \texttt{wc cs}\\
&\quad \textbf{where}\ \texttt{isNewWord} &&= \neg\ (\texttt{isSpace c}) \wedge (\texttt{null cs} \vee \texttt{isSpace}\ (\texttt{head cs}))
\end{aligned}
$$

Again in the second case, `cs` is used besides `wc cs`, making it not a direct instance of catamorphisms either.

To express `factorial` and `wc` as catamorphisms, the tupling tactic (Meertens, 1992) can be used by expressing them as mutumorphisms — which correspond to catamorphisms computing a pair type $(\texttt{a}, \texttt{b})$ — defined together with the identity function. For example,

$$
\begin{aligned}
&\texttt{factorial}' = \texttt{fst}\ (\texttt{mutu alg}\ \texttt{alg}_{\texttt{id}})\ \textbf{where}\\
&\quad \texttt{alg Zero} &&= 1\\
&\quad \texttt{alg}\ (\texttt{Succ}\ (\texttt{fn}, \texttt{n})) &&= (\texttt{In}\ (\texttt{Succ n})) * \texttt{fn}\\
&\quad \texttt{alg}_{\texttt{id}}\ \texttt{Zero} &&= \texttt{In Zero}\\
&\quad \texttt{alg}_{\texttt{id}}\ (\texttt{Succ}\ (\_, \texttt{n})) &&= \texttt{In}\ (\texttt{Succ n})
\end{aligned}
$$

Better is to use a recursion scheme that captures this common pattern.

**Definition 7.** The *paramorphisms* (Meertens, 1992) is:

$$
\begin{aligned}
&\texttt{para} :: \texttt{Functor f} \Rightarrow (\texttt{f}\ (\mu\ \texttt{f}, \texttt{a}) \to \texttt{a}) \to \mu\ \texttt{f} \to \texttt{a}\\
&\texttt{para alg} = \texttt{alg} \circ \texttt{fmap}\ (\texttt{id} \vartriangle \texttt{para alg}) \circ \texttt{in}^{\circ}\ \textbf{where}\\
&\quad (\texttt{f} \vartriangle \texttt{g})\ \texttt{x} = (\texttt{f x}, \texttt{g x})
\end{aligned}
$$

The prefix para- is derived from Greek παρά, meaning 'beside'.

**Example 11.** With `para`, `factorial` is defined as

$$
\begin{aligned}
&\texttt{factorial} = \texttt{para alg}\ \textbf{where}\\
&\quad \texttt{alg Zero} &&= 1\\
&\quad \texttt{alg}\ (\texttt{Succ}\ (\texttt{n}, \texttt{fn})) &&= \texttt{In}\ (\texttt{Succ n}) * \texttt{fn}
\end{aligned}
$$

Compared with `cata`, `para` also supplies the original substructures besides their images to the algebra. However, `cata` and `para` are interdefinable in Haskell. Every catamorphism is simply a paramorphism that makes no use of the additional information:

$$\texttt{cata alg} = \texttt{para}\ (\texttt{alg} \circ \texttt{fmap snd})$$

Conversely, every paramorphism together with the identity function is a mutumorphism, which in turn is a catamorphism for a pair type $(\texttt{a}, \texttt{b})$, or directly:

$$\texttt{para alg} = \texttt{snd} \circ \texttt{cata}\ ((\texttt{In} \circ \texttt{fmap fst}) \vartriangle \texttt{alg})$$

In the unifying theory of conjugate hylomorphisms (Hinze *et al.*, 2015), paramorphisms are para-hylomorphisms with the final para-recursive coalgebra

$\texttt{in}^\circ$, which is arguably the most basic building block for constructing recursion schemes in this theory. Sometimes the recursion scheme of paramorphisms is called *primitive recursion*. However, functions definable with paramorphisms in Haskell are beyond primitive recursive functions in computability theory because of the presence of higher order functions. Indeed, the cannonical example of non-primitive recursive function, the Ackermann function, is definable with `cata` and thus `para`:

```
ack :: Nat → Nat → Nat
ack = cata alg where
  alg :: NatF (Nat → Nat) → (Nat → Nat)
  alg Zero      = In ∘ Succ
  alg (Succ a_n) = cata alg' where
    alg' :: NatF Nat → Nat
    alg' Zero = a_n (In (Succ (In Zero)))
    alg' (Succ a_{n+1,m}) = a_n a_{n+1,m}
```

## 6.2 Apomorphisms

Paramorphisms can be dualised to corecursion: The algebra of a paramorphism has type $\texttt{f}\ (\mu\ \texttt{f}, \texttt{a}) \to \texttt{a}$, in which $\mu\ \texttt{f}$ is dual to $\nu\ \texttt{f}$, and the pair type is dual to the `Either` type. Thus the coalgebra of the dual recursion scheme should have type $\texttt{c} \to \texttt{f}\ (\texttt{Either}\ (\nu\ \texttt{f})\ \texttt{c})$.

**Definition 8.** We call the following recursion scheme the *apomorphism* (Vene & Uustalu, 1998; Uustalu & Vene, 1999). Prefix apo- comes from Greek απο meaning 'apart from'.

```
apo :: Functor f ⇒ (c → f (Either (ν f) c)) → c → ν f
apo coalg = Out° ∘ fmap (either id (apo coalg)) ∘ coalg
```

which is sometimes called *primitive corecursion*.

Similar to anamorphisms, the coalgebra of an apomorphism generates a layer of `f`-structure in each step, but for substructures, it either generates a new seed of type `c` for corecursion as in anamorphisms, or a complete structure of $\nu\ \texttt{f}$ and stop the corecursion there.

In the same way that `cata` and `para` are interdefinable, `ana` and `apo` are interdefinable in Haskell too, but `apo` are particularly suitable for corecursive functions in which the future output is fully known at some step. Consider a function `maphd` from Vene & Uustalu (1998) that applies a function `f` to the first element (if there is) of a coinductive list.

```
maphd :: (a → a) → ν (ListF a) → ν (ListF a)
```

As an anamorphism, it is expressed as

```
maphd f = ana c ∘ Left where
  c (Left (Out° Nil))          = Nil
  c (Right (Out° Nil))         = Nil
  c (Left (Out° (Cons x xs)))  = Cons (f x) (Right xs)
  c (Right (Out° (Cons x xs))) = Cons x     (Right xs)
```

in which the seed for generation is of type `Either` ($\nu$ (`ListF a`)) ($\nu$ (`ListF a`)) to distinguish if the head element has been processed. This function is more intuitively an apomorphism since the future output is instantly known when the head element gets processed:

```
maphd′ f = apo coalg where
  coalg (Out° Nil) = Nil
  coalg (Out° (Cons x xs)) = Cons (f x) (Left xs)
```

Moreover, this definition is more efficient than the previous one because it avoids deconstructing and reconstructing the tail of the input list.

**Example 12.** Another instructive example of apomorphisms is inserting a value into an ordered (coinductive) list:

```
insert :: Ord a ⇒ a → ν (ListF a) → ν (ListF a)
insert y = apo c where
  c (Out° Nil)    = Cons y (Left (Out° Nil))
  c xxs@(Out° (Cons x xs))
    | y ⩽ x       = Cons y (Left xxs)
    | otherwise = Cons x (Right xs)
```

In each case, `Cons x` emits `x` as output, and `Cons _ (Left b)` makes list `b` the rest of the output whereas `Cons _ (Right b)` continues the corecursion to insert `y` into `b`.

## 6.3 Zygomorphisms

When computing a recursive function on a datatype, it is usual the case that some auxiliary information about substructures is needed in addition to the images of substructures under the recursive function being computed. For instance, when determining if a binary tree is a perfect tree — a tree in which all leaf nodes have the same depth and all interior nodes have two children — by structural recursion, besides checking that the left and right subtrees are both perfect, it is also needed to check that they have the same depth

```
perfect :: μ (TreeF e)      → Bool
perfect (In Empty)          = True
perfect (In (Node l _ r)) = perfect l ∧ perfect r ∧ (depth l ≡ depth r)
depth :: μ (TreeF e)        → Integer
depth (In Empty)            = 0
depth (In (Node l _ r))     = 1 + max (depth l) (depth r)
```

Function `perfect` is not directly a catamorphism because the algebra is not provided with `depth l` and `depth r` by the `cata` recursion scheme. However we can define `perfect` as a paramorphism:

$$\begin{aligned}&\texttt{perfect}' = \texttt{para alg where}\\&\quad \texttt{alg Empty} = \texttt{True}\\&\quad \texttt{alg (Node (l, p}_\texttt{l}\texttt{) \_ (r, p}_\texttt{r}\texttt{))} = \texttt{p}_\texttt{l} \wedge \texttt{p}_\texttt{r} \wedge (\texttt{depth l} \equiv \texttt{depth r})\end{aligned}$$

but this is inefficient because the depth of a subtree is computed repeatedly at each of its ancestor nodes, despite the fact that `depth` can be computed structurally too. Thus we need a generalisation of paramorphisms in which instead of the original structure being kept and supplied to the algebra, some auxiliary information (that can be computed structurally) is maintained along the recursion and supplied to the algebra, which leads to the following recursion scheme.

**Definition 9.** The *zygomorphisms* (Malcolm, 1990a) is:

$$\begin{aligned}&\texttt{zygo} :: \texttt{Functor f} \Rightarrow (\texttt{f (a, b)} \to \texttt{a}) \to (\texttt{f b} \to \texttt{b}) \to \mu\,\texttt{f} \to \texttt{a}\\&\texttt{zygo alg}_1\ \texttt{alg}_2 = \texttt{fst (mutu alg}_1\ (\texttt{alg}_2 \circ \texttt{fmap snd}))\end{aligned}$$

in which $\texttt{alg}_1$ computes the function of interest from the recursive results together with auxiliary information of type `b`, and $\texttt{alg}_2$ maintains the auxiliary information. Malcolm called zygomorphisms 'yoking together of paramorphisms and catamorphisms' and prefix 'zygo-' is derived from Greek ζυγόν meaning 'yoke'.

**Example 13.** As we said, `zygo` is a generalisation of `para`:

$$\texttt{para alg} = \texttt{zygo alg In}$$

**Example 14.** In terms of `zygo`, `perfect` is `zygo p d` where

$$\begin{aligned}&\texttt{p} :: \texttt{TreeF e (Bool, Integer)} \to \texttt{Bool}\\&\texttt{p Empty} &&= \texttt{True}\\&\texttt{p (Node (p}_\texttt{l}\texttt{, d}_\texttt{l}\texttt{) \_ (p}_\texttt{r}\texttt{, d}_\texttt{r}\texttt{))} &&= \texttt{p}_\texttt{l} \wedge \texttt{p}_\texttt{r} \wedge (\texttt{d}_\texttt{l} \equiv \texttt{d}_\texttt{r})\\&\texttt{d} :: \texttt{TreeF e Integer} \to \texttt{Integer}\\&\texttt{d Empty} &&= 0\\&\texttt{d (Node d}_\texttt{l}\texttt{ \_ d}_\texttt{r}\texttt{)} &&= 1 + (\texttt{max d}_\texttt{l}\ \texttt{d}_\texttt{r})\end{aligned}$$

In the unifying framework by means of adjunction, zygomorphisms arise from an adjunction between the slice category $C \downarrow b$ and the base category $C$ (Hinze & Wu, 2016). The same adjunction also leads to the dual of zygomorphisms — the recursion scheme in which a seed is unfolded to a recursive datatype that defined with some auxiliary datatype.

# 7 Course-of-Value (Co)Recursion

This section is about the patterns in dynamic programming algorithms, in which a problem is solved based on solutions to subproblems just as in catamorphisms. But in dynamic programming algorithms, subproblems are largely shared among problems, and thus a common implementation technique is to memoise solved subproblems with an array. This section shows the recursion schemes for dynamic programming, the *histomorphism* and a slight generalisation the *dynamorphism*, and the corecursive dual the *futumorphism*.

## 7.1 Histomorphisms

A powerful generalisation of catamorphisms is to provide the algebra with all the recursively computed results of direct and indirect substructures rather than only the *immediate* substructures. Consider the longest increasing subsequence (*LIS*) problem: given a sequence of integers, its subsequences are obtained by deleting some (or none) of its elements and keeping the remaining elements in its original order, and the problem is to find (the length of) longest subsequences in which the elements are in increasing order. For example, the longest increasing subsequences of $[1, 6, -5, 4, 2, 3, 9]$ have length 4 and one of them is $[1, 2, 3, 9]$.

An effective way to find LIS follows the observation that an LIS of $\mathtt{x} : \mathtt{xs}$ is either an LIS of $\mathtt{xs}$ or a subsequence beginning with the head element $\mathtt{x}$, and moreover in the latter case the LIS must have a tail that itself is also an LIS (or the whole LIS could be made longer). This idea is implemented by the program below.

$$
\begin{aligned}
&\mathtt{lis} = \mathtt{snd} \circ \mathtt{lis}' \\
&\mathtt{lis}' :: \mathtt{Ord\ a} \Rightarrow [\mathtt{a}] \to (\mathtt{Integer}, \mathtt{Integer}) \\
&\mathtt{lis}'\ [\,] \qquad = (0, 0) \\
&\mathtt{lis}'\ (\mathtt{x} : \mathtt{xs}) = (\mathtt{a}, \mathtt{b})\ \mathbf{where} \\
&\quad \mathtt{a} = 1 + \mathtt{maximum}\ [\mathtt{fst}\ (\mathtt{lis}'\ \mathtt{sub}) \mid \mathtt{sub} \leftarrow \mathtt{tails\ xs}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{null\ sub} \vee \mathtt{x} < \mathtt{head\ sub}] \\
&\quad \mathtt{b} = \mathtt{max\ a}\ (\mathtt{snd}\ (\mathtt{lis}'\ \mathtt{xs}))
\end{aligned}
$$

where the first component of $\mathtt{lis}'\ (\mathtt{x} : \mathtt{xs})$ is the length of the longest increasing subsequence that is restricted to begin with the first element $\mathtt{x}$, and the second component is the length of LIS without this restriction and thus $\mathtt{lis} = \mathtt{snd} \circ \mathtt{lis}'$.

Unfortunately this implementation is very inefficient because $\mathtt{lis}'$ is recursively applied to possibly all substructures of the input, leading to exponential running time with respect to the length of the input. The inefficiency is mainly due to redundant recomputation of $\mathtt{lis}'$ on substructures: when computing $\mathtt{lis}'\ (\mathtt{xs} +\!\!+ \mathtt{ys})$, for each $\mathtt{x}$ in $\mathtt{xs}$, $\mathtt{lis}'\ \mathtt{ys}$ is recomputed although the results are identical. A technique to speed up the algorithm is to memoise the results of $\mathtt{lis}'$ on substructures and skip recomputing the function when identical input is encountered, a technique called *dynamic programming*.

To implement dynamic programming, what we want is a scheme that provides the algebra with a table of the results for all substructures that have been computed. A table is represented by the `Cofree` comonad

```
data Cofree f a where
  (◁) :: a → f (Cofree f a) → Cofree f a
```

which can be intuitively understood as a (coinductive) tree whose branching structure is determined by functor `f` and all nodes are tagged with a value of type `a`, which can be extracted with

```
extract :: Cofree f a → a
extract (x ◁ _) = x
```

**Definition 10.** The recursion scheme *histomorphism* (Uustalu & Vene, 1999) is:

```
histo :: Functor f ⇒ (f (Cofree f a) → a) → μ f → a
histo alg = extract ∘ cata (λx → alg x ◁ x)
```

which is a catamorphism computing a memo-table of type `Cofree f a` followed by extracting the result for the whole structure. The name histo- follows that the entire computation history is passed to the algebra. It is also called *course-of-value* recursion.

**Example 15.** The dynamic programming implementation of `lis` is then:

```
lis'' :: Ord a ⇒ μ (ListF a) → Integer
lis'' = snd ∘ histo alg
alg :: Ord a ⇒ ListF a (Cofree (ListF a) (Integer, Integer))
            → (Integer, Integer)
alg Nil              = (0, 0)
alg (Cons x table) = (a, b) where
  a = 1 + findNext x table
  b = max a (snd (extract table))
```

where `findNext` searches in the rest of the list for the element that is greater than `x` and begins a longest increasing subsequence:

```
findNext :: Ord a ⇒ a → Cofree (ListF a) (Integer, Integer) → Integer
findNext x ((a, _) ◁ Nil)                = a
findNext x ((a, _) ◁ (Cons y table')) = if x < y then max a b else b
  where b = findNext x table'
```

which improves the time complexity to quadratic time because `alg` runs in linear time for each element and `alg` is computed only once for each element.

In the unifying theory of recursion schemes by adjunctions, histomorphisms arise from the adjunction $U \dashv \mathit{Cofree}_F$ (Hinze & Wu, 2013) where $\mathit{Cofree}_F$ sends an object to its cofree coalgebra in the category of $F$-coalgebras, and $U$ is the forgetful functor. As we have seen, cofree coalgebras are used to model the memo-table of computation history in histomorphisms, but an oddity here is that (the carrier of) the cofree coalgebra is a possibly infinite structure, while the computation history is in fact finite because the input is a finite inductive structure. A remedy for this problem is to replace cofree coalgebras with *cofree para-recursive coalgebras* in the construction, and the `Cofree f a` comonad in `histo` is replaced by its para-recursive counterpart, which is exactly *finite* trees whose branching structure is `f` and nodes are tagged with `a`-values (Hinze *et al.*, 2015).

## 7.2 Dynamorphisms

Histomorphisms require the input to be an initial algebra, and this is inconvenient in applications whose structure of computation is determined on the fly while computing. An example is the following program finding the length of *longest common subsequences* (LCS) of two sequences (Bergroth *et al.*, 2000).

```
lcs :: Eq a ⇒ [a] → [a] → Integer
lcs [] _ = 0
lcs _ [] = 0
lcs xxs@(x : xs) yys@(y : ys)
  | x ≡ y       = lcs xs ys + 1
  | otherwise = max (lcs xs yys) (lcs xxs ys)
```

This program runs in exponential time but it is well suited for optimisation with dynamic programming because a lot of subproblems are shared across recursion. However, it is not accommodated by `histo` because the input, a pair of lists, is not an initial algebra. Therefore it is handy to generalise `histo` by replacing $\mathtt{in}^\circ$ with a user-supplied recursive coalgebra:

**Definition 11.** The *dynamorphism* (evidently the name is derived from *dyna*mic programming) introduced by Kabanov & Vene (2006) is:

```
dyna :: Functor f ⇒ (f (Cofree f a) → a)
                  → (c → f c) → c → a
dyna alg coalg = extract ∘ hylo (λx → alg x ◁ x) coalg
```

in which the recursive coalgebra `c` breaks a problem into subproblems, which are recursively solved, and the algebra `alg` solves a problem with solutions to all direct and indirect subproblems.

Because the subproblems of a dynamic programming algorithm together with the dependency relation of subproblems form an acyclic graph, an appealing choice of the functor `f` in `dyna` is `ListF` and the coalgebra `c` generates subproblems in a topological order of the dependency graph of subproblems, so that a subproblem is solved exactly once when it is needed by bigger problems.

**Example 16.** Continuing the example of LCS, the set of subproblems of `lcs` $s_1$ $s_2$ is all $(\mathtt{x}, \mathtt{y})$ for `x` and `y` being suffixes of $s_1$ and $s_2$ respectively. An ordering of subproblems that respects their computing dependency is:

$$
\begin{aligned}
&\mathtt{g} :: ([\mathtt{a}], [\mathtt{a}]) \to \mathtt{ListF}\ ([\mathtt{a}], [\mathtt{a}])\ ([\mathtt{a}], [\mathtt{a}]) \\
&\mathtt{g}\ ([\,], [\,]) = \mathtt{Nil} \\
&\mathtt{g}\ (\mathtt{x}, \mathtt{y}) \ = \textbf{if}\ \mathtt{null\ y}\ \textbf{then}\ \mathtt{Cons}\ (\mathtt{x}, \mathtt{y})\ (\mathtt{tail\ x}, s_2) \\
&\qquad\qquad\qquad\quad \textbf{else}\ \ \mathtt{Cons}\ (\mathtt{x}, \mathtt{y})\ (\mathtt{x}, \mathtt{tail\ y})
\end{aligned}
$$

The algebra `a` solves a problem with solutions to subproblems available:

$$
\begin{aligned}
&\mathtt{a} :: \mathtt{ListF}\ ([\mathtt{a}], [\mathtt{a}])\ (\mathtt{Cofree}\ (\mathtt{ListF}\ ([\mathtt{a}], [\mathtt{a}]))\ \mathtt{Integer}) \to \mathtt{Integer} \\
&\mathtt{a}\ \mathtt{Nil} = 0 \\
&\mathtt{a}\ (\mathtt{Cons}\ (\mathtt{x}, \mathtt{y})\ \mathtt{table}) \\
&\quad \mid \mathtt{null\ x} \vee \mathtt{null\ y} = 0 \\
&\quad \mid \mathtt{head\ x} \equiv \mathtt{head\ y} = \mathtt{index\ table}\ (\mathtt{offset}\ 1\ 1) + 1 \\
&\quad \mid \mathtt{otherwise} \qquad = \mathtt{max}\ (\mathtt{index\ table}\ (\mathtt{offset}\ 1\ 0)) \\
&\qquad\qquad\qquad\qquad\quad\ \ (\mathtt{index\ table}\ (\mathtt{offset}\ 0\ 1))
\end{aligned}
$$

where `index t n` extracts the `n`-th entry of the memo-table:

$$
\begin{aligned}
&\mathtt{index} :: \mathtt{Cofree}\ (\mathtt{ListF\ a})\ \mathtt{p} \to \mathtt{Integer} \to \mathtt{p} \\
&\mathtt{index\ t}\ 0 \qquad\qquad\quad = \mathtt{extract\ t} \\
&\mathtt{index}\ (\_ \lhd (\mathtt{Cons}\ \_\ \mathtt{t}'))\ \mathtt{n} = \mathtt{index\ t}'\ (\mathtt{n} - 1)
\end{aligned}
$$

The tricky part is computing the indices for entries to subproblems in the memo-table. Because subproblems are enumerated by `g` in the order that reduces the second sequence first, thus the entry for $(\mathtt{drop\ n\ x}, \mathtt{drop\ m\ y})$ in the memo-table when computing $(\mathtt{x}, \mathtt{y})$ is:

$$\mathtt{offset\ n\ m} = \mathtt{n} * (\mathtt{length}\ s_2 + 1) + \mathtt{m} - 1$$

Putting them together, we get the dynamic programming solution to LCS with `dyna`:

$$\mathtt{lcs}'\ s_1\ s_2 = \mathtt{dyna\ a\ g}\ (s_1, s_2)$$

which improves the exponential running time of specification `lcs` to $\mathcal{O}(|s_1||s_2|^2)$, yet slower than the $\mathcal{O}(|s_1||s_2|)$ array-based implementation of dynamic programming because of the cost of indexing the list-structured memo-table.

## 7.3 Futumorphisms

Histomorphisms are generalised catamorphisms that can *inspect the history* of computation. The dual generalisation is anamorphisms that can *control the future*. As an example, consider the problem of decoding the *run-length encoding* of a sequence: the input is a list of elements $(\mathtt{n}, \mathtt{x})$ of type $(\mathtt{Int}, \mathtt{a})$ and $\mathtt{n} > 0$ for all elements. The output is a list of element type `a` and each $(\mathtt{n}, \mathtt{x})$ in the

input is interpreted as `n` consecutive copies of `x`. As an anamorphism, it can be expressed as

```
rld :: [(Int, a)] → ν (ListF a)
rld = ana c where
  c [] = Nil
  c ((n, x) : xs)
    | n ≡ 1       = Cons x xs
    | otherwise = Cons x ((n − 1, x) : xs)
```

This is slightly awkward because anamorphisms can emit only one layer of the structure in each step, while in this example it is more natural to emit `n` copies of `x` in a batch. This can be done if the recursion scheme allows the coalgebra to generate more than one layer in a single step — in a sense controlling the future of the computation.

Multiple layers of the structure determined by a functor `f` are represented by the `Free` algebra:

```
data Free f a = Ret a | Op (f (Free f a))
```

which is (inductive) trees whose branching structure is determined by `f` and leaf nodes are `a`-values. Free algebras subsume initial algebras as `Free f Void` $\cong \mu$ `f` where `Void` is the bottom type, and `cata` for $\mu$ `f` is replaced by

```
eval :: Functor f ⇒ (f b → b) → (a → b) → Free f a → b
eval alg g (Ret a) = g a
eval alg g (Op k)  = alg (fmap (eval alg g) k)
```

**Definition 12.** With these constructions, we define the recursion scheme *futumorphisms* (Uustalu & Vene, 1999):

```
futu :: Functor f ⇒ (c → f (Free f c)) → c → ν f
futu coalg = ana coalg′ ∘ Ret where
  coalg′ (Ret a) = coalg a
  coalg′ (Op k)  = k
```

**Example 17.** We can redefine `rld` as a futumorphism:

```
rld′ :: [(Int, a)] → ν (ListF a)
rld′ = futu dec

dec []           = Nil
dec ((n, c) : xs) = let (Op g) = rep n in g where
  rep 0 = Ret xs
  rep m = Op (Cons c (rep (m − 1)))
```

Note that `dec` assumes `n`$>0$ because `futu` demands that the coalgebra generate at least one layer of `f`-structure.

Theoretically, futumorphisms are adjoint unfolds from the adjunction $Free_F \dashv U$ where $Free_F$ maps object $a$ to the free algebra generated by $a$ in the category of $F$-algebras. In the same way that dynamorphisms generalise histomorphisms, futumorphisms can be generalised by replacing $(\nu\ \mathtt{F}, \mathtt{Out}^\circ)$ with a user-supplied corecursive $F$-algebra. A broader generalisation is to combine futumorphisms and histomorphisms in a similar way to hylomorphisms combining anamorphisms and catamorphisms:

```
chrono :: Functor f ⇒ (f (Cofree f b) → b)
                    → (a → f (Free f a))
                    → a → b
chrono alg coalg = extract ∘ hylo alg′ coalg′ ∘ Ret where
  alg′ x = alg x ◁ x
  coalg′ (Ret a) = coalg a
  coalg′ (Op k)  = k
```

which is dubbed *chronomorphisms* by Kmett (2008) (prefix chrono- from Greek χρόνος meaning 'time').

# 8 Monadic Structural Recursion

Up to now we have been working in the world of pure functions. It is certainly possible to extend the recursion schemes to the non-pure world where computational effects are modelled with monads.

## 8.1 Monadic Catamorphism

Let us start with a straightforward example of printing a tree with the `IO` monad:

```
printTree :: Show a ⇒ μ (TreeF a) → IO ()
printTree (In Empty)        = return ()
printTree (In (Node l a r)) = do printTree l; printTree r; print a
```

The reader may have recognised that it is already a catamorphism:

```
printTree′ :: Show a ⇒ μ (TreeF a) → IO ()
printTree′ = cata printAlg where
  printAlg :: Show a ⇒ TreeF a (IO ()) → IO ()
  printAlg Empty          = return ()
  printAlg (Node ml a mr) = do ml; mr; print a
```

Thus a straightforward way of abstracting 'monadic catamorphisms' is to restrict `cata` to monadic values.

**Definition 13** (`cataM`)**.** We call the following recursion scheme *catamorphisms on monadic values*:

```
cataM :: (Functor f, Monad m) ⇒ (f (m a) → m a) → μ f → m a
cataM algM = cata algM
```

which is the second approach to monadic catamorphisms in (Pardo, 2005)

However, `cataM` does not fully capture our intuition for 'monadic catamorphism' because the algebra `algM :: f (m a) → m a` is allowed to combine computations from substructures in arbitrary ways. For a more precise characterisation of 'monadic catamorphisms', we decompose `algM :: f (m a) → m a` in `cataM` into two parts: a function `alg :: f a → m a` which (monadically) computes the result for the whole structure given the results of substructures, and a polymorphic function

$$\texttt{seq} :: \forall \texttt{x.f}\ (\texttt{m x}) \to \texttt{m}\ (\texttt{f x})$$

called a *sequencing* of `f` over `m`, which combines computations for substructures into one monadic computation. The decomposition reflects the intuition that a monadic catamorphism processes substructures (in the order determined by `seq`) and combines their results (by `alg`) to process the root structure:

$$\texttt{algM r} = \texttt{seq r} \gg\!= \texttt{alg}.$$

**Example 18.** The functor of binary trees `TreeF` can be sequenced from left to right

```
lToR :: Monad m ⇒ TreeF a (m x) → m (TreeF a x)
lToR Empty          = return Empty
lToR (Node ml a mr) = do l ← ml; r ← mr; return (Node l a r)
```

and also from right to left

```
rToL :: Monad m ⇒ TreeF a (m x) → m (TreeF a x)
rToL Empty          = return Empty
rToL (Node ml a mr) = do r ← mr; l ← ml; return (Node l a r)
```

**Definition 14** (`mcata`)**.** The *monadic catamorphism* (Fokkinga, 1994; Pardo, 2005) is the following recursion scheme.

```
mcata :: (Monad m, Functor f)
      ⇒ (∀x.f (m x) → m (f x))
      → (f a → m a) → μ f → m a
mcata seq alg = cata ((≫=alg) ∘ seq)
```

**Example 19.** The program `printTree` discussed earlier is a monadic catamorphism:

```
printTree'' :: Show a ⇒ μ (TreeF a) → IO ()
printTree'' = mcata lToR printElem where
  printElem Empty        = return ()
  printElem (Node _ a _) = print a
```

Note that `mcata` is strictly less expressive because `mcata` requires all subtrees processed before the root.

**Distributive Conditions** In the literature (Fokkinga, 1994; Pardo, 2005; Hinze & Wu, 2016), the sequencing of a monadic catamorphism is required to be a *distributive law* of functor `f` over monad `m`, which means that `seq`::$\forall$`x.f (m x)` $\to$ `m (f x)` satisfies two conditions:

$$\texttt{seq} \circ \texttt{fmap return} = \texttt{return} \tag{5}$$

$$\texttt{seq} \circ \texttt{fmap join} = \texttt{join} \circ \texttt{fmap seq} \circ \texttt{seq} \tag{6}$$

Intuitively, condition (5) prohibits `seq` from inserting additional computational effects when combining computations for substructures, which is a reasonable requirement. Condition (6) requires `seq` to be commutative with monadic sequencing. These requirements are theoretically elegant, because they allow functor `f` to be lifted to the Kleisli category of `m` and consequently `mcata seq alg` is also a catamorphism in the Kleisli category (`mcata` by definition is a catamorphism in the base category)— giving us more calculational properties. Unfortunately, condition (6) is too strong in practice. For example, neither `lToR` nor `rToL` in Example (18) satisfies condition (6) when `m` is the `IO` monad. Let

```
c = Node (putStr "A" ≫ return (putStr "C"))
         (putStr "B" ≫ return (putStr "D"))
```

Then (`lToR` $\circ$ `fmap join`) `c` prints `"ACBD"` but (`join` $\circ$ `fmap lToR` $\circ$ `lToR`) `c` prints `"ABCD"`. In fact, there is no distributive law of `TreeF a` over a monad unless it is commutative, which excludes the `IO` monad and `State` monad. Thus we drop the requirement for `seq` being a distributive law in our definition of monadic catamorphism.

## 8.2 More Monadic Recursion Schemes

As we mentioned above, `mcata` is the catamorphism in the Kleisli category provided `seq` is a distributive law. No doubt, we can replay our development of recursion schemes in the Kleisli category to get the monadic version of more recursion schemes. For example, we have *monadic hylomorphisms* Pardo (1998, 2005):

```
mhylo :: (Monad m, Functor f) ⇒ (∀x.f (m x) → m (f x))
                              → (f a → m a) → (c → m (f c))
                              → c → m a
mhylo seq alg coalg c = do x ← coalg c;
                           y ← seq (fmap (mhylo seq alg coalg) x)
                           alg y
```

which specialises to `mcata` by `mhylo seq alg` (`return` $\circ$ `in`$^\circ$) and *monadic anamorphisms* by

```
mana :: (Monad m, Functor f) ⇒ (∀x.f (m x) → m (f x))
                             → (c → m (f c)) → c → m (ν f)
mana seq coalg = mhylo seq (return ∘ Out°) coalg
```

Other recursion schemes discussed in this paper can be devised in the same way.

**Example 20.** Generating a random tree of some depth with `randomIO :: IO Int` is a monadic anamorphism:

```
ranTree :: Integer → IO (ν (TreeF Int))
ranTree = mana lToR gen where
  gen :: Integer → IO (TreeF Int Integer)
  gen 0 = return Empty
  gen n = do a ← randomIO :: IO Int
             return (Node (n − 1) a (n − 1))
```

Another way to dualise `mcata` is to replace monads with comonads, resulting in the following recursion scheme.

**Definition 15.** The *comonadic anamorphism* is:

```
wana :: (Comonad w, Functor f)
     ⇒ (∀x.w (f x) → f (w x))
     → (w c → f c) → w c → ν f
wana dist coalg = ana (dist ∘ extend coalg)
```

TODO: Find an example ...

# 9 Structural Recursion on GADTs

So far we have worked exclusively with (co)inductive datatypes, but they do not cover all algebraic datatypes and *generalised algebraic datatypes* (GADTs), which are proved to be useful in practice. An example of algebraic datatypes that is not (co)inductive is the datatype for purely functional *random-access lists* (Okasaki, 1995):

```
data RList a = Null | Zero (RList (a, a)) | One a (RList (a, a))
```

The recursive occurrences of `RList` in constructor `Zero` and `One` are `RList (a, a)` rather than `RList a`, and consequently we cannot model `RList a` as $\mu$ `f` for some functor `f` as we did for lists. Algebraic datatypes such as `RList` whose defining equation has on the right-hand side any occurrence of the declared type applied to parameters different from those on the left-hand side are called *non-regular* datatypes or *nested* datatypes (Bird & Meertens, 1998).

Nested datatypes are covered by a broader range of datatypes called *generalised algebraic datatypes* (GADTs). In terms of the **data** syntax in Haskell, the generalisation of GADTs is to allow the parameters `P` supplied to the declared type `D` on the left-hand side of the defining equation

```
data D P = ...
```

to be more complex than type variables. GADTs have a different syntax from that of ADTs in Haskell. For example, as a GADT, `RList` is

```
data RList :: * → *where
  Null :: RList a
  Zero :: RList (a, a) → RList a
  One  :: a → RList (a, a) → RList a
```

in which each constructor is directly declared with a type signature. Translating to this syntax, allowing parameters on the left-hand side of an ADT equation to be not just variables means that the finally returned type of constructors of a GADT `G` can be more complex than `G a` where `a` is a type variable. A classic example is fixed length vectors of `a`-values:

```
data Z'
data S' a
data Vec :: * → * → *where
  Nil  :: Vec a Z'
  Cons :: a → Vec a n → Vec a (S' n)
```

in which `Z'` and `S' a` have no (non-bottom) inhabitants and only serve for encoding natural numbers at the type level.

GADTs are a powerful tool to ensure program correctness by indexing datatypes with sophisticated properties of data, such as the size or shape of data, and then the type checker can check these properties statically. For example, the following program extracting the first element of a vector is always safe because the type of its argument guarantees it is non-empty.

```
safeHead :: Vec a (S' n) → a
safeHead (Cons a _)    = a
```

**GADTs as Fixed Points** Although nested datatypes and GADTs cannot be interpreted as fixed points of endofunctors on the base category $C$, surprisingly, there is another category, in which GADTs are fixed points. If we restrict our attention to GADTs with one type parameter, such as `RList`, they have kind $* \to *$ in Haskell, so we might hope that they are fixed points of higher-order functors $(* \to *) \to (* \to *)$. Pleasingly, Bird & Meertens (1998) showed nested types are indeed fixed points of endofunctors on the functor category $C^C$ (whose objects are functors from $C$ to $C$ and arrows are natural transformations between functors), and Johann & Ghani (2007) demonstrated the associated catamorphism is expressive enough to capture structural recursion on nested datatypes.

Alas, not all GADTs are an endofunctor on $C$ (an object in $C^C$), let alone a fixed point of some endofunctor on $C^C$. For example, given GADT

```
data G a where
  Leaf :: a → G a
  Prod :: G a → G b → G (a, b)
```

we cannot define a `Functor` instance for `G` because for the `Prod` case

$$\texttt{fmap f (Prod ga gb)} = \_ :: \texttt{G c}$$

we have no way to construct a `G c` given `f` $::$ `(a, b)` $\to$ `c`, `ga` $::$ `G a` and `gb` $::$ `G b`. Johann & Ghani (2008) proposed a solution to this problem by instead interpreting GADTs as functors from $|C|$ to $C$, where $|C|$ is the discrete category of $C$, i.e. the category having the same objects as $C$ but having no arrows except the identity ones. Consequently GADTs become fixed points of endofunctors on the functor category $C^{|C|}$. In Haskell, endofunctors on $C^{|C|}$ are encoded as

```
class HFunctor f where
  hfmap :: (a ∴→ b) → (f a ∴→ f b)
```

where `a` $\dot{\to}$ `b` is the type synonym for $\forall$`i.a i` $\to$ `b i`. Given two GADTs `a, b`$::* \to *$ interpreted as functors $|C| \to C$, polymorphic functions $\forall$`i.a i` $\to$ `b i` can be interpreted as natural transformations from `a` to `b` which are arrows from `a` to `b` in category $C^{|C|}$. Thus `fmap`'s counterpart `hfmap` maps an arrow `a` $\dot{\to}$ `b` to an arrow `f a` $\dot{\to}$ `f b`. On top of these, the least-fixed-point operator for an `HFunctor` is

```
data μ̇ :: ((∗ → ∗) → (∗ → ∗)) → (∗ → ∗) where
  İn :: f (μ̇ f) i → μ̇ f i
```

**Example 21.** Fixed-length vectors `Vec e` are isomorphic to $\dot{\mu}$ `(VecF e)` where

```
data VecF :: ∗ → (∗ → ∗) → (∗ → ∗) where
  NilF  :: VecF e f Z′
  ConsF :: e → f n → VecF e f (S′ n)
```

which has `HFunctor` instance

```
instance HFunctor (VecF e) where
  hfmap phi NilF         = NilF
  hfmap phi (ConsF e es) = ConsF e (phi es)
```

As an aside, in the example above the base functor `VecF a` for GADT `Vec` is itself a GADT. This is a problem if we want to give semantics to GADTs by interpreting them as fixed points. Johann & Ghani (2008) showed how to break the circle by using left Kan-extensions to turn GADT `VecF a` into one with only the equality GADT and existential quantification.

**Indexed Catamorphisms** It is then straightforward to devise the catamorphism for $\dot{\mu}$ in the category $C^{|C|}$.

**Definition 16** (`icata`)**.** We call the following recursion scheme the *indexed catamorphism*.

```
icata :: HFunctor f ⇒ (f a ∴→ a) → μ̇ f ∴→ a
icata alg (İn x) = alg (hfmap (icata alg) x)
```

**Example 22.** Unsurprisingly, mapping a function over a vector is an `icata`:

```
vmap :: ∀a b.(a → b) → μ̇ (VecF a) ↛ μ̇ (VecF b)
vmap f = icata alg where
  alg :: VecF a (μ̇ (VecF b)) ↛ μ̇ (VecF b)
  alg NilF          = İn NilF
  alg (ConsF a bs) = İn (ConsF (f a) bs)
```

**Example 23.** Terms of untyped lambda calculus with de Bruijn indices can be modelled as the fixed point of the following GADT.

```
data LambdaF :: (∗ → ∗) → (∗ → ∗) where
  Var :: a → LambdaF f a
  App :: f a → f a → LambdaF f a
  Abs :: f (Maybe a) → LambdaF f a
```

Letting `a` be some type, inhabitants of $\dot{\mu}$ `LambdaF a` are precisely the lambda terms in which free variables range over `a`. Thus $\dot{\mu}$ `LambdaF Void` is the type of closed lambda terms where `Void` is the type has no inhabitants. Note that the constructor `Abs` applies the recursive placeholder `f` to `Maybe a`, providing the inner term with exactly one more fresh variable `Nothing`.

The size of a lambda term can certainly be computed structurally. However, what we get from `icata` is always an arrow $\dot{\mu}$ `LambdaF` $\dot{\to}$ `a` for some `a` $:: * \to *$. If we want to compute just an integer, we need to wrap it in a constant functor:

```
newtype K a x = K {unwrap :: a}
```

Computing the size of a term is done by

```
size :: μ̇ LambdaF ↛ K Integer
size = icata alg where
  alg :: LambdaF (K Integer) ↛ K Integer
  alg (Var _)           = K 1
  alg (App (K n) (K m)) = K (n + m + 1)
  alg (Abs (K n))       = K (n + 1)
```

**Example 24.** An indexed catamorphism `icata alg` is a function $\forall$`i`.$\dot{\mu}$ `f i` $\to$ `a i` polymorphic in index `i`, however, we might be interested in GADTs and nested datatypes applied to some monomorphic index. Consider the following program summing up a random-access list of integers.

```
sumRList :: RList Integer → Integer
sumRList Null       = 0
sumRList (Zero xs)  = sumRList (fmap (uncurry (+)) xs)
sumRList (One x xs) = x + sumRList (fmap (uncurry (+)) xs)
```

Does it fit into an indexed catamorphism from $\dot{\mu}$ `RListF`?

```
data RListF f a = NullF | ZeroF (f (a, a)) | OneF a (f (a, a))
```

The answer is yes, with the clever choice of the continuation monad `Cont Integer a` as the result type of `icata`.

```
newtype Cont r a = Cont {runCont :: (a → r) → r}
sumRList′ :: μ̇ RListF Integer → Integer
sumRList′ x = runCont (h x) id where
  h :: μ̇ RListF ·→ Cont Integer
  h = icata sum where
    sum :: RListF (Cont Integer) ·→ Cont Integer
    sum NullF      = Cont (λk → 0)
    sum (ZeroF s)  = Cont (λk → runCont s (fork k))
    sum (OneF a s) = Cont (λk → k a + runCont s (fork k))
    fork :: (y → Integer) → (y, y) → Integer
    fork k (a, b) = k a + k b
```

Historically, structural recursion on nested datatypes applied to a monomorphic type was thought as falling out of `icata` and led to the development of *generalised folds* (Bird & Paterson, 1999; Abel *et al.*, 2005). Later, Johann & Ghani (2007) showed `icata` is in fact expressive enough by using right Kan-extensions as the result type of `icata`, of which `Cont` used in this example is a special case.

# 10 More Recursion Schemes

Lastly, we give two general methods of finding more *fantastic morphisms*.

**From Categories and Adjunctions** As we have seen, recursion schemes live with categories and adjunctions. Thus when we encounter a new category, it is a good idea to think about the catamorphism or the anamorphism in this category, as we did for the Kleisli category, where we obtained `mcata`, and the functor category $C^{|C|}$, where we obtained `icata`, etc. And whenever we encounter an adjunction $L \dashv R$, we can think about if functions of type `L c → a`, especially `L (μ f) → a`, are anything interesting. If they are, there might be an interesting adjoint fold or conjugate hylomorphism. For example, Paykin & Zdancewic (2017) introduced the linearity adjunction between a non-linear host language and a linear embedded domain specific language, and an adjoint fold from this adjunction would be linearly consuming a non-linear inductive structure. We refrain from going into details of this recursion scheme and invite the reader to search for more interesting recursion schemes in his or her domain.

**Composing Recursion Schemes** Up to now we have considered recursion schemes in isolation, each of which provides an extra functionality compared with `cata` or `ana`, such as mutual recursion, accessing the original structure,

accessing the computation history. However, when writing larger programs in practice, we might want combine the functionalities provided by multiple recursion schemes. For example, we possibly want to define two mutually recursive functions with accumulating parameters. Thus we need a recursion scheme of type

$$\begin{aligned}\texttt{mutuAccu} :: \texttt{Functor f} \Rightarrow\ & (\texttt{f}\ (\texttt{p} \to (\texttt{a}, \texttt{b})) \to \texttt{p} \to \texttt{a}) \\ & \to (\texttt{f}\ (\texttt{p} \to (\texttt{a}, \texttt{b})) \to \texttt{p} \to \texttt{b}) \\ & \to ((\mu\ \texttt{f}, \texttt{p}) \to \texttt{a}, (\mu\ \texttt{f}, \texttt{p}) \to \texttt{b})\end{aligned}$$

Theoretically, `mutuAccu` is the composite of `mutu` and `accu` in the sense that the adjunction $- \times p \dashv (-)^p$ underlying `accu` and the adjunction $\Delta \dashv \times$ underlying `mutu` can be composed to adjunction

$$\Delta \circ (- \times p) \dashv (-)^p \circ \times$$

which exactly induces `mutuAccu` as an adjoint fold (Hinze, 2013). Unfortunately, the Haskell implementations of `mutu` and `accu` in this paper are not composable. A composable library of recursion schemes in Haskell would require considerable machinery for encoding categories other than the one interpreting the types and adjunctions between them, and how to do this while maintaining usability is a question worth exploring.

# 11 Calculational Properties

We have talked about a dozen recursion schemes, which are recognised common patterns in programming recursive functions. Recognising common patterns help programmers understand a new problem and communicate their solutions with others. Better still, recursion schemes offer rigorous and formal *calculational properties* with which the programmer can manipulate programs in a way similar to manipulate standard mathematical objects such as numbers and polynomials. In this section, we briefly show some of the properties and an example of reasoning about programs using them. We refer to Bird & de Moor (1997) for a comprehensive introduction to this subject and Bird (2010) for more examples of reasoning about and optimising algorithms in this approach.

We focus on *hylomorphisms*, as almost all recursion schemes are a hylomorphism in a certain category. The fundamental property is the unique existence of the solution to a hylo equation given a recursive coalgebra `c` (or dually, a corecursive algebra `a`): for any `x`,

$$\texttt{x} = \texttt{a} \circ \texttt{fmap x} \circ \texttt{c} \iff \texttt{x} = \texttt{hylo a c} \qquad \text{(HyloUniq)}$$

which directly follows the definition of a recursive coalgebra. Instantiating `x` to `hylo a c`, we get the defining equation of `hylo`

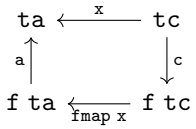
$$\texttt{hylo a c} = \texttt{a} \circ \texttt{fmap}\ (\texttt{hylo a c}) \circ \texttt{c} \qquad \text{(HyloComp)}$$

which is sometimes called the *computation law*, because it tells how to compute `hylo a c` recursively. Instantiating `x` to `id`, we get

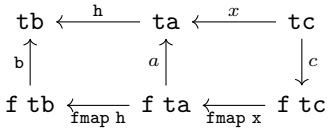
$$\texttt{id} = \texttt{a} \circ \texttt{c} \iff \texttt{id} = \texttt{hylo a c} \qquad \text{(HyloRefl)}$$

called the *reflection law*, which gives a necessary and sufficient condition for `hylo a c` being the identity function. Note that in this law, $\texttt{c} :: \texttt{r} \to \texttt{f r}$ and $\texttt{a} :: \texttt{f r} \to \texttt{r}$ share the same carrier type `r`. A direct consequence of HyloRefl is `cata In` $=$ `id` because `cata a` $=$ `hylo a in`$^\circ$ and `id` $=$ `In` $\circ$ `in`$^\circ$. Dually, we also have `ana Out` $=$ `id`.

An important consequence of HyloUniq is the following *fusion law*. It is easier to describe diagrammatically: The HyloUniq law states that there is exactly one `x`, i.e. `hylo a c`, such that the following diagram commutes (i.e. all paths with the same start and end points give the same result when their edges are composed together):




If we put another *commuting* square beside it,




(7)

the outer rectangle (with top edge `h` $\circ$ `x`) also commutes, and it is also an instance of HyloUniq with coalgebra `c` and algebra `b`. Because HyloUniq states `hylo c b` is the only arrow making the outer rectangle commute, thus `hylo c b` $=$ `h` $\circ$ `x` $=$ `h` $\circ$ `hylo a c`. In summary, the fusion law is:

$$\texttt{h} \circ \texttt{hylo a c} = \texttt{hylo b c} \impliedby \texttt{h} \circ \texttt{a} = \texttt{b} \circ \texttt{fmap h} \qquad \text{(HyloFusion)}$$

and its dual version for corecursive algebra `a` is

$$\texttt{hylo a c} \circ \texttt{h} = \texttt{hylo a d} \impliedby \texttt{c} \circ \texttt{h} = \texttt{fmap h} \circ \texttt{d} \qquad \text{(HyloFusionCo)}$$

where $\texttt{d} :: \texttt{td} \to \texttt{f td}$. Fusion laws combine a funciton after or before a hylomorphism into one hylomorphism, and thus it is widely used for optimisation (Coutts *et al.*, 2007).

We demonstrate how these calculational properties can be used to reason about programs with an example.

**Example 25.** Suppose that `f` is some function on integers satisfying that for any $\texttt{a}, \texttt{b} :: \texttt{Integer}$

$$\texttt{f}\ (\texttt{a} + \texttt{b}) = \texttt{f a} + \texttt{f b} \quad \wedge \quad \texttt{f}\ 0 = 0 \qquad (8)$$

and sum and map are the familiar Haskell functions defined with hylo:

```
type List a = μ (ListF a)
sum :: List Integer → Integer
sum = hylo plus in° where
  plus Nil         = 0
  plus (Cons a b) = a + b
map :: (a → b) → List a → List b
map f = hylo app in° where
  app Nil          = In Nil
  app (Cons a bs) = In (Cons (f a) bs)
```

Let us prove sum∘map f = f∘sum with the calculational properties shown above.

*Proof.* Both sum ∘ map f and f ∘ sum are in the form of a function after a hylomorphism, and thus we can try to use the fusion law to establish

$$\texttt{sum} \circ \texttt{map f} = \texttt{hylo g in}^\circ = \texttt{f} \circ \texttt{sum}$$

for some g. The correct choice of g is

```
g :: ListF Integer → Integer
g Nil         = f 0
g (Cons x y) = f x + y
```

First, sum ∘ map f = sum ∘ hylo app in°, and by HyloFusion,

$$\texttt{sum} \circ \texttt{hylo app in}^\circ = \texttt{hylo g in}^\circ$$

is implied by

$$\texttt{sum} \circ \texttt{app} = \texttt{g} \circ \texttt{fmap sum} \tag{9}$$

Expanding sum on the left-hand side, it is equivalent to

$$(\texttt{plus} \circ \texttt{fmap sum} \circ \texttt{in}^\circ) \circ \texttt{app} = \texttt{g} \circ \texttt{fmap sum} \tag{10}$$

which is an equation of functions

$$\texttt{ListF Integer } (\mu \ (\texttt{ListF Integer})) \to \texttt{Integer}$$

and it can be shown by a case analysis on the input. For Nil, the left-hand side of (10) equals to

```
   plus (fmap sum (in° (app Nil)))
= plus (fmap sum (in° (In Nil)))
= plus (fmap sum Nil)
= plus Nil                          (by definition of fmap for ListF)
= 0
```

and the right-hand side of (10) equals to

$$\texttt{g}\ (\texttt{fmap sum Nil}) = \texttt{g Nil} = \texttt{g}\ 0 = \texttt{f}\ 0$$

and by assumption (8) about `f`, $\texttt{f}\ 0 = 0$. Similarly when the input is `Cons a b`, we can calculate that both sides equal to $\texttt{f a} + \texttt{sum b}$. Thus we have shown Equation (9), and therefore $\texttt{sum} \circ \texttt{hylo app in}^\circ = \texttt{hylo g in}^\circ$.

Similarly, by HyloFusion, $\texttt{f} \circ \texttt{sum} = \texttt{hylo g in}^\circ$ is implied by

$$\texttt{f} \circ \texttt{plus} = \texttt{g} \circ \texttt{fmap f}$$

which can be verified by case analysis on the input: When the input is `Nil`, both sides equal to $\texttt{f}\ 0$. When the input is `Cons a b`, the left-hand side equals to $\texttt{f}\ (\texttt{a} + \texttt{b})$ and the right-hand side is $\texttt{f a} + \texttt{f b}$. By assumption (8) on `f`, $\texttt{f}\ (\texttt{a} + \texttt{b}) = \texttt{f a} + \texttt{f b}$. □

# References


Abel, A., Matthes, R. and Uustalu, T. (2005) Iteration and coiteration schemes for higher-order and nested datatypes. *Theoretical Computer Science* **333**(1-2):3–66.

Adámek, J. (1974) Free algebras and automata realizations in the language of categories. *Commentationes Mathematicae Universitatis Carolinae* **15**(4):589–602.

Bergroth, L., Hakonen, H. and Raita, T. (2000) A survey of longest common subsequence algorithms. *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000* pp. 39–48.

Bird, R. (2010) *Pearls of functional algorithm design.* Cambridge University Press.

Bird, R. and de Moor, O. (1997) *Algebra of Programming.*

Bird, R. and Paterson, R. (1999) Generalised folds for nested datatypes. *Formal Aspects of Computing* **11**(2):200–222.

Bird, R. S. and Meertens, L. G. L. T. (1998) Nested datatypes. *Proceedings of the Mathematics of Program Construction.* MPC '98, p. 52–67. Springer-Verlag.

Capretta, V., Uustalu, T. and Vene, V. (2006) Recursive coalgebras from comonads. *Information and Computation* **204**(4):437–468.

Coutts, D., Leshchinskiy, R. and Stewart, D. (2007) Stream fusion: From lists to streams to nothing at all. *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming.* ICFP '07, p. 315–326. Association for Computing Machinery.

Felleisen, M., Findler, R. B., Flatt, M. and Krishnamurthi, S. (2018) *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press.

Fokkinga, M. (1992) *Law and Order in Algorithmics*. PhD thesis, University of Twente, 7500 AE Enschede, Netherlands.

Fokkinga, M. (1994) *Monadic Maps and Folds for Arbitrary Datatypes*. Memoranda Informatica 94-28. Department of Computer Science, University of Twente.

Fokkinga, M. M. (1990) Tupling and mutumorphisms. *The Squiggolist* **1**(4):81–82.

Gibbons, J. (2000) Generic downwards accumulations. *Sci. Comput. Program.* **37**(1-3):37–65.

Gibbons, J. (2007) Metamorphisms: Streaming representation-changers. *Science of Computer Programming* **65**(2):108–139.

Gibbons, J. (2019) Coding with asymmetric numeral systems. Hutton, G. (ed), *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*. Lecture Notes in Computer Science 11825, pp. 444–465. Springer.

Hagino, T. (1987) *Category Theoretic Approach to Data Types*. PhD thesis, University of Edinburgh.

Hinze, R. (2013) Adjoint folds and unfolds—an extended study. *Science of Computer Programming* **78**(11):2108–2159.

Hinze, R. and Wu, N. (2013) Histo- and dynamorphisms revisited. *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming, WGP '13* pp. 1–12.

Hinze, R. and Wu, N. (2016) Unifying structured recursion schemes - an extended study. *J. Funct. Program.* **26**:47.

Hinze, R., Wu, N. and Gibbons, J. (2013) Unifying structured recursion schemes. *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13* pp. 209–220.

Hinze, R., Wu, N. and Gibbons, J. (2015) Conjugate hylomorphisms – or: The mother of all structured recursion schemes. *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15* pp. 527–538. ACM.

Hoare, C. A. R. (1972) *Chapter II: Notes on Data Structuring*. Academic Press Ltd. p. 83–174.

Hu, Z., Iwasaki, H. and Takeichi, M. (1999) Calculating accumulations. *New Generation Computing* **17**(2):153–173.

Johann, P. and Ghani, N. (2007) Initial algebra semantics is enough! Della Rocca, S. R. (ed), *Typed Lambda Calculi and Applications* pp. 207–222. Springer Berlin Heidelberg.

Johann, P. and Ghani, N. (2008) Foundations for structured programming with gadts. *SIGPLAN Not.* **43**(1):297–308.

Kabanov, J. and Vene, V. (2006) Recursion schemes for dynamic programming. Uustalu, T. (ed), *Mathematics of Program Construction* pp. 235–252. Springer Berlin Heidelberg.

Kmett, E. (2008) *Time for Chronomorphisms*. `http://comonad.com/reader/2008/time-for-chronomorphisms/`. Accessed: 2020-06-15.

Malcolm, G. (1990a) *Algebraic Data Types and Program Transformation*. PhD thesis, University of Groningen.

Malcolm, G. (1990b) Data structures and program transformation. *Science of Computer Programming* **14**(2-3):255–280.

Meertens, L. (1988) First steps towards the theory of rose trees. *CWI, Amsterdam*.

Meertens, L. (1992) Paramorphisms. *Formal Aspects of Computing* **4**(5):413–424.

Meijer, E., Fokkinga, M. and Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. Hughes, J. (ed), *5th ACM Conference on Functional Programming Languages and Computer Architecture, FPCA'91*, vol. 523, pp. 124–144.

Okasaki, C. (1995) Purely functional random-access lists. *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95, p. 86–95. Association for Computing Machinery.

Pardo, A. (1998) Monadic corecursion —definition, fusion laws, and applications—. *Electron. Notes Theor. Comput. Sci.* **11**(C):105–139.

Pardo, A. (2002) Generic accumulations. Gibbons, J. and Jeuring, J. (eds), *Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Programming*. International Federation for Information Processing 115, pp. 49–78. Kluwer Academic Publishers.

Pardo, A. (2005) Combining datatypes and effects. Vene, V. and Uustalu, T. (eds), *Advanced Functional Programming* pp. 171–209. Springer Berlin Heidelberg.

Paykin, J. and Zdancewic, S. (2017) The linearity monad. *SIGPLAN Not.* **52**(10):117–132.

Uustalu, T. and Vene, V. (1999) Primitive (co)recursion and course-of-value (co)iteration, categorically. *Informatica* **10**(1):5–26.

Uustalu, T. and Vene, V. (2008) Comonadic notions of computation **203**(5):263–284.

Uustalu, T., Vene, V. and Pardo, A. (2001) Recursion schemes from comonads **8**(3):366–390.

Vene, V. and Uustalu, T. (1998) Functional programming with apomorphisms (corecursion). *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics* **47**(3):147–161.