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

Zhixuan Yang and Nicolas Wu

Imperial College London, United Kingdom
{s.yang20,n.wu}@imperial.ac.uk

Abstract. *Structured recursion schemes* have been widely used in constructing, optimising, and reasoning about programs over inductive and coinductive datatypes. Their plain forms, *catamorphisms* and *anamorphisms*, are restricted in expressiveness. Thus many generalisations have been proposed, which further lead to several unifying frameworks of structured recursion schemes. However, the existing work on unifying frameworks typically focuses on the categorical foundation, and thus is perhaps inaccessible to practitioners who are willing to apply recursion schemes in practice but are not versed in category theory. To fill this gap, this expository paper introduces structured recursion schemes from a practical point of view: a variety of recursion schemes are motivated and explained in contexts of concrete programming examples. The categorical duals of these recursion schemes are also explained.

Keywords: Recursion schemes · Generic programming · (Un)Folds · (Co)Inductive datatypes · Equational reasoning · Haskell

1 Introduction

Since the introduction of *catamorphisms* by Malcolm [38], they have been a valuable item in the toolkit of functional programmers for many of their benefits: by expressing recursive programs as catamorphisms, the structure of the programs is made obvious; the recursion is ensured to terminate; and the program can be reasoned about using the calculational properties of catamorphisms.

These benefits motivated a whole research agenda concerned with identifying *structural recursion schemes* that capture the pattern of many other recursive functions that did not quite fit as catamorphisms. Just as with catamorphisms, these structural recursion schemes attracted attention since they make termination or productivity 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 catamorphisms was paramorphisms [40], about which Meertens talked at the 41st IFIP Working Group 2.1 (WG2.1) meeting in Burton, UK (1990). Paramorphisms describe recursive functions in which the body of structural recursion has access to not only the (recursively computed) sub-results of the input, but also the original subterms of the input.

Then came a whole zoo of morphisms. Mutumorphisms [14], which are pairs of mutually recursive functions; zygomorphisms [37], which consist of a main recursive function and an auxiliary one on which it depends; monadic catamorphisms [13], which are recursive functions that also cause computational effects; histomorphisms [46], in which the body has access to the recursive images of all subterms, not just the immediate ones; so-called generalised folds [6], which use polymorphic recursion to handle nested datatypes; and then there were generic accumulations [44], 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 [49], which may generate subterms monolithically rather than step by step; futumorphisms [46], which may generate multiple levels of a subterm in a single step, rather than just one; and many other anonymous schemes that dualise 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 (rsfcs for short) by Uustalu et al. [48]. Comonads capture the general idea of 'evaluation in context' [47], 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 [22] 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, zygomorphisms, and generalised folds. Later, it was observed that adjoint folds could be used to subsume rsfcs [25], which in turn draws on material from [22].

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, Wu, and Gibbons [26]'s *conjugate hylomorphisms*, which WG2.1 dubbed *mamamorphisms*. This worked by viewing all recursion schemes as specialised forms of hylomorphisms, and showing that they are well-defined hylomorphisms using adjunctions and conjugate natural transformations.

1.3 Contributions

The existing literature [25, 26, 47] on unifying accounts to structured recursion schemes has focused on the categorical foundation of recursion schemes rather than their motivations or applications, and thus is perhaps not quite useful for practitioners who would like to learn about recursion schemes and apply them in practice. To fill the gap, this paper introduces the zoo of recursion schemes by putting them in programming contexts. Hence this paper is not meant to be a regular research paper presenting new results, but a survey of recursion schemes in functional programming. The paper is organised as follows.

- Section 2 explains the idea of modelling (co)inductive datatypes as fixed points of functors, which makes generic recursion schemes possible.
- Section 3 explains the three fundamental recursion schemes: *catamorphisms*, which compute values by consuming inductive data; *anamorphisms*, which build coinductive data from values; and their common generalisation, *hylomorphisms*, which build data from values and consume them.
- Section 4 introduces structural recursion with an accumulating parameter.
- Section 5 is about mutual recursion on inductive datatypes, known as *mutumorphisms*, and their unnamed duals, which build mutually defined coinductive datatypes from a single value.
- Section 6 talks about primitive recursion, known as *paramorphisms*, featuring the ability to access both the original subterms and the corresponding output in the recursive function. Their corecursive counterpart, *apomorphisms*, and a generalisation, *zygomorphisms*, are also shown.
- Section 7 discusses the so-called course-of-values recursion, *histomorphisms*, featuring the ability to access the results of all direct and indirect subterms in the body of recursive function, which is typically necessary in dynamic programming. Several related schemes, *futumorphisms*, *dynamorphisms*, and *chronomorphisms* are briefly discussed.
- Section 8 introduces recursion schemes that cause computational effects.
- Section 9 explains recursion schemes on nested datatypes and GADTs.
- Section 11 discusses two general recipes for finding more recursion schemes.
- Finally, Section 10 briefly demonstrates how one can do equational reasoning about programs using calculational properties of recursion schemes.

The recursion schemes that we will see in this paper are summarised in Table 1. Sections 3–9 are loosely ordered by their complexity, rather than by their time appearing in the literature, and these sections are mutually independent so can be read in an arbitrary order. A common pattern in these sections is that we start with a concrete programming example, from which we distill a recursion scheme, followed by more examples. Then we consider their dual corecursion scheme and hylomorphic generalisation.

Table 1: Recursion schemes introduced in this paper

Scheme	Type Signature	Usage
Catamorphism	$(f\ a \to a) \to \mu\, f \to a$	Consuming inductive data
Anamorphism	$(c \to f\ c) \to c \to \nu\, f$	Generating coinductive data
Hylomorphism	$(f\ a \to a) \to (c \to f\ c) \to c \to a$	Generating followed by consuming
Accumulation	$(\forall x. f\ x \to p \to f\ (x, p)) \to (f\ a \to p \to a) \to \mu\, f \to p \to a$	Recursion with an accumulating parameter
Mutumorphism	$(f\ (a, b) \to a) \to (f\ (a, b) \to b) \to (\mu\, f \to a, \mu\, f \to b)$	Mutual recursion on inductive data
Dual of mutumorphism	$(c \to f\ c\ c) \to (c \to g\ c\ c) \to c \to (\nu_1\ f\ g, \nu_2\ f\ g)$	Generating mutually defined coinductive data
Paramorphism	$(f\ (\mu\, f, a) \to a) \to \mu\, f \to a$	Primitive recursion, i.e. access to original input
Apomorphism	$(c \to f\ (\mathit{Either}\ (\nu\, f)\ c)) \to c \to \nu\, f$	Early termination of generation
Zygomorphism	$(f\ (a, b) \to a) \to (f\ b \to b) \to \mu\, f \to a$	Recursion with auxiliary information
Histomorphism	$(f\ (\mathit{Cofree}\ f\ a) \to a) \to \mu\, f \to a$	Access to all sub-results
Dynamorphism	$(f\ (\mathit{Cofree}\ f\ a) \to a) \to (c \to f\ c) \to c \to a$	Dynamic programing
Futumorphism	$(c \to f\ (\mathit{Free}\ f\ c)) \to c \to \nu\, f$	Generating multiple layers
Monadic catamorphism	$(\forall x. f\ (m\ x) \to m\ (f\ x)) \to (f\ a \to m\ a) \to \mu\, f \to m\ a$	Recursion causing computational effects
Indexed catamorphism	$(f\ a \dot\to a) \to \dot\mu\, f \dot\to a$	Consuming nested datatypes and GADTs

2 Datatypes and Fixed Points

This paper assumes basic familiarity with Haskell as we use it to present all examples and recursion schemes, but we do not assume any knowledge of category theory. In this section, we briefly review the prerequisite of recursive schemes—recursive datatypes, viewed as fixed points of functors.

Datatypes *Algebraic data types* (ADTs) in Haskell allow the programmer to create new datatypes from existing ones. For example, the type *List a* of lists of elements of type *a* can be declared as follows:

$$\mathbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \tag{1}$$

which means that an element of $List\ a$ is exactly Nil or $Cons\ x\ xs$ for all $x :: a$ and $xs :: List\ a$. Similarly, the type $Tree\ a$ of binary trees whose nodes are labelled with a-elements can be declared as follows:

$$\mathbf{data}\ Tree\ a = Empty \mid Node\ (Tree\ a)\ a\ (Tree\ a) \tag{2}$$

In definitions like $List\ a$ and $Tree\ a$, the datatypes being defined also appear on the right-hand side of the declaration, so they are *recursive types*. Moreover, $List\ a$ and $Tree\ a$ are among a special family of recursive types, called *inductive datatypes*, meaning that they are fixed points of *functors*.

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

$$\mathbf{class}\ Functor\ f\ \mathbf{where}\ fmap :: (a \to b) \to f\ a \to f\ b$$

Additionally, $fmap$ is expected to satisfy two functor laws:

$$fmap\ id = id \qquad fmap\ (f \circ g) = fmap\ f \circ fmap\ g$$

for all functions $f :: B \to C$ and $g :: A \to B$.

Given a functor f, we call a function of type $f\ a \to a$, for some type a, an *f-algebra*, and a function of type $a \to f\ a$ an *f-coalgebra*. In either case, 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 isomorphic to $f\ p$. In the set theoretic semantics, a functor may have more than one fixed points: the *least fixed point*, denoted by $\mu\, f$, is the set of f-branching trees of *finite* depths, while the *greatest fixed point*, denoted by $\nu\, f$, is intuitively the set of f-branching trees of *possibly infinite* depths.

However, due to the fact that Haskell is a lazy language with general recursion, the least and greatest fixed points of a Haskell functor f *coincide* as the following datatype of possibly infinite f-branching trees:

$$\mathbf{newtype}\ Fix\ f = In\ \{\ out :: f\ (Fix\ f)\}$$

Although Haskell allows general recursion, the point of using structural recursion is precisely avoiding general recursion whenever possible, since general recursion is typically tricky to reason about. Hence in this paper we use Haskell as if it is a total programming language, by making sure all recursive functions that we use are structural recursion. And we distinguish the least and greatest fixed points as two datatypes:

$$\mathbf{newtype}\ \mu\, f = In\ (f\ (\mu\, f)) \qquad \mathbf{newtype}\ \nu\, f = Out^{\circ}\ (f\ (\nu\, f))$$

While these two datatypes $\mu\ f$ and $\nu\ f$ are the same datatype declaration, we mentally understand $\mu\ f$ as the type of *finite* f-branching trees, and $\nu\ f$ as the type of *possibly infinite* one, as in the set-theoretic semantics. Making such a nominal distinction is not entirely pointless: the type system at least ensures that we never accidentally misuse an element of $\nu\, f$ as an element of $\mu\, f$, unless we make an explicit conversion. But it is our own responsibility to make sure that we never construct an infinite element in $\mu\, f$ using general recursion.

Example 1. The datatypes (1) and (2) that we saw earlier are isomorphic to fixed points of functors *ListF* and *TreeF* defined as follows (with the evident *fmap* that can be derived by `GHC` automatically[1]):

$$\begin{array}{l} \mathbf{data}\ \mathit{ListF}\ a\ x = \mathit{Nil} \mid \mathit{Cons}\ a\ x\ \ \mathbf{deriving}\ \mathit{Functor} \\ \mathbf{data}\ \mathit{TreeF}\ a\ x = \mathit{Empty} \mid \mathit{Node}\ x\ a\ x\ \mathbf{deriving}\ \mathit{Functor} \end{array}$$

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

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

$$\begin{array}{ll} \mathit{conv}_\mu :: [a] \to \mu\ (\mathit{ListF}\ a) & \mathit{conv}^\circ_\mu :: \mu\ (\mathit{ListF}\ a) \to [a] \\ \mathit{conv}_\mu\ [\,] = \mathit{In}\ \mathit{Nil} & \mathit{conv}^\circ_\mu\ (\mathit{In}\ \mathit{Nil}) = [\,] \\ \mathit{conv}_\mu\ (a : as) = \mathit{In}\ (\mathit{Cons}\ a\ (\mathit{conv}_\mu\ as)) & \mathit{conv}^\circ_\mu\ (\mathit{In}\ (\mathit{Cons}\ a\ as)) = a : \mathit{conv}^\circ_\mu\ as \end{array}$$

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

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

The type checker of Haskell will then ensure that we never pass a value of $\nu\ (\mathit{ListF}\ a)$ to this function.

Initial and Final (Co)Algebra The constructor $\mathit{In} :: f\ (\mu\ f) \to \mu\ f$ is an f-algebra with carrier $\mu\, f$, and it has an inverse $\mathit{in}^\circ :: \mu\, f \to f\ (\mu\, f)$ defined as

$$\mathit{in}^\circ\ (\mathit{In}\ x) = x$$

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

$$\mathit{out}\ (\mathit{Out}^\circ\ x) = x$$

is an f-coalgebra with carrier $\nu\, f$.

What is special with *In* and *out* is that *In* is the so-called *initial algebra* of f, in the sense that it has the nice property that for any f-algebra $\mathit{alg} :: f\ a \to a$, there is exactly one function $h :: \mu\ f \to a$ such that

$$h \circ \mathit{In} = \mathit{alg} \circ \mathit{fmap}\ h \tag{3}$$

[1] It requires the `DeriveFunctor` extension of `GHC` to derive functors automatically.

Dually, *out* is called the *final coalgebra* of f since for any f-coalgebra $coalg :: c \to f\ c$, there is exactly one function $h :: c \to \nu\, f$ such that

$$out \circ h = fmap\ h \circ coalg \tag{4}$$

The h's in (3) and (4) are precisely the two fundamental recursion schemes, *catamorphisms* and *anamorphism*, which we will talk about in the next section.

3 Fundamental Recursion Schemes

Most if not all programs are about processing data, and as Hoare [27] 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* [11, 18]. The determination is abstracted as recursion schemes for programs processing recursive datatypes.

In this section, we look at the three fundamental recursion schemes: catamorphisms, in which the program is structured by its input; anamorphisms, in which the program is structured by its output; and hylomorphisms, 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 the first example, consider the program computing the length of a list:

$$\begin{array}{lll} length :: [\,a\,] & \to Integer \\ length\ \ [\,] & = 0 \\ length\ \ (x : xs) & = 1 + length\ xs \end{array}$$

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 reflected 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 usage of xs.

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

$$\begin{array}{l} foldr :: (a \to b \to b) \to b \to [\,a\,] \to b \\ foldr\ f\ \ e\ [\,] \qquad\quad = e \\ foldr\ f\ \ e\ (x : xs) = f\ x\ (foldr\ f\ \ e\ xs) \end{array}$$

which is very useful in list processing. As a fold, $length = foldr\ (\lambda\_\ l \to 1 + l)\ 0$. The frequently used function *map* is also a fold:

$$\begin{array}{l} map :: (a \to b) \to [\,a\,] \to [\,b\,] \\ map\ f = foldr\ (\lambda x\ xs \to f\ x : xs)\ [\,] \end{array}$$

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

$$\begin{aligned}&\mathit{concat} :: [[a]] \to [a]\\&\mathit{concat} = \mathit{foldr}\ (+\!\!+)\ [\,]\end{aligned}$$

By expressing structural recursive functions as folds, their structure becomes clearer, similarly in spirit to the well accepted practice of structuring programs with if-conditionals and for-/while-loops in imperative languages.

Recursion Scheme 1 (*cata*). Folds on lists can be readily generalised to the generic setting, where the shape of the datatype is determined by a functor [38, 12, 19]. The resulting recursion scheme is called *catamorphisms*:

$$\begin{aligned}&\mathit{cata} :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to \mu\, f \to a\\&\mathit{cata}\ \mathit{alg} = \mathit{alg} \circ \mathit{fmap}\ (\mathit{cata}\ \mathit{alg}) \circ \mathit{in}^{\circ}\end{aligned}$$

Intuitively, catamorphisms gradually break down the inductively defined input data, computing the result by replacing constructors with the given algebra *alg*. The name *cata* dubbed by Meertens [39] is from Greek κατά meaning 'downwards along' or 'according to'. A notation for *cata alg* is the so-called banana bracket $(\!|\, \mathit{alg}\, |\!)$ introduced by Meijer et al. [41], but we will not use this style of notation in this paper, as we will not have enough squiggly brackets for all recursion schemes that we will see.

Example 2. By converting the builtin list type $[a]$ to the initial algebra of *ListF* as in Example 1, we can recover *foldr* from *cata* as follows:

$$\begin{aligned}&\mathit{foldr}' :: (a \to b \to b) \to b \to [a] \to b\\&\mathit{foldr}'\ f\ e = \mathit{cata}\ \mathit{alg} \circ \mathit{conv}_{\mu}\ \textbf{where}\\&\quad \mathit{alg}\ \mathit{Nil} \qquad\quad\ = e\\&\quad \mathit{alg}\ (\mathit{Cons}\ a\ x) = f\ a\ x\end{aligned}$$

Now we can also fold datatypes other than lists, such as binary trees:

$$\begin{aligned}&\mathit{size} :: \mu\ (\mathit{TreeF}\ e) \to \mathit{Integer}\\&\mathit{size} = \mathit{cata}\ \mathit{alg}\ \textbf{where}\\&\quad \mathit{alg} :: \mathit{TreeF}\ a\ \mathit{Integer} \to \mathit{Integer}\\&\quad \mathit{alg}\ \mathit{Empty} \qquad\ = 0\\&\quad \mathit{alg}\ (\mathit{Node}\ l\ e\ r) = l + 1 + r\end{aligned}$$

Example 3 (Interpreting DSLs). The 'killer application' of catamorphisms is using them to implement *domain-specific languages* (DSLs) [30]. The abstract syntax of a DSL can usually be modelled as an inductive datatype, and then the (denotational) semantics of the DSL can be given as a catamorphism. The semantics given in this way is *compositional*, meaning that the semantics of a program is determined the semantics of its immediate sub-parts—exactly the pattern of catamorphisms.

As a small example here, 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 abstract syntax of the language can be modelled as the initial algebra $\mu\ (ProgF\ s\ a)$ of the following functor:

$$\mathbf{data}\ ProgF\ s\ a\ x = Ret\ a \mid Put\ (Int, s)\ x \mid Get\ Int\ (s \to x)$$

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

$$\begin{aligned} &p_1 :: \mu\ (ProgF\ Int\ Int) \\ &p_1 = In\ (Get\ 0\ (\lambda s \to (In\ (Put\ (0, s+1)\ (In\ (Ret\ s)))))) \end{aligned}$$

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 $ret = In \circ Ret$ are defined.

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

$$\begin{aligned} &interp :: \mu\ (ProgF\ s\ a) \to (Map\ Int\ s \to a) \\ &interp = cata\ handle\ \mathbf{where} \\ &\quad handle\ (Ret\ a) \qquad\quad\ = \lambda\_ \to a \\ &\quad handle\ (Put\ (i, x)\ k) = \lambda m \to k\ (update\ m\ i\ x) \\ &\quad handle\ (Get\ i\ k) \qquad = \lambda m \to k\ (m\ !\ i)\ m \end{aligned}$$

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 run programs:

$$*\!>\ \ interp\ p_1\ (fromList\ [(0, 100)]) \quad \text{-- outputs 100}$$

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:

$$\begin{aligned} &\mathbf{data}\ Person = Person\ \{name :: String, addr :: String, phone :: [Int]\} \\ &mkEntry :: StaffInfo \to Person \\ &mkEntry\ i = Person\ n\ a\ p\ \mathbf{where}\ n = ...; a = ...; p = ... \end{aligned}$$

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, called *(structural) corecursion* [18]. Consider the following program generating evenly spaced numbers over an interval.

$$
\begin{array}{l}
\mathit{linspace} :: \mathit{RealFrac}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Integer} \rightarrow [\,a\,] \\
\mathit{linspace}\ s\ e\ n = \mathit{gen}\ s\ \mathbf{where} \\
\quad \mathit{step} = (e - s)\ /\ \mathit{fromIntegral}\ (n + 1) \\
\quad \mathit{gen}\ i \\
\qquad \mid i < e \qquad\quad = i : \mathit{gen}\ (i + \mathit{step}) \\
\qquad \mid \mathit{otherwise} = [\,]
\end{array}
$$

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*:

$$
\begin{array}{l}
\mathit{unfoldr} :: (b \rightarrow \mathit{Maybe}\ (a, b)) \rightarrow b \rightarrow [\,a\,] \\
\mathit{unfoldr}\ g\ s = \mathbf{case}\ g\ s\ \mathbf{of} \\
\quad (\mathit{Just}\ (a, s')) \rightarrow a : \mathit{unfoldr}\ g\ s' \\
\quad \mathit{Nothing} \qquad\quad \rightarrow [\,]
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{linspace}\ s\ e\ n = \mathit{unfoldr}\ \mathit{gen}\ s\ \mathbf{where} \\
\quad \mathit{step} = (e - s)\ /\ \mathit{fromIntegral}\ (n + 1) \\
\quad \mathit{gen}\ i = \mathbf{if}\ i < e\ \mathbf{then}\ \mathit{Just}\ (i, i + \mathit{step})\ \mathbf{else}\ \mathit{Nothing}
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{from} :: \mathit{Integer} \rightarrow [\,\mathit{Integer}\,] \\
\mathit{from} = \mathit{unfoldr}\ (\lambda n \rightarrow \mathit{Just}\ (n, n + 1))
\end{array}
$$

generates the infinite list of all integers from n.

Recursion Scheme 2 (*ana*). In the same way that *cata* generalises *foldr*, *unfoldr* can be generalised from lists to arbitrary coinductive datatypes. The (co)recursion scheme is called *anamorphisms*:

$$
\begin{array}{l}
\mathit{ana} :: \mathit{Functor}\ f \Rightarrow (c \rightarrow f\ c) \rightarrow c \rightarrow \nu\ f \\
\mathit{ana}\ \mathit{coalg} = \mathit{Out}^{\circ} \circ \mathit{fmap}\ (\mathit{ana}\ \mathit{coalg}) \circ \mathit{coalg}
\end{array}
$$

The name is due to Meijer et al. [41]: *ana* from the Greek preposition ανά means 'upwards', dual to *cata* meaning 'downwards'.

Example 4. Modulo the isomorphism between $[\,a\,]$ and $\nu\ (\mathit{ListF}\ a)$, *unfoldr* is an anamorphism:

$$
\begin{array}{l}
\mathit{unfoldr}' :: (b \rightarrow \mathit{Maybe}\ (a, b)) \rightarrow b \rightarrow [\,a\,] \\
\mathit{unfoldr}'\ g = \mathit{conv}_{\nu}^{\circ} \circ \mathit{ana}\ \mathit{coalg}\ \mathbf{where} \\
\quad \mathit{coalg}\ b = \mathbf{case}\ g\ b\ \mathbf{of}\ \mathit{Nothing} \qquad\quad \rightarrow \mathit{Nil} \\
\qquad\qquad\qquad\qquad\quad\ \ (\mathit{Just}\ (a, b)) \rightarrow \mathit{Cons}\ a\ b
\end{array}
$$

Example 5. A more interesting example of anamorphisms is merging a pair of ordered lists:

$$\begin{aligned}
&\mathit{merge} :: \mathit{Ord}\ a \Rightarrow (\nu\ (\mathit{ListF}\ a), \nu\ (\mathit{ListF}\ a)) \rightarrow \nu\ (\mathit{ListF}\ a)\\
&\mathit{merge} = \mathit{ana}\ c\ \textbf{where}\\
&\quad c\ (x, y)\\
&\qquad \mid \mathit{null}_\nu\ x \wedge \mathit{null}_\nu\ y = \mathit{Nil}\\
&\qquad \mid \mathit{null}_\nu\ y \vee \mathit{head}_\nu\ x < \mathit{head}_\nu\ y\\
&\qquad\qquad\qquad = \mathit{Cons}\ (\mathit{head}_\nu\ x)\ (\mathit{tail}_\nu\ x, y)\\
&\qquad \mid \mathit{otherwise} = \mathit{Cons}\ (\mathit{head}_\nu\ y)\ (x, \mathit{tail}_\nu\ y)
\end{aligned}$$

where null_ν, head_ν and tail_ν are the corresponding list functions for $\nu\ (\mathit{ListF}\ a)$.

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:

$$\begin{aligned}
&\mathit{qsort} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a]\\
&\mathit{qsort}\ [\,] \qquad\quad = [\,]\\
&\mathit{qsort}\ (a : \mathit{as}) = \mathit{qsort}\ l + [a] + \mathit{qsort}\ r\ \textbf{where}\\
&\quad l = [b \mid b \leftarrow \mathit{as}, b < a]\\
&\quad r = [b \mid b \leftarrow \mathit{as}, b \geqslant a]
\end{aligned}$$

Although the input $[a]$ is an inductive datatype, *qsort* is not a catamorphism as the recursion is not performed on the sub-list *as*. Neither is it an anamorphism, since the output is not produced in the head-and-recursion manner.

Felleisen et al. [11] referred to this form of recursive programs as *generative recursion* since the input $a : \mathit{as}$ is used to generate a set of sub-problems, namely l and r, which are recursively solved, and their solutions are combined to solve the overall problem $a : \mathit{as}$. The structure of computing *qsort* is manifested in the following rewrite of *qsort*:

$$\begin{aligned}
&\mathit{qsort}' :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow [a]\\
&\mathit{qsort}' = \mathit{combine} \circ \mathit{fmap}\ \mathit{qsort}' \circ \mathit{partition}\\
&\mathit{partition} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow \mathit{TreeF}\ a\ [a]\\
&\mathit{partition}\ [\,] \qquad\quad = \mathit{Empty}\\
&\mathit{partition}\ (a : \mathit{as}) = \mathit{Node}\ [b \mid b \leftarrow \mathit{as}, b < a]\ a\ [b \mid b \leftarrow \mathit{as}, b \geqslant a]\\
&\mathit{combine} :: \mathit{TreeF}\ a\ [a] \rightarrow [a]\\
&\mathit{combine}\ \mathit{Empty} \qquad\quad = [\,]\\
&\mathit{combine}\ (\mathit{Node}\ l\ x\ r) = l + [x] + r
\end{aligned}$$

The functor $\mathit{TreeF}\ a\ x = \mathit{Empty} \mid \mathit{Node}\ x\ a\ x$ governs the recursive call structure, which is a binary tree. The $(\mathit{TreeF}\ a)$-coalgebra *partition* divides a problem (if not trivial) into two sub-problems, and the $(\mathit{TreeF}\ a)$-algebra *combine* concatenates the results of sub-problems to form a solution to the whole problem.

Recursion Scheme 3 (*hylo*)**.** Abstracting the pattern of divide-and-conquer algorithms like *qsort* results in the recursion scheme *hylomorphisms*:

$$\mathit{hylo} :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to (c \to f\ c) \to c \to a$$
$$\mathit{hylo}\ a\ c = a \circ \mathit{fmap}\ (\mathit{hylo}\ a\ c) \circ c$$

The name is due to Meijer et al. [41] and is a term from Aristotelian philosophy that objects are compounded of matter and form, where the prefix hylo- (Greek ὕλη-) means ‘matter’.

Hylomorphisms are highly expressive. In fact, all recursion schemes in this paper can be defined as special cases of hylomorphisms, and Hu et al. [29] showed a mechanical way to transform almost all recursive functions in practice into hylomorphisms. In particular, hylomorphisms subsume both catamorphisms and anamorphisms: for all $\mathit{alg} :: f\ a \to a$ and $\mathit{coalg} :: c \to f\ c$, we have

$$\mathit{cata}\ \mathit{alg} = \mathit{hylo}\ \mathit{alg}\ \mathit{in}^{\circ} \qquad \text{and} \qquad \mathit{ana}\ \mathit{coalg} = \mathit{hylo}\ \mathit{Out}^{\circ}\ \mathit{coalg}.$$

However, the expressiveness of *hylo* comes at a cost: even when both $\mathit{alg} :: f\ a \to a$ and $\mathit{coalg} :: c \to f\ c$ 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). Intuitively, it is because the coalgebra *coalg* may infinitely generate sub-problems 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

$$\mathit{geo} :: \mathit{Integer} \to \mathit{ListF}\ \mathit{Double}\ \mathit{Integer}$$
$$\mathit{geo}\ n = \mathit{Cons}\ (1\ /\ \mathit{fromIntegral}\ n)\ (2 * n)$$

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

$$\mathit{sum} :: \mathit{ListF}\ \mathit{Double}\ \mathit{Double} \to \mathit{Double}$$
$$\mathit{sum}\ \mathit{Nil} \qquad\quad = 0$$
$$\mathit{sum}\ (\mathit{Cons}\ n\ p) = n + p$$

which sums a sequence. Both *geo* and *sum* are total Haskell functions, but the function $\mathit{zeno} = \mathit{hylo}\ \mathit{sum}\ \mathit{geo}$ diverges for all input $i :: \mathit{Integer}$. (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.)

Recover Totality One way to tame the well-definedness of *hylo* is to consider coalgebras $\mathit{coalg} :: c \to f\ c$ with the special properties that the equation

$$x = \mathit{alg} \circ \mathit{fmap}\ x \circ \mathit{coalg} \tag{5}$$

has a unique solution $x :: c \to a$ for *all* algebras $\mathit{alg} :: f\ a \to a$. Such coalgebras are called *recursive coalgebras*. Dually, one can also consider *corecursive algebras* *alg* that make (5) have a unique solution for all *coalg*. For example, the coalgebra $in^{\circ} :: \mu\, f \to f\ (\mu\, f)$ is recursive, since the equation

$$x = alg \circ fmap\ x \circ in^{\circ} \qquad \Longleftrightarrow \qquad x \circ In = alg \circ fmap\ x$$

has a unique solution by property (3) of the initial algebra. Dually, $Out^{\circ} :: f\ (\nu\, f) \to \nu\, f$ is a corecursive algebra by (4).

Besides these two basic examples, quite some effort has been made in searching for more recursive coalgebras (and corecursive algebras): Capretta et al. [9] first show that it is possible to construct new recursive coalgebras from existing ones using comonads, and later Hinze et al. [26] show a more general technique using adjunctions and conjugate pairs. With these techniques, all recursion schemes on (co)inductive datatypes presented in this paper can be uniformly understood as hylomorphisms with a recursive coalgebra or corecursive algebra. However, we shall not emphasise this perspective in this paper since it sometimes involves non-trivial category theory to massage a recursion scheme into a hylomorphism with a recursive coalgebras (or a corecursive algebra).

Example 7. The coalgebra $partition :: [a] \to TreeF\ a\ [a]$ above is recursive (when only finite lists are allowed as input). This can be proved by an easy inductive argument: for any total $alg :: TreeF\ a\ b \to b$, suppose that $x :: [a] \to b$ satisfies

$$x = alg \circ fmap\ x \circ partition. \tag{6}$$

Given any finite list l, we show $x\ l$ is determined by *alg* by an induction on l. For the base case $l = [\,]$, we have

$$x\ [\,] = alg\ (fmap\ x\ (partition\ [\,])) = alg\ (fmap\ x\ Empty) = alg\ Empty.$$

For the inductive case $l = y : ys$, we have

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

where $ls = [b \mid b \leftarrow as, b < a]$ and $rs = [b \mid b \leftarrow as, b \geqslant a]$ are strictly smaller than $l = (a : as)$, and thus $x\ ls$ and $x\ rs$ are uniquely determined by *alg*. Consequently, $x\ (y : ys)$ is uniquely determined by *alg*. Thus we conclude that x satisfying the hylo equation (6) is unique.

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

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

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

$$\begin{aligned}&\mathit{meta} :: (\mathit{Functor}\ f, \mathit{Functor}\ g) \Rightarrow (c \to g\ c) \to (f\ c \to c) \to \mu\, f \to \nu\, g\\&\mathit{meta}\ \mathit{coalg}\ \mathit{alg} = \mathit{ana}\ \mathit{coalg} \circ \mathit{cata}\ \mathit{alg}\end{aligned}$$

which is called *metamorphisms* by Gibbons [16] because it *metamorphoses* data represented by functor f to g. Unlike hylomorphisms, the producing and consuming phases in metamorphisms cannot be straightforwardly fused into a single recursive function. Gibbons [16, 17] gives conditions for doing this when f is *ListF*, but we will not expand on this in this paper.

4 Accumulations

Accumulating parameters are a well known technique for optimising recursive functions. An example is optimising the following *reverse* function that runs in quadratic time (due to the fact that $xs +\!\!+\ ys$ runs in $\mathcal{O}(\mathit{length}\ xs)$ time)

$$\begin{aligned}&\mathit{reverse} :: [\,a\,] \to [\,a\,]\\&\mathit{reverse}\quad [\,] \qquad\; = [\,]\\&\mathit{reverse}\quad (x : xs) = \mathit{reverse}\ xs +\!\!+ [\,x\,]\end{aligned}$$

to linear time by first generalising the function with an additional parameter—an *accumulating parameter ys*:

$$\begin{aligned}&\mathit{revCat} :: [\,a\,] \to [\,a\,] \to [\,a\,]\\&\mathit{revCat}\ ys\ [\,] \qquad = ys\\&\mathit{revCat}\ ys\ (x : xs) = \mathit{revCat}\ (x : ys)\ xs\end{aligned}$$

and it specialises to *reverse* by letting $\mathit{reverse} = \mathit{recCat}\ [\,]$. This pattern of scanning a list from left to right and accumulating a parameter at the same time is abstracted as the Haskell function *foldl*:

$$\begin{aligned}&\mathit{foldl} :: (b \to a \to b) \to b \to [\,a\,] \to b\\&\mathit{foldl}\ f\ e\ [\,] \qquad = e\\&\mathit{foldl}\ f\ e\ (x : xs) = \mathit{foldl}\ f\ (f\ e\ x)\ xs\end{aligned}$$

which specialises to *revCat* for $f = \lambda ys\ x \to x : ys$. Similar to *foldr*, *foldl* follows the structure of the input—a base case for $[\,]$ and an inductive case for $x : xs$. What differs is that *foldl* has an argument e varied during the recursion.

The pattern of accumulation is not limited to lists. For example, consider writing a program that transforms a binary tree labelled with integers to the tree whose nodes are relabelled with the *sum* of the labels along the path from the root in the original tree. A natural idea is to keep an accumulating parameter for the sum of labels from the root:

$$\begin{array}{l} \mathit{relabel} :: \mu\ (\mathit{TreeF\ Integer}) \to \mathit{Integer} \to \mu\ (\mathit{TreeF\ Integer}) \\ \mathit{relabel}\ (\mathit{In\ Empty}) \qquad\quad s = \mathit{In\ Empty} \\ \mathit{relabel}\ (\mathit{In}\ (\mathit{Node}\ l\ e\ r))\ s = \mathit{In}\ (\mathit{Node}\ (\mathit{relabel}\ l\ s')\ s'\ (\mathit{relabel}\ r\ s')) \\ \quad \textbf{where}\ s' = s + e \end{array}$$

In the *Node* case, the current accumulating parameter s is updated to s' for both of the subtrees, but we can certainly accumulate the parameter for the subtrees using other accumulating strategies. In general, an accumulating strategy can be captured as a function of type

$$\forall x. \mathit{TreeF\ Integer}\ x \to \mathit{Integer} \to \mathit{TreeF\ Integer}\ (x, \mathit{Integer})$$

For example, the strategy for *relabel* is

$$\begin{array}{l} st_{relabel}\ \mathit{Empty}\ s \qquad\quad = \mathit{Empty} \\ st_{relabel}\ (\mathit{Node}\ l\ e\ r)\ s = \mathit{Node}\ (l, s')\ e\ (r, s')\ \textbf{where}\ s' = s + e \end{array}$$

Notice that the type above is polymorphic over x, which means that the accumulation cannot depend on the subtrees. This is not strictly necessary, but it reflects the pattern of most accumulations in practice.

Recursion Scheme 4 (*accu*). Abstracting the idea for a generic initial algebra, we obtain the recursion scheme *accumulations* [28, 44, 15][2]:

$$\begin{array}{l} \mathit{accu} :: \mathit{Functor}\ f \Rightarrow (\forall x. f\ x \to p \to f\ (x, p)) \\ \qquad\qquad\qquad\quad \to (f\ a \to p \to a) \to \mu\, f \to p \to a \\ \mathit{accu}\ st\ alg\ (\mathit{In}\ t)\ p = alg\ (\mathit{fmap}\ (\mathit{uncurry}\ (\mathit{accu}\ st\ alg))\ (st\ t\ p))\ p \end{array}$$

Using the recursion scheme, the *relabel* function can be rewritten as

$$\begin{array}{l} \mathit{relabel}' :: \mu\ (\mathit{TreeF\ Integer}) \to \mathit{Integer} \to \mu\ (\mathit{TreeF\ Integer}) \\ \mathit{relabel}' = \mathit{accu}\ st_{relabel}\ alg\ \textbf{where} \\ \quad alg\ \mathit{Empty} \qquad\quad s = \mathit{In\ Empty} \\ \quad alg\ (\mathit{Node}\ l\ \_\ r)\ s = \mathit{In}\ (\mathit{Node}\ l\ s\ r) \end{array}$$

Example 8. In Example 3, the semantics function *interp* is written as a catamorphism into a function type $\mathit{Map\ Int}\ s \to a$. With a closer look, we can see that the $\mathit{Map\ Int}\ s$ parameter is an accumulating parameter, so we can more accurately express *interp* using *accu*:

$$\begin{array}{l} \mathit{interp}' :: \mu\ (\mathit{ProgF}\ s\ a) \to \mathit{Map\ Int}\ s \to a \\ \mathit{interp}' = \mathit{accu}\ st\ alg\ \textbf{where} \\ \quad st :: \mathit{ProgF}\ s\ a\ x \to \mathit{Map\ Int}\ s \to \mathit{ProgF}\ s\ a\ (x, \mathit{Map\ Int}\ s) \\ \quad st\ (\mathit{Ret}\ a) \qquad\quad m = \mathit{Ret}\ a \end{array}$$

[2] The recursion scheme requires the `GHC` extension `RankNTypes` since the first argument involves a polymorphic function.

$$\begin{aligned}
&\mathit{st}\ (\mathit{Put}\ (i,x)\ k)\ m = \mathit{Put}\ (i,x)\ (k, \mathit{update}\ m\ i\ x)\\
&\mathit{st}\ (\mathit{Get}\ i\ k)\qquad m = \mathit{Get}\ i\ (\lambda x \to (k\ x, m))
\end{aligned}$$

$$\begin{aligned}
&\mathit{alg} :: \mathit{ProgF}\ s\ a\ a \to \mathit{Map}\ \mathit{Int}\ s \to a\\
&\mathit{alg}\ (\mathit{Ret}\ a)\quad m = a\\
&\mathit{alg}\ (\mathit{Put}\ _\ k)\ m = k\\
&\mathit{alg}\ (\mathit{Get}\ i\ k)\ m = k\ (m\ !\ i)
\end{aligned}$$

Compared to the previous version *interp* in Example 3, this version *interp′* here singles out *st*, which controls how the memory m is altered by each operation, whereas *alg* shows how each operation continues.

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. A simple example is *isOdd* and *isEven* determining the parity of a natural number:

$$\textbf{data}\ \mathit{NatF}\ a = \mathit{Zero} \mid \mathit{Succ}\ a \qquad\qquad \textbf{type}\ \mathit{Nat} = \mu\ \mathit{NatF}$$

$$\begin{aligned}
&\mathit{isEven} :: \mathit{Nat} \to \mathit{Bool}\\
&\mathit{isEven}\ (\mathit{In}\ \mathit{Zero}) \qquad = \mathit{True}\\
&\mathit{isEven}\ (\mathit{In}\ (\mathit{Succ}\ n)) = \mathit{isOdd}\ n
\end{aligned}
\qquad
\begin{aligned}
&\mathit{isOdd} :: \mathit{Nat} \to \mathit{Bool}\\
&\mathit{isOdd}\ (\mathit{In}\ \mathit{Zero}) \qquad = \mathit{False}\\
&\mathit{isOdd}\ (\mathit{In}\ (\mathit{Succ}\ n)) = \mathit{isEven}\ n
\end{aligned}$$

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.

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

$$\begin{aligned}
&\mathit{fib} :: \mathit{Nat} \to \mathit{Integer}\\
&\mathit{fib}\ (\mathit{In}\ \mathit{Zero}) = 0\\
&\mathit{fib}\ (\mathit{In}\ (\mathit{Succ}\ n)) = \mathit{fib}\ n + \mathit{aux}\ n
\end{aligned}
\qquad
\begin{aligned}
&\mathit{aux} :: \mathit{Nat} \to \mathit{Integer}\\
&\mathit{aux}\ (\mathit{In}\ \mathit{Zero}) = 1\\
&\mathit{aux}\ (\mathit{In}\ (\mathit{Succ}\ n)) = \mathit{fib}\ n
\end{aligned}$$

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

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

and $\mathit{fib}\ n = \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)$ 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. For the example of *fib* and *aux*, the values at *Zero* are uniquely determined for both functions:

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

Then the values at *Succ Zero* are uniquely determined for both functions too, according to their inductive cases: $\langle \mathit{fib}\ 1, \mathit{aux}\ 1\rangle = \langle 1, 0\rangle$, and so on for all inputs:

$$\langle \mathit{fib}\ 2, \mathit{aux}\ 2\rangle = \langle 1, 1\rangle,\ \langle \mathit{fib}\ 3, \mathit{aux}\ 3\rangle = \langle 2, 1\rangle,\ \langle \mathit{fib}\ 4, \mathit{aux}\ 4\rangle = \langle 3, 2\rangle,\ \ldots$$

The same line of reasoning applies too when we generalise this pattern to mutual recursion on a generic inductive datatype.

Recursion Scheme 5. The *mutumorphism* [12] is the recursion scheme for defining two mutually recursive functions on inductive datatypes:

```
mutu :: Functor f ⇒ (f (a, b) → a) → (f (a, b) → b) → (μ f → a, μ f → b)
mutu alg₁ alg₂ = (fst ∘ cata alg, snd ∘ cata alg)
  where alg x = (alg₁ x, alg₂ x)
```

in which alg_1 and alg_2 respectively compute the results of the two functions being defined, from the sub-results of both functions. The name *mutumorphism* is a bit special in the zoo of recursion schemes: the prefix mutu- is from Latin rather than Greek.

For example, using *mutu*, *fib* can be expressed as

```
fib′ = fst (mutu f g) where
  f Zero = 0                      g Zero = 1
  f (Succ (n, m)) = n + m         g (Succ (n, _)) = n
```

In the unifying theory of recursion schemes of conjugate hylomorphisms, a mutumorphism $\mathit{mutu}\ \mathit{alg}_1\ \mathit{alg}_2 :: (\mu\ f \to a, \mu\ f \to b)$ is the left-adjunct of a catamorphism of type $\mu\ f \to (a, b)$ via the adjunction $\Delta \dashv \times$ between the product category $C \times C$ and some base category C [24] (In the setting of this paper, $C = \mathbf{Set}$). The same adjunction also underlies a dual corecursion scheme that we explain below.

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:

$$\mathbf{data}\ \mathit{Expr}\ = \mathit{Add}\ \mathit{Expr}\ \mathit{Term} \mid \mathit{Minus}\ \mathit{Expr}\ \mathit{Term} \mid \mathit{FromT}\ \mathit{Term}$$
$$\mathbf{data}\ \mathit{Term} = \mathit{Lit}\ \mathit{Integer} \mid \mathit{Neg}\ \mathit{Term} \mid \mathit{Paren}\ \mathit{Expr}$$

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:

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

where $\mathit{encLit}\ n = \mathbf{if}\ n \geqslant 0\ \mathbf{then}\ 2 * n + 1\ \mathbf{else}\ 2 * (-n)$ 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:

```
decE :: Integer → Expr
decE n = let (e₂, e₃, e₅, e₁, e₁₁) = factorise11 n
  in if e₂ > 0 ∨ e₃ > 0 then Add (decE e₂) (decT e₃)
    else if e₅ > 0 ∨ e₁ > 0
         then Minus (decE e₅) (decT e₁)
           else FromT (decT e₁₁)
decT :: Integer → Term
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 $\mathit{factorise}11\ n$ 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 can let a recursion scheme do the job for us.

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

For instance, *Expr* is isomorphic to ν_1 *ExprF TermF* and *Term* is isomorphic to ν_2 *ExprF TermF*:

$$\begin{array}{l}\textbf{data}\ \mathit{ExprF}\ e\ t\ = \mathit{Add}'\ e\ t \mid \mathit{Minus}'\ e\ t \mid \mathit{FromT}'\ t\\ \textbf{data}\ \mathit{TermF}\ e\ t = \mathit{Lit}'\ \mathit{Int} \mid \mathit{Neg}'\ t \mid \mathit{Paren}'\ e\end{array}$$

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

$$\begin{array}{l}\mathit{comutu} :: (\mathit{Bifunctor}\ f, \mathit{Bifunctor}\ g) \Rightarrow (c \to f\ c\ c) \to (c \to g\ c\ c)\\ \qquad\quad \to c \to (\nu_1\ f\ g, \nu_2\ f\ g)\\ \mathit{comutu}\ c_1\ c_2\ s = (x\ s, y\ s)\ \textbf{where}\\ \quad x = \mathit{Out}_1^{\circ} \circ \mathit{bimap}\ x\ y \circ c_1\\ \quad y = \mathit{Out}_2^{\circ} \circ \mathit{bimap}\ x\ y \circ c_2\end{array}$$

which remains unnamed in the literature.

Example 9. The *comutu* scheme renders our decoding example to become

$$\begin{array}{l}\mathit{decExprTerm} :: \mathit{Integer} \to (\nu_1\ \mathit{ExprF}\ \mathit{TermF}, \nu_2\ \mathit{ExprF}\ \mathit{TermF})\\ \mathit{decExprTerm} = \mathit{comutu}\ \mathit{genExpr}\ \mathit{genTerm}\\ \mathit{genExpr} :: \mathit{Integer} \to \mathit{ExprF}\ \mathit{Integer}\ \mathit{Integer}\\ \mathit{genExpr}\ n =\\ \quad \textbf{let}\ (e_2, e_3, e_5, e_1, e_{11}) = \mathit{factorise}11\ n\\ \quad \textbf{in if}\ e_2 > 0 \lor e_3 > 0\ \textbf{then}\ \mathit{Add}'\ e_2\ e_3\\ \qquad \textbf{else if}\ e_5 > 0 \lor e_1 > 0\\ \qquad\qquad \textbf{then}\ \mathit{Minus}'\ e_5\ e_1\ \textbf{else}\ \mathit{FromT}'\ e_{11}\\ \mathit{genTerm} :: \mathit{Integer} \to \mathit{TermF}\ \mathit{Integer}\ \mathit{Integer}\\ \mathit{genTerm}\ n =\\ \quad \textbf{let}\ (e_2, e_3, e_5, \_, \_) = \mathit{factorise}11\ n\\ \quad \textbf{in if}\ e_2 > 0\ \textbf{then}\ \mathit{Lit}'\ (\mathit{decLit}\ e_2)\\ \qquad \textbf{else if}\ e_3 > 0\ \textbf{then}\ \mathit{Neg}'\ e_3\ \textbf{else}\ \mathit{Paren}'\ e_5\end{array}$$

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 $\Delta \dashv \times$: $\mathit{comutu}\ c_1\ c_2 :: c \to (\nu_1\ f\ g, \nu_2\ f\ g)$ is the right-adjunct of an anamorphism of type $(c \to \nu_1\ f\ g, c \to \nu_2\ f\ g)$ 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* $c_1\ c_2$, 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 catamorphisms—*paramorphisms*. We also discuss a generalisation, *zygomorphisms*, and the categorical dual *apomorphisms*.

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*:

$$\begin{aligned}
&\mathit{factorial} :: \mathit{Nat} \to \mathit{Nat}\\
&\mathit{factorial}\ (\mathit{In}\ \mathit{Zero}) \quad\;\;\; = 1\\
&\mathit{factorial}\ (\mathit{In}\ (\mathit{Succ}\ n)) = \mathit{In}\ (\mathit{Succ}\ n) * \mathit{factorial}\ n
\end{aligned}$$

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:

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

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* with a structural recursion scheme, we can use mutumorphisms by understanding *factorial* and *wc* as mutually defined with with the identity function. For example,

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

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

Recursion Scheme 7. Structural recursion with access to the original subparts of the input are captured as the following scheme called *paramorphisms* [40]:

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

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

Example 10. With *para*, *factorial* is defined neatly:

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

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:

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

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

$$\mathit{para}\ \mathit{alg} = \mathit{snd} \circ \mathit{cata}\ ((\mathit{In} \circ \mathit{fmap}\ \mathit{fst}) \bigtriangleup \mathit{alg})$$

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 canonical example of non-primitive recursive function, the Ackermann function, is definable with *cata* and thus *para*:

$$
\begin{aligned}
&\mathit{ack} :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}\\
&\mathit{ack} = \mathit{cata}\ \mathit{alg}\ \textbf{where}\\
&\quad \mathit{alg} :: \mathit{NatF}\ (\mathit{Nat} \to \mathit{Nat}) \to (\mathit{Nat} \to \mathit{Nat})\\
&\quad \mathit{alg}\ \mathit{Zero} \qquad\ = \mathit{In} \circ \mathit{Succ}\\
&\quad \mathit{alg}\ (\mathit{Succ}\ a_n) = \mathit{cata}\ \mathit{alg}'\ \textbf{where}\\
&\qquad \mathit{alg}' :: \mathit{NatF}\ \mathit{Nat} \to \mathit{Nat}\\
&\qquad \mathit{alg}'\ \mathit{Zero} = a_n\ (\mathit{In}\ (\mathit{Succ}\ (\mathit{In}\ \mathit{Zero})))\\
&\qquad \mathit{alg}'\ (\mathit{Succ}\ a_{n+1,m}) = a_n\ a_{n+1,m}
\end{aligned}
$$

6.2 Apomorphisms

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

Recursion Scheme 8. We call the following recursion scheme the *apomorphism* [49, 46]. Prefix apo- comes from Greek απο meaning 'apart from'.

$$
\begin{aligned}
&\mathit{apo} :: \mathit{Functor}\ f \Rightarrow (c \to f\ (\mathit{Either}\ (\nu\, f)\ c)) \to c \to \nu\, f\\
&\mathit{apo}\ \mathit{coalg} = \mathit{Out}^{\circ} \circ \mathit{fmap}\ (\mathit{either}\ \mathit{id}\ (\mathit{apo}\ \mathit{coalg})) \circ \mathit{coalg}
\end{aligned}
$$

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 $\nu\ f$ 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 and Uustalu [49] that applies a function f to the first element (if there is) of a coinductive list.

$$maphd :: (a \to a) \to \nu\ (ListF\ a) \to \nu\ (ListF\ a)$$

As an anamorphism, it is expressed as

$$\begin{aligned}
&maphd\ f = ana\ c \circ Left\ \textbf{where} \\
&\quad c\ (Left\ (Out^{\circ}\ Nil)) && = Nil \\
&\quad c\ (Right\ (Out^{\circ}\ Nil)) && = Nil \\
&\quad c\ (Left\ (Out^{\circ}\ (Cons\ x\ xs))) && = Cons\ (f\ x)\ (Right\ xs) \\
&\quad c\ (Right\ (Out^{\circ}\ (Cons\ x\ xs))) && = Cons\ x\ \ \ \ (Right\ xs)
\end{aligned}$$

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:

$$\begin{aligned}
&maphd'\ f = apo\ coalg\ \textbf{where} \\
&\quad coalg\ (Out^{\circ}\ Nil) = Nil \\
&\quad coalg\ (Out^{\circ}\ (Cons\ x\ xs)) = Cons\ (f\ x)\ (Left\ xs)
\end{aligned}$$

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

Example 11. Another instructive example of apomorphisms is inserting a value into an ordered (coinductive) list:

$$\begin{aligned}
&insert :: Ord\ a \Rightarrow a \to \nu\ (ListF\ a) \to \nu\ (ListF\ a) \\
&insert\ y = apo\ c\ \textbf{where} \\
&\quad c\ (Out^{\circ}\ Nil) \ \ = Cons\ y\ (Left\ (Out^{\circ}\ Nil)) \\
&\quad c\ xxs@(Out^{\circ}\ (Cons\ x\ xs)) \\
&\qquad \mid y \leqslant x \qquad = Cons\ y\ (Left\ xxs) \\
&\qquad \mid otherwise = Cons\ x\ (Right\ xs)
\end{aligned}$$

In both cases, an element y or x is emitted, and *Left xxs* makes *xxs* the rest of the output, whereas *Right xs* continues the corecursion to insert y into xs.

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:

$$
\begin{aligned}
&\mathit{perfect} :: \mu\ (\mathit{TreeF}\ e) &&\to \mathit{Bool}\\
&\mathit{perfect}\ (\mathit{In}\ \mathit{Empty}) &&= \mathit{True}\\
&\mathit{perfect}\ (\mathit{In}\ (\mathit{Node}\ l\ \_\ r)) &&= \mathit{perfect}\ l \wedge \mathit{perfect}\ r \wedge (\mathit{depth}\ l \equiv \mathit{depth}\ r)\\
&\mathit{depth} :: \mu\ (\mathit{TreeF}\ e) &&\to \mathit{Integer}\\
&\mathit{depth}\ (\mathit{In}\ \mathit{Empty}) &&= 0\\
&\mathit{depth}\ (\mathit{In}\ (\mathit{Node}\ l\ \_\ r)) &&= 1 + \mathit{max}\ (\mathit{depth}\ l)\ (\mathit{depth}\ r)
\end{aligned}
$$

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:

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

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.

Recursion Scheme 9. Structural recursion with auxiliary information is called *zygomorphisms* [37]:

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

in which alg_1 computes the function of interest from the recursive results together with auxiliary information of type b, and alg_2 maintains the auxiliary information. Malcolm called zygomorphisms ‘yoking together of paramorphisms and catamorphisms’ and prefix ‘zygo-’ is from Greek ζυγόν meaning ‘yoke’.

Example 12. As we said, *zygo* is a generalisation of paramorphisms: $\mathit{para}\ \mathit{alg} = \mathit{zygo}\ \mathit{alg}\ \mathit{In}$. And the above *perfect* is $\mathit{zygo}\ p\ d$ where

$$
\begin{aligned}
&p :: \mathit{TreeF}\ e\ (\mathit{Bool}, \mathit{Integer}) \to \mathit{Bool}\\
&p\ \mathit{Empty} &&= \mathit{True}\\
&p\ (\mathit{Node}\ (p_l, d_l)\ \_\ (p_r, d_r)) &&= p_l \wedge p_r \wedge (d_l \equiv d_r)\\
&d :: \mathit{TreeF}\ e\ \mathit{Integer} \to \mathit{Integer}\\
&d\ \mathit{Empty} &&= 0\\
&d\ (\mathit{Node}\ d_l\ \_\ d_r) &&= 1 + (\mathit{max}\ d_l\ d_r)
\end{aligned}
$$

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 [24]. 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 scheme for dynamic programming, *histomorphisms*, and a generalisation called *dynamorphisms*, 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 $[1, 2, 3, 9]$.

A way to find LIS follows the observation that an LIS of $x : xs$ is either an LIS of xs or a subsequence beginning with the head element 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 longer). This idea is implemented by the program below.

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

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

Unfortunately this implementation is very inefficient because 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 lis' on substructures: when computing $\mathit{lis}'\ (xs +\!\!+\ ys)$, for each x in xs, $\mathit{lis}'\ ys$ is recomputed although the results are identical. A technique to speed up the algorithm is to memoise the results of 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

$$\begin{aligned}&\textbf{data}\ \mathit{Cofree}\ f\ a\ \textbf{where}\\ &\quad (\triangleleft) :: a \to f\ (\mathit{Cofree}\ f\ a) \to \mathit{Cofree}\ f\ a\end{aligned}$$

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

$$\begin{aligned}&\mathit{extract} :: \mathit{Cofree}\ f\ a \to a\\ &\mathit{extract}\ (x \triangleleft \_) = x\end{aligned}$$

Recursion Scheme 10. The recursion scheme *histomorphism* [46] is:

$$\begin{aligned}&\mathit{histo} :: \mathit{Functor}\ f \Rightarrow (f\ (\mathit{Cofree}\ f\ a) \to a) \to \mu\, f \to a\\ &\mathit{histo}\ \mathit{alg} = \mathit{extract} \circ \mathit{cata}\ (\lambda x \to \mathit{alg}\ x \triangleleft x)\end{aligned}$$

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 13. The dynamic programming implementation of *lis* is then:

$$\begin{aligned}&\mathit{lis}'' :: \mathit{Ord}\ a \Rightarrow \mu\ (\mathit{ListF}\ a) \to \mathit{Integer}\\ &\mathit{lis}'' = \mathit{snd} \circ \mathit{histo}\ \mathit{alg}\\ &\mathit{alg} :: \mathit{Ord}\ a \Rightarrow \mathit{ListF}\ a\ (\mathit{Cofree}\ (\mathit{ListF}\ a)\ (\mathit{Integer}, \mathit{Integer}))\\ &\qquad\qquad \to (\mathit{Integer}, \mathit{Integer})\\ &\mathit{alg}\ \mathit{Nil} \qquad\qquad\quad = (0, 0)\\ &\mathit{alg}\ (\mathit{Cons}\ x\ \mathit{table}) = (a, b)\ \textbf{where}\\ &\quad a = 1 + \mathit{findNext}\ x\ \mathit{table}\\ &\quad b = \mathit{max}\ a\ (\mathit{snd}\ (\mathit{extract}\ \mathit{table}))\end{aligned}$$

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

$$\begin{aligned}&\mathit{findNext} :: \mathit{Ord}\ a \Rightarrow a \to \mathit{Cofree}\ (\mathit{ListF}\ a)\ (\mathit{Integer}, \mathit{Integer}) \to \mathit{Integer}\\ &\mathit{findNext}\ x\ ((a, \_) \triangleleft \mathit{Nil}) \qquad\qquad\quad = a\\ &\mathit{findNext}\ x\ ((a, \_) \triangleleft (\mathit{Cons}\ y\ \mathit{table}')) = \textbf{if}\ x < y\ \textbf{then}\ \mathit{max}\ a\ b\ \textbf{else}\ b\\ &\quad \textbf{where}\ b = \mathit{findNext}\ x\ \mathit{table}'\end{aligned}$$

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 $U \dashv \mathit{Cofree}_F$ [23] where 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 [26].

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 [2].

$$
\begin{array}{l}
\mathit{lcs} :: \mathit{Eq}\ a \Rightarrow [\,a\,] \to [\,a\,] \to \mathit{Integer} \\
\mathit{lcs}\ [\,]\ \_ = 0 \\
\mathit{lcs}\ \_\ [\,] = 0 \\
\mathit{lcs}\ \mathit{xxs}@(x : \mathit{xs})\ \mathit{yys}@(y : \mathit{ys}) \\
\quad \mid x \equiv y \qquad\quad = \mathit{lcs}\ \mathit{xs}\ \mathit{ys} + 1 \\
\quad \mid \mathit{otherwise} = \mathit{max}\ (\mathit{lcs}\ \mathit{xs}\ \mathit{yys})\ (\mathit{lcs}\ \mathit{xxs}\ \mathit{ys})
\end{array}
$$

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 in° with a user-supplied recursive coalgebra:

Recursion Scheme 11. The *dynamorphism* (evidently the name is derived from *dyna*mic programming) introduced by Kabanov and Vene [34] is:

$$
\begin{array}{l}
\mathit{dyna} :: \mathit{Functor}\ f \Rightarrow (f\ (\mathit{Cofree}\ f\ a) \to a) \to (c \to f\ c) \to c \to a \\
\mathit{dyna}\ \mathit{alg}\ \mathit{coalg} = \mathit{extract} \circ \mathit{hylo}\ (\lambda x \to \mathit{alg}\ x \lhd x)\ \mathit{coalg}
\end{array}
$$

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

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

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

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

where $\mathit{index}\ t\ n$ extracts the n-th entry of the memo-table:

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

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 $(\mathit{drop}\ n\ x, \mathit{drop}\ m\ y)$ in the memo-table when computing (x, y) is:

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

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

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

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 (n, x) of type (Int, a) and $n > 0$ for all elements. The output is a list $[a]$ and each (n, x) in the input is interpreted as n consecutive copies of x. As an anamorphism, it is expressed as

$$
\begin{aligned}
&\mathit{rld} :: [(\mathit{Int}, a)] \to \nu\ (\mathit{ListF}\ a)\\
&\mathit{rld} = \mathit{ana}\ c\ \textbf{where}\\
&\quad c\ [\,] = \mathit{Nil}\\
&\quad c\ ((n, x) : \mathit{xs})\\
&\qquad \mid n \equiv 1 \qquad\ \ = \mathit{Cons}\ x\ \mathit{xs}\\
&\qquad \mid \mathit{otherwise} = \mathit{Cons}\ x\ ((n-1, x) : \mathit{xs})
\end{aligned}
$$

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 a structure given by a functor f are represented by the *Free* monad:

$$\textbf{data}\ \mathit{Free}\ f\ a = \mathit{Ret}\ a \mid \mathit{Op}\ (f\ (\mathit{Free}\ f\ a))$$

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

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Functor}\ f \Rightarrow (f\ b \to b) \to (a \to b) \to \mathit{Free}\ f\ a \to b\\
&\mathit{eval}\ \mathit{alg}\ g\ (\mathit{Ret}\ a) = g\ a\\
&\mathit{eval}\ \mathit{alg}\ g\ (\mathit{Op}\ k)\ \ = \mathit{alg}\ (\mathit{fmap}\ (\mathit{eval}\ \mathit{alg}\ g)\ k)
\end{aligned}
$$

Recursion Scheme 12. With these constructions, we define the recursion scheme *futumorphisms* [46]:

$$
\begin{aligned}
&\mathit{futu} :: \mathit{Functor}\ f \Rightarrow (c \to f\ (\mathit{Free}\ f\ c)) \to c \to \nu\, f\\
&\mathit{futu}\ \mathit{coalg} = \mathit{ana}\ \mathit{coalg}' \circ \mathit{Ret}\ \textbf{where}\\
&\quad \mathit{coalg}'\ (\mathit{Ret}\ a) = \mathit{coalg}\ a\\
&\quad \mathit{coalg}'\ (\mathit{Op}\ k)\ \ = k
\end{aligned}
$$

Example 15. We can redefine *rld* as a futumorphism:

$$
\begin{aligned}
&\mathit{rld}' :: [(\mathit{Int}, a)] \to \nu\ (\mathit{ListF}\ a)\\
&\mathit{rld}' = \mathit{futu}\ \mathit{dec}\\
&\mathit{dec}\ [\,] \qquad\qquad\ = \mathit{Nil}\\
&\mathit{dec}\ ((n, c) : \mathit{xs}) = \textbf{let}\ (\mathit{Op}\ g) = \mathit{rep}\ n\ \textbf{in}\ g\ \textbf{where}\\
&\quad \mathit{rep}\ 0\ \ = \mathit{Ret}\ \mathit{xs}\\
&\quad \mathit{rep}\ m = \mathit{Op}\ (\mathit{Cons}\ c\ (\mathit{rep}\ (m-1)))
\end{aligned}
$$

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 $(\nu\ F, Out^{\circ})$ 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:

$$
\begin{aligned}
&chrono :: Functor\ f \Rightarrow (f\ (Cofree\ f\ b) \to b) \\
&\qquad\qquad\qquad\quad \to (a \to f\ (Free\ f\ a)) \\
&\qquad\qquad\qquad\quad \to a \to b \\
&chrono\ alg\ coalg = extract \circ hylo\ alg'\ coalg' \circ Ret\ \mathbf{where} \\
&\quad alg'\ x = alg\ x \lhd x \\
&\quad coalg'\ (Ret\ a) = coalg\ a \\
&\quad coalg'\ (Op\ k)\ = k
\end{aligned}
$$

which is dubbed *chronomorphisms* by Kmett [35] (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:

$$
\begin{aligned}
&printTree :: Show\ a \Rightarrow \mu\ (TreeF\ a) \to IO\ () \\
&printTree\ (In\ Empty) \qquad\quad\ = return\ () \\
&printTree\ (In\ (Node\ l\ a\ r)) = \mathbf{do}\ printTree\ l; printTree\ r; print\ a
\end{aligned}
$$

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

$$
\begin{aligned}
&printTree' :: Show\ a \Rightarrow \mu\ (TreeF\ a) \to IO\ () \\
&printTree' = cata\ printAlg\ \mathbf{where} \\
&\quad printAlg :: Show\ a \Rightarrow TreeF\ a\ (IO\ ()) \to IO\ () \\
&\quad printAlg\ Empty \qquad\quad\ = return\ () \\
&\quad printAlg\ (Node\ ml\ a\ mr) = \mathbf{do}\ ml; mr; print\ a
\end{aligned}
$$

Thus a straightforward way of abstracting ‘monadic catamorphisms’ is to restrict *cata* to monadic values.

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

$$\begin{aligned}&\mathit{cataM} :: (\mathit{Functor}\ f, \mathit{Monad}\ m) \Rightarrow (f\ (m\ a) \to m\ a) \to \mu\, f \to m\ a\\&\mathit{cataM}\ \mathit{algM} = \mathit{cata}\ \mathit{algM}\end{aligned}$$

which is the second approach to monadic catamorphisms in [45].

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

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

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:

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

Example 16. Binary trees *TreeF* can be sequenced from left to right:

$$\begin{aligned}&\mathit{lToR} :: \mathit{Monad}\ m \Rightarrow \mathit{TreeF}\ a\ (m\ x) \to m\ (\mathit{TreeF}\ a\ x)\\&\mathit{lToR}\ \mathit{Empty} \qquad\qquad\quad = \mathit{return}\ \mathit{Empty}\\&\mathit{lToR}\ (\mathit{Node}\ \mathit{ml}\ a\ \mathit{mr}) = \mathbf{do}\ l \leftarrow \mathit{ml}; r \leftarrow \mathit{mr}; \mathit{return}\ (\mathit{Node}\ l\ a\ r)\end{aligned}$$

and also from right to left:

$$\begin{aligned}&\mathit{rToL} :: \mathit{Monad}\ m \Rightarrow \mathit{TreeF}\ a\ (m\ x) \to m\ (\mathit{TreeF}\ a\ x)\\&\mathit{rToL}\ \mathit{Empty} \qquad\qquad\quad = \mathit{return}\ \mathit{Empty}\\&\mathit{rToL}\ (\mathit{Node}\ \mathit{ml}\ a\ \mathit{mr}) = \mathbf{do}\ r \leftarrow \mathit{mr}; l \leftarrow \mathit{ml}; \mathit{return}\ (\mathit{Node}\ l\ a\ r)\end{aligned}$$

Recursion Scheme 14 (*mcata*). The *monadic catamorphism* [13, 45] is the following recursion scheme:

$$\begin{aligned}&\mathit{mcata} :: (\mathit{Monad}\ m, \mathit{Functor}\ f) \Rightarrow (\forall x. f\ (m\ x) \to m\ (f\ x))\\&\qquad \to (f\ a \to m\ a) \to \mu\, f \to m\ a\\&\mathit{mcata}\ \mathit{seq}\ \mathit{alg} = \mathit{cata}\ ((\gg\!= \mathit{alg}) \circ \mathit{seq})\end{aligned}$$

Example 17. The program *printTree* above is a monadic catamorphism:

$$\begin{aligned}&\mathit{printTree}'' :: \mathit{Show}\ a \Rightarrow \mu\ (\mathit{TreeF}\ a) \to \mathit{IO}\ ()\\&\mathit{printTree}'' = \mathit{mcata}\ \mathit{lToR}\ \mathit{printElem}\ \mathbf{where}\\&\quad \mathit{printElem}\ \mathit{Empty} \qquad\quad = \mathit{return}\ ()\\&\quad \mathit{printElem}\ (\mathit{Node}\ \_\ a\ \_) = \mathit{print}\ a\end{aligned}$$

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

Distributive Conditions In the literature [13, 45, 24], the sequencing of a monadic catamorphism is required to be a *distributive law* of functor f over monad m, which means that $seq :: \forall x. f\ (m\ x) \rightarrow m\ (f\ x)$ satisfies two conditions:

$$seq \circ fmap\ return = return \tag{7}$$

$$seq \circ fmap\ join = join \circ fmap\ seq \circ seq \tag{8}$$

Intuitively, condition (7) prohibits *seq* from inserting additional computational effects when combining computations for substructures, which is a reasonable requirement. Condition (8) 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 nicer calculational properties.

Unfortunately, condition (8) is usually too strong in practice. For example, neither *lToR* nor *rToL* in Example 16 satisfies condition (8) when m is the *IO* monad. To see this, let

$$\begin{aligned} c = Node\ &(putStr\ \texttt{"A"} \gg return\ (putStr\ \texttt{"C"})) \\ &(putStr\ \texttt{"B"} \gg return\ (putStr\ \texttt{"D"})) \end{aligned}$$

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, excluding 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* [43, 45]:

$$\begin{aligned} &mhylo :: (Monad\ m, Functor\ f) \Rightarrow (\forall x. f\ (m\ x) \rightarrow m\ (f\ x)) \\ &\qquad \rightarrow (f\ a \rightarrow m\ a) \rightarrow (c \rightarrow m\ (f\ c)) \rightarrow c \rightarrow m\ a \\ &mhylo\ seq\ alg\ coalg\ c = \mathbf{do}\ x \leftarrow coalg\ c \\ &\qquad\qquad\qquad\qquad\qquad\quad y \leftarrow seq\ (fmap\ (mhylo\ seq\ alg\ coalg)\ x) \\ &\qquad\qquad\qquad\qquad\qquad\quad alg\ y \end{aligned}$$

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

$$\begin{aligned} &mana :: (Monad\ m, Functor\ f) \Rightarrow (\forall x. f\ (m\ x) \rightarrow m\ (f\ x)) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow (c \rightarrow m\ (f\ c)) \rightarrow c \rightarrow m\ (\nu\ f) \\ &mana\ seq\ coalg = mhylo\ seq\ (return \circ Out^{\circ})\ coalg \end{aligned}$$

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

Example 18. Generating a random tree of some depth with $randomIO :: IO\ Int$ is a monadic anamorphism:

$$
\begin{aligned}
&ranTree :: Integer \to IO\ (\nu\ (TreeF\ Int))\\
&ranTree = mana\ lToR\ gen\ \textbf{where}\\
&\quad gen :: Integer \to IO\ (TreeF\ Int\ Integer)\\
&\quad gen\ 0 = return\ Empty\\
&\quad gen\ n = \textbf{do}\ a \leftarrow randomIO :: IO\ Int\\
&\qquad\qquad\qquad return\ (Node\ (n-1)\ a\ (n-1))
\end{aligned}
$$

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). An example of algebraic datatypes that is not (co)inductive is the datatype for purely functional *random-access lists* [42]:

$$\textbf{data}\ RList\ a = Null \mid Zero\ (RList\ (a,a)) \mid One\ a\ (RList\ (a,a))$$

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 [7, 31, 33].

Nested datatypes are covered by a broader range of datatypes called *generalised algebraic datatypes* (GADTs) [32, 20]. 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 an defining equation $\textbf{data}\ D\ P = ...$ to be more complex than type variables. GADTs have a different syntax from that of ADTs in Haskell[3]. For example, as a GADT, *RList* is

$$
\begin{aligned}
&\textbf{data}\ RList :: * \to * \textbf{where}\\
&\quad Null :: RList\ a\\
&\quad Zero :: RList\ (a,a) \to RList\ a\\
&\quad One :: a \to RList\ (a,a) \to RList\ a
\end{aligned}
$$

in which each constructor is directly declared with a type signature. With 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: first we define two datatypes $\textbf{data}\ Z'$ and $\textbf{data}\ S'\ n$ with no constructors, then the GADT for vectors is

[3] Support of GADTs is turned on by the extension `GADTs` in `GHC`.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Vec}\ (a :: *) :: * \to *\ \textbf{where}\\
&\quad \mathit{Nil}\ \ \ :: \mathit{Vec}\ a\ Z'\\
&\quad \mathit{Cons} :: a \to \mathit{Vec}\ a\ n \to \mathit{Vec}\ a\ (S'\ n)
\end{aligned}
$$

in which types Z' and $S'\ n$ encode natural numbers at the type level, thus it does not matter what their term constructors are.

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.

$$
\begin{aligned}
&\mathit{safeHead} :: \mathit{Vec}\ a\ (S'\ n) \to a\\
&\mathit{safeHead}\ (\mathit{Cons}\ a\ \_) \quad = a
\end{aligned}
$$

GADTs as Fixed Points As we mentioned earlier, nested datatypes and GADTs cannot be modelled as fixed points of Haskell functors in general, making them out of the reach of the recursion schemes that we have seen so far. However, there are other ways to view them as fixed points. Let us look at the *RList* datatype again,

$$\textbf{data}\ \mathit{RList}\ a = \mathit{Null} \mid \mathit{Zero}\ (\mathit{RList}\ (a, a)) \mid \mathit{One}\ a\ (\mathit{RList}\ (a, a))$$

instead of viewing it as defining a type $\mathit{RList}\ a :: *$, we can alternatively understand it as defining a functor $\mathit{RList} :: * \to *$, where $*$ is the category of Haskell types, such that *RList* satisfies the fixed point equation $\mathit{RList} \cong \mathit{RListF}\ \mathit{RList}$ for a *higher-order* functor $\mathit{RListF} :: (* \to *) \to (* \to *)$ defined as

$$\textbf{data}\ \mathit{RListF}\ f\ a = \mathit{NullF} \mid \mathit{ZeroF}\ (f\ (a, a)) \mid \mathit{OneF}\ a\ (f\ (a, a))$$

In this way, nested datatypes are still fixed points, but of higher-order functors, rather than usual Haskell functors [7, 31].

This idea applies to GADTs as well, but with a caveat: consider the GADT G defined as follows:

$$
\begin{aligned}
&\textbf{data}\ G\ a\ \textbf{where}\\
&\quad \mathit{Leaf}\ :: a \to G\ a\\
&\quad \mathit{Prod} :: G\ a \to G\ b \to G\ (a, b)
\end{aligned}
$$

then G *cannot* be a functor at all, let alone a fixed point of some higher-order functor. The problem is defining *fmap* for the *Prod* constructor:

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

but we have no way to construct a $G\ c$ given $f :: (a, b) \to c$, $ga :: G\ a$ and $gb :: G\ b$. Luckily, Johann and Ghani [32] shows how to fix this problem. In fact, all we need to do is to give up the expectation that a GADT $G :: * \to *$ is functorial in its domain. In categorical terminology, we view GADTs as functors from the *discrete category* $|*|$ of Haskell types to the category $*$ of Haskell types, rather than functors from $*$ to $*$. In other words, a GADT $G :: * \rightarrow *$ is then merely a type constructor in Haskell, without necessarily a *Functor* instance. A *natural transformation* between two functors a and b from $|*|$ to $*$ is a polymorphic function $\forall i.a\ i \rightarrow b\ i$, which we give a type synonym $a \dot{\rightarrow} b$[4]:

$$\mathbf{type}\ (\dot{\rightarrow})\ a\ b = \forall i.a\ i \rightarrow b\ i$$

And a higher-order endofunctor (on the functor category $*^{|*|}$) is f instantiating the following type class, which is analogous to the *Functor* type class of Haskell:

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

in which *fmap*'s counterpart *hfmap* maps a natural transformation $a \dot{\rightarrow} b$ to another natural transformation $f\ a \dot{\rightarrow} f\ b$. 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 19. Fixed-length vectors *Vec e* are isomorphic to $\dot{\mu}\ (\mathit{VecF}\ e)$ 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)
```

Recursion Scheme 15 **(*icata*).** With the machinery above, we can devise the structural recursion scheme for $\dot{\mu}$, which we call *indexed catamorphisms*:

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

Example 20. Just like list processing functions such as *map* are catamorphisms, their counterparts for vectors can also be written as indexed catamorphisms:

```
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)
```

[4] It requires the `RankNTypes` extension of `GHC`.

Example 21. Terms of untyped lambda calculus with de Bruijn indices can be modelled as the fixed point of the following higher-order functor [8]:

$$\begin{aligned}&\mathbf{data}\ \mathit{LambdaF} :: (* \to *) \to (* \to *)\ \mathbf{where}\\ &\quad \mathit{Var} :: a \to \mathit{LambdaF}\ f\ a\\ &\quad \mathit{App} :: f\ a \to f\ a \to \mathit{LambdaF}\ f\ a\\ &\quad \mathit{Abs} :: f\ (\mathit{Maybe}\ a) \to \mathit{LambdaF}\ f\ a\end{aligned}$$

Letting a be some type, inhabitants of $\dot{\mu}\ \mathit{LambdaF}\ a$ are precisely the lambda terms in which free variables range over a. Thus $\dot{\mu}\ \mathit{LambdaF}\ \mathit{Void}$ 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 $\dot{\mu}\ \mathit{LambdaF} \mathrel{\dot{\to}} a$ for some $a :: * \to *$. If we want to compute just an integer, we need to wrap it in a constant functor:

$$\mathbf{newtype}\ K\ a\ x = K\ \{\mathit{unwrap} :: a\}$$

Computing the size of a term is done by

$$\begin{aligned}&\mathit{size} :: \dot{\mu}\ \mathit{LambdaF} \mathrel{\dot{\to}} K\ \mathit{Integer}\\ &\mathit{size} = \mathit{icata}\ \mathit{alg}\ \mathbf{where}\\ &\quad \mathit{alg} :: \mathit{LambdaF}\ (K\ \mathit{Integer}) \mathrel{\dot{\to}} K\ \mathit{Integer}\\ &\quad \mathit{alg}\ (\mathit{Var}\ \_) \qquad\qquad\quad\ = K\ 1\\ &\quad \mathit{alg}\ (\mathit{App}\ (K\ n)\ (K\ m)) = K\ (n + m + 1)\\ &\quad \mathit{alg}\ (\mathit{Abs}\ (K\ n)) \qquad\quad\ = K\ (n + 1)\end{aligned}$$

Example 22. 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.

$$\begin{aligned}&\mathit{sumRList} :: \mathit{RList}\ \mathit{Integer} \to \mathit{Integer}\\ &\mathit{sumRList}\ \mathit{Null} \qquad\quad = 0\\ &\mathit{sumRList}\ (\mathit{Zero}\ \mathit{xs}) \ \ = \mathit{sumRList}\ (\mathit{fmap}\ (\mathit{uncurry}\ (+))\ \mathit{xs})\\ &\mathit{sumRList}\ (\mathit{One}\ x\ \mathit{xs}) = x + \mathit{sumRList}\ (\mathit{fmap}\ (\mathit{uncurry}\ (+))\ \mathit{xs})\end{aligned}$$

Does it fit into an indexed catamorphism from $\dot{\mu}\ \mathit{RListF}$? The answer is yes, with the clever choice of the continuation monad *Cont Integer* a as the result type of *icata*.

$$\begin{aligned}&\mathbf{newtype}\ \mathit{Cont}\ r\ a = \mathit{Cont}\ \{\mathit{runCont} :: (a \to r) \to r\}\\ &\mathit{sumRList}' :: \dot{\mu}\ \mathit{RListF}\ \mathit{Integer} \to \mathit{Integer}\\ &\mathit{sumRList}'\ x = \mathit{runCont}\ (h\ x)\ \mathit{id}\ \mathbf{where}\\ &\quad h :: \dot{\mu}\ \mathit{RListF} \mathrel{\dot{\to}} \mathit{Cont}\ \mathit{Integer}\\ &\quad h = \mathit{icata}\ \mathit{sum}\ \mathbf{where}\end{aligned}$$

$$
\begin{aligned}
&\mathit{sum} :: \mathit{RListF}\ (\mathit{Cont\ Integer}) \mathrel{\dot{\to}} \mathit{Cont\ Integer}\\
&\mathit{sum\ NullF} \qquad\quad = \mathit{Cont}\ (\lambda k \to 0)\\
&\mathit{sum}\ (\mathit{ZeroF}\ s) \quad = \mathit{Cont}\ (\lambda k \to \mathit{runCont}\ s\ (\mathit{fork}\ k))\\
&\mathit{sum}\ (\mathit{OneF}\ a\ s) = \mathit{Cont}\ (\lambda k \to k\ a + \mathit{runCont}\ s\ (\mathit{fork}\ k))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{fork} :: (y \to \mathit{Integer}) \to (y, y) \to \mathit{Integer}\\
&\mathit{fork}\ k\ (a, b) = k\ a + k\ b
\end{aligned}
$$

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* [6, 1]. Later, Johann and Ghani [31] 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 Equational Reasoning with Recursion Schemes

We have talked about a handful of recursion schemes, which are recognised common patterns in 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 and de Moor [4] for a comprehensive introduction to this subject and Bird [3] 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,

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

which directly follows the definition of a recursive coalgebra. Instantiating x to $\mathit{hylo}\ a\ c$, we get the defining equation of *hylo*

$$\mathit{hylo}\ a\ c = a \circ \mathit{fmap}\ (\mathit{hylo}\ a\ c) \circ c \qquad (\textsc{HyloComp})$$

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

$$\mathit{id} = a \circ c \iff \mathit{id} = \mathit{hylo}\ a\ c \qquad (\textsc{HyloRefl})$$

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

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. $\mathit{hylo}\ a\ c$, 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):

$$\begin{array}{ccc} ta & \xleftarrow{\ x\ } & tc \\ {\scriptstyle a}\uparrow & & \downarrow{\scriptstyle c} \\ f\ ta & \xleftarrow[\mathit{fmap}\ x]{} & f\ tc \end{array}$$

If we put another *commuting* square beside it,

$$\begin{array}{ccccc} tb & \xleftarrow{\ h\ } & ta & \xleftarrow{\ x\ } & tc \\ {\scriptstyle b}\uparrow & & {\scriptstyle a}\uparrow & & \downarrow{\scriptstyle c} \\ f\ tb & \xleftarrow[\mathit{fmap}\ h]{} & f\ ta & \xleftarrow[\mathit{fmap}\ x]{} & f\ tc \end{array} \qquad (9)$$

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 $\mathit{hylo}\ c\ b$ is the only arrow making the outer rectangle commute, thus $\mathit{hylo}\ c\ b = h \circ x = h \circ \mathit{hylo}\ a\ c$. In summary, the fusion law is:

$$h \circ \mathit{hylo}\ a\ c = \mathit{hylo}\ b\ c \iff h \circ a = b \circ \mathit{fmap}\ h \qquad (\text{HYLOFUSION})$$

and its dual version for corecursive algebra a is

$$\mathit{hylo}\ a\ c \circ h = \mathit{hylo}\ a\ d \iff c \circ h = \mathit{fmap}\ h \circ d \qquad (\text{HYLOFUSIONCO})$$

where $d :: td \to f\ td$. Fusion laws combine a funciton after or before a hylomorphism into one hylomorphism, and thus it is widely used for optimisation [10].

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

Example 23. Suppose some $f :: \mathit{Integer} \to \mathit{Integer}$ such that for all $a, b :: \mathit{Integer}$,

$$f\ (a + b) = f\ a + f\ b \quad \wedge \quad f\ 0 = 0 \qquad (10)$$

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 $\mathit{sum} \circ \mathit{map}\ f = f \circ \mathit{sum}$ with the properties of *hylo*.

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

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

for some g. The correct choice of g is

$$\begin{aligned} &g :: ListF\ Integer \to Integer \\ &g\ Nil \qquad\quad\ = f\ 0 \\ &g\ (Cons\ x\ y) = f\ x + y \end{aligned}$$

First, $sum \circ map\ f = sum \circ hylo\ app\ in^{\circ}$, and by HyloFusion,

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

is implied by

$$sum \circ app = g \circ fmap\ sum \tag{11}$$

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

$$(plus \circ fmap\ sum \circ in^{\circ}) \circ app = g \circ fmap\ sum \tag{12}$$

which is an equation of functions

$$ListF\ Integer\ (\mu\ (ListF\ Integer)) \to Integer$$

and it can be shown by a case analysis on the input. For Nil, the left-hand side of (12) equals to

$$\begin{aligned} &\ \ plus\ (fmap\ sum\ (in^{\circ}\ (app\ Nil))) \\ &= plus\ (fmap\ sum\ (in^{\circ}\ (In\ Nil))) \\ &= plus\ (fmap\ sum\ Nil) \\ &= plus\ Nil \qquad\qquad \text{(by definition of } fmap \text{ for } ListF) \\ &= 0 \end{aligned}$$

and the right-hand side of (12) equals to

$$g\ (fmap\ sum\ Nil) = g\ Nil = g\ 0 = f\ 0$$

and by assumption (10) about f, $f\ 0 = 0$. Similarly when the input is $Cons\ a\ b$, we can calculate that both sides equal to $f\ a + sum\ b$. Thus we have shown (11), and therefore $sum \circ hylo\ app\ in^{\circ} = hylo\ g\ in^{\circ}$.

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

$$f \circ plus = g \circ fmap\ f$$

which can be verified by case analysis on the input: When the input is Nil, both sides equal to $f\ 0$. When the input is $Cons\ a\ b$, the left-hand side equals to $f\ (a + b)$ and the right-hand side is $f\ a + f\ b$. By assumption (10) on f, $f\ (a + b) = f\ a + f\ b$.

11 Closing Remarks and Further Reading

We have shown a handful of structural recursion schemes and their applications by examples. We hope that this paper can be an accessible introduction to this subject and a quick reference when functional programmers hear about some *morphism* with an obscure Greek prefix. We end this paper with some remarks on general approaches to find more *fantastic morphisms* and some pointers to further reading about the theory and applications of recursion schemes.

From Categories and Adjunctions As we have seen, recursion schemes live with categories and adjunctions, so whenever we see a new category, it is a good idea to think about catamorphisms and anamorphisms in this category, as we did for the Kleisli category, where we obtained *mcata*, and the functor category $*^{|*|}$, where we obtained *icata*, etc. Also, whenever we encounter an adjunction $L \dashv R$, we can think about if functions of type $L\ c \to a$, especially $L\ (\mu\ f) \to a$, are anything interesting. If they are, there might be interesting conjugate hylomorphisms from this adjunction.

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 probably want to combine the functionalities of recursion schemes. For example, if we want to define two mutually recursive functions with historical information, we need a recursion scheme of type

$$\begin{aligned} \mathit{mutuHist} :: \mathit{Functor}\ f \Rightarrow {} & (f\ (\mathit{Cofree}\ f\ (a, b)) \to a) \\ & \to (f\ (\mathit{Cofree}\ f\ (a, b)) \to b) \to (\mu\ f \to a, \mu\ f \to b) \end{aligned}$$

Theoretically, *mutuHist* is the composite of *mutu* and *accu* in the sense that the adjunction $U \dashv \mathit{Cofree}_F$ underlying *hist* and the adjunction $\Delta \dashv \times$ underlying *mutu* can be composed to an adjunction inducing *mutuHist* [22]. Unfortunately, our Haskell implementations of *mutu* and *hist* are not composable. A composable library of recursion schemes in Haskell would require considerable machinery for doing category theory in Haskell, and how to do it with good usability is a question worth exploring.

Further Reading The examples in this paper are fairly small ones, but recursion schemes are surely useful in real-world programs and algorithms. For the reader who wants to see recursion schemes in real-world algorithms, we recommend books by Bird [3] and Bird and Gibbons [5]. Their books provide a great deal of examples of proving correctness of algorithms using properties of recursion schemes, which we only briefly showcased in Section 10.

We have only glossed over the category theory of the unifying theories of recursion schemes. For the reader interested in them, a good place to start is Hinze [21]'s lecture notes on adjoint folds and unfolds, and then Uustalu et al.

[48]'s paper on recursion schemes from comonads, which are less general than adjoint folds, but they have generic implementations in Haskell [36]. Finally, Hinze et al. [26]'s conjugate hylomorphisms are the most general framework of recursion schemes so far, although they do not have an implementation yet.

Bibliography

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

[2] Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, pp. 39–48 (Sep 2000), `https://doi.org/10.1109/SPIRE.2000.878178`

[3] Bird, R.: Pearls of functional algorithm design. Cambridge University Press (2010)

[4] Bird, R., de Moor, O.: Algebra of Programming. London (1997), ISBN 0-13-507245-X

[5] Bird, R., Gibbons, J.: Algorithm Design with Haskell. Cambridge University Press (July 2020), ISBN 9781108491617, URL `http://www.cs.ox.ac.uk/publications/books/adwh/`

[6] Bird, R., Paterson, R.: Generalised folds for nested datatypes. Formal Aspects of Computing **11**(2), 200–222 (1999), ISSN 0934-5043, `https://doi.org/10.1007/s001650050047`

[7] Bird, R.S., Meertens, L.G.L.T.: Nested datatypes. In: Proceedings of the Mathematics of Program Construction, p. 52–67, MPC '98, Springer-Verlag, Berlin, Heidelberg (1998), ISBN 3540645918

[8] Bird, R.S., Paterson, R.: de bruijn notation as a nested datatype. Journal of Functional Programming **9**(1), 77–91 (1999), `https://doi.org/10.1017/S0956796899003366`

[9] Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation **204**(4), 437–468 (2006), ISSN 0890-5401, `https://doi.org/10.1016/j.ic.2005.08.005`

[10] Coutts, D., Leshchinskiy, R., Stewart, D.: Stream fusion: From lists to streams to nothing at all. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, p. 315–326, ICFP '07, Association for Computing Machinery, New York, NY, USA (2007), ISBN 9781595938152, `https://doi.org/10.1145/1291151.1291199`

[11] Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs: An Introduction to Programming and Computing. The MIT Press (2018), ISBN 0262534800

[12] Fokkinga, M.: Law and Order in Algorithmics. Ph.D. thesis, University of Twente, 7500 AE Enschede, Netherlands (Feb 1992)

[13] Fokkinga, M.: Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, Department of Computer Science, University of Twente (June 1994), URL `http://doc.utwente.nl/66622/`

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

[15] Gibbons, J.: Generic downwards accumulations. Sci. Comput. Program. **37**(1-3), 37–65 (2000), https://doi.org/10.1016/S0167-6423(99)00022-2
[16] Gibbons, J.: Metamorphisms: Streaming representation-changers. Science of Computer Programming **65**(2), 108–139 (2007), https://doi.org/10.1016/j.scico.2006.01.006
[17] Gibbons, J.: Coding with asymmetric numeral systems. In: Hutton, G. (ed.) Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings, Lecture Notes in Computer Science, vol. 11825, pp. 444–465, Springer (2019), https://doi.org/10.1007/978-3-030-33636-3_16
[18] Gibbons, J.: How to design co-programs. Journal of Functional Programming **31**, e15 (2021), https://doi.org/10.1017/S0956796821000113
[19] Hagino, T.: Category Theoretic Approach to Data Types. Ph.D. thesis, University of Edinburgh (1987)
[20] Hinze, R.: Fun with phantom types. The fun of programming pp. 245–262 (2003)
[21] Hinze, R.: Generic Programming with Adjunctions, pp. 47–129. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), ISBN 978-3-642-32202-0, https://doi.org/10.1007/978-3-642-32202-0_2
[22] Hinze, R.: Adjoint folds and unfolds—an extended study. Sci. Comput. Program. **78**(11), 2108–2159 (Aug 2013), ISSN 0167-6423, https://doi.org/10.1016/j.scico.2012.07.011
[23] Hinze, R., Wu, N.: Histo- and dynamorphisms revisited. In: Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming, WGP '13, pp. 1–12, New York, NY, USA (2013), ISBN 978-1-4503-2389-5, https://doi.org/10.1145/2502488.2502496
[24] Hinze, R., Wu, N.: Unifying structured recursion schemes - an extended study. J. Funct. Program. **26**, 47 (2016)
[25] Hinze, R., Wu, N., Gibbons, J.: Unifying structured recursion schemes. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP '13, pp. 209–220, New York, NY, USA (2013), ISBN 978-1-4503-2326-0, https://doi.org/10.1145/2500365.2500578
[26] Hinze, R., Wu, N., Gibbons, J.: Conjugate hylomorphisms – or: The mother of all structured recursion schemes. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, pp. 527–538, ACM, New York, NY, USA (2015), ISBN 978-1-4503-3300-9, https://doi.org/10.1145/2676726.2676989
[27] Hoare, C.A.R.: Chapter II: Notes on Data Structuring, p. 83–174. Academic Press Ltd., GBR (1972), ISBN 0122005503
[28] Hu, Z., Iwasaki, H., Takeichi, M.: Calculating accumulations. New Generation Computing **17**(2), 153–173 (1999)
[29] Hu, Z., Iwasaki, H., Takeichi, M.: Deriving structural hylomorphisms from recursive definitions. ACM SIGPLAN Notices **31** (12 1999), https://doi.org/10.1145/232629.232637

[30] Hutton, G.: Fold and unfold for program semantics. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming, p. 280–288, ICFP '98, Association for Computing Machinery, New York, NY, USA (1998), ISBN 1581130244, `https://doi.org/10.1145/289423.289457`

[31] Johann, P., Ghani, N.: Initial algebra semantics is enough! In: Della Rocca, S.R. (ed.) Typed Lambda Calculi and Applications, pp. 207–222, Springer Berlin Heidelberg, Berlin, Heidelberg (2007), ISBN 978-3-540-73228-0

[32] Johann, P., Ghani, N.: Foundations for structured programming with gadts. SIGPLAN Not. **43**(1), 297–308 (Jan 2008), ISSN 0362-1340, `https://doi.org/10.1145/1328897.1328475`

[33] Johann, P., Ghani, N.: A principled approach to programming with nested types in haskell. Higher-Order and Symbolic Computation **22**(2), 155–189 (June 2009), `https://doi.org/10.1007/s10990-009-9047-7`

[34] Kabanov, J., Vene, V.: Recursion schemes for dynamic programming. In: Uustalu, T. (ed.) Mathematics of Program Construction, pp. 235–252, Springer Berlin Heidelberg, Berlin, Heidelberg (2006), ISBN 978-3-540-35632-5

[35] Kmett, E.: Time for chronomorphisms. `http://comonad.com/reader/2008/time-for-chronomorphisms/` (2008), accessed: 2020-06-15

[36] Kmett, E.: recursion-schemes: Representing common recursion patterns as higher-order functions (2011), URL `https://hackage.haskell.org/package/recursion-schemes`

[37] Malcolm, G.: Algebraic Data Types and Program Transformation. Ph.D. thesis, University of Groningen (1990)

[38] Malcolm, G.: Data structures and program transformation. Sci. Comput. Program. **14**(2-3), 255–280 (1990), ISSN 0167-6423, `https://doi.org/10.1016/0167-6423(90)90023-7`

[39] Meertens, L.: First steps towards the theory of rose trees. CWI, Amsterdam (1988)

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

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

[42] Okasaki, C.: Purely functional random-access lists. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, p. 86–95, FPCA '95, Association for Computing Machinery, New York, NY, USA (1995), ISBN 0897917197, `https://doi.org/10.1145/224164.224187`

[43] Pardo, A.: Monadic corecursion —definition, fusion laws, and applications—. Electron. Notes Theor. Comput. Sci. **11**(C), 105–139 (May 1998), ISSN 1571-0661, `https://doi.org/10.1016/S1571-0661(04)00055-6`

[44] Pardo, A.: Generic accumulations. In: Gibbons, J., Jeuring, J. (eds.) Generic Programming: IFIP TC2/WG2.1 Working Conference on Generic Program-

ming, International Federation for Information Processing, vol. 115, pp. 49–78, Kluwer Academic Publishers (Jul 2002)

[45] Pardo, A.: Combining datatypes and effects. In: Vene, V., Uustalu, T. (eds.) Advanced Functional Programming, pp. 171–209, Springer Berlin Heidelberg, Berlin, Heidelberg (2005), ISBN 978-3-540-31872-9

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

[47] Uustalu, T., Vene, V.: Comonadic notions of computation **203**(5), 263–284 (2008), https://doi.org/10.1016/j.entcs.2008.05.029

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

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