Histo- and Dynamorphisms Revisited

Ralf Hinze Nicolas Wu [*]

Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England
{ralf.hinze,nicolas.wu}@cs.ox.ac.uk

Abstract

Dynamic programming algorithms embody a widely used programming technique that optimizes recursively defined equations that have repeating subproblems. The standard solution uses arrays to share common results between successive steps, and while effective, this fails to exploit the structural properties present in these problems. Histomorphisms and dynamorphisms have been introduced to expresses such algorithms in terms of structured recursion schemes that leverage this structure. In this paper, we revisit and relate these schemes and show how they can be expressed in terms of recursion schemes from comonads, as well as from recursive coalgebras. Our constructions rely on properties of bialgebras and dicoalgebras, and we are careful to consider optimizations and efficiency concerns. Throughout the paper we illustrate these techniques through several worked-out examples discussed in a tutorial style, and show how a recursive specification can be expressed both as an array-based algorithm as well as one that uses recursion schemes.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—specification techniques

Keywords dynamic programming; recursion schemes; histomorphisms; dynamorphisms

1. Introduction

Many important algorithms can be expressed using recursion equations where solutions are built up from recursive steps. In divide-and-conquer algorithms such recursion equations can often be executed efficiently, since a problem is divided up into *independent* subproblems that are then solved recursively before being combined to form a final solution. However, in the particular case where a subproblem is repeated at different stages in the computation, recursion equations should be considered no more than a specification: the solution to each repeated subproblem is naively recomputed, often leading to exponential complexity that can be avoided. The key observation made by dynamic programming algorithms is that that solutions to subproblems can be memoized and reused when identical subproblems are later encountered, thus preventing the expense of needless recomputation.

The standard approach to implementing a dynamic programming algorithm is to use a table of values to store intermediate results. Such tables are usually indexed by the input parameters of the recursion, and are populated with values the first time a particular subproblem has been encountered. These values are then used when the subproblem is revisited. In a functional programming language with lazy evaluation, such tables are easy to construct, since values are populated as they are demanded. This leads to implementations that closely resemble their recursive counterparts, but do not suffer from their inefficiencies.

We aim to show how categorically-inspired recursion schemes can be used to solve such problems, and we hope that this study will become useful to programmers who are interested in understanding how category theory can be used in the design of algorithms where the structure of the computation is important. This is not the first time that dynamic programming algorithms have been investigated from a categorical perspective, and indeed, this paper provides a general survey of the existing body of work.

Histomorphisms have long been understood as a means of capturing course-of-values recursion, which is sufficient for dynamic programming algorithms where the recursion follows the pattern of the input data [14]. In particular, histomorphisms are constrained to work on problems where the input can be expressed as an initial algebra. However, not all problems follow such a rigid pattern, and dynamorphisms are a more general recursion scheme that was introduced to lift this restriction in the setting of CPO [10]. We extend this development by using recursive coalgebras to prove uniqueness of both histo- and dynamorphisms for a wider range of categories.

In order to make this material accessible, we present solutions to various classic dynamic programming problems. Our development is in Haskell [13], not only because the ensuing programs can be efficiently executed, but also because the language allows us to express solutions that closely resemble the categorical notions that underpin the theory.

On a more theoretical note, we also show how histo- and dynamorphisms relate not only to one another, but also how they can be expressed as recursion schemes from comonads, and as recursion schemes from recursive coalgebras. This paper serves as an extended case-study that builds on previous material, where distributive laws and bialgebras witness the correspondence between adjoint folds and recursion schemes from comonads [8]. We add to this by considering the efficiency of the recursion schemes we study.

This paper makes the following novel contributions:

- We demonstrate how histomorphisms and dynamorphisms can be applied to a number of different problems.
- We show how bialgebras relate histomorphisms to recursion schemes from comonads, and how dicoalgebras relate dynamorphisms to recursion schemes from recursive coalgebras.

[*] This work has been funded by EPSRC grant number EP/J010995/1.

WGP '13, September 28, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2389-5/13/09…$15.00.
http://dx.doi.org/10.1145/2502488.2502496

- We use these relationships to derive optimized versions of histo- and dynamorphisms.
- We use type families to witness efficient implementations of inductive types from base functors in Haskell.

The work in this paper draws significantly from categorical machinery. As such, we assume that the reader has at least some basic knowledge of the categorical trinity: categories, functors and natural transformations. Aside from these, we also assume that the reader has an understanding of initial algebras and comonads; these notions will be introduced formally, but we will not linger long on these constructions. No further knowledge will be required, and we will introduce such notions as cofree comonads, distributive laws, bialgebras, and the (co)-Eilenberg-Moore category when they are required.

The paper is structured as follows. Section 2 introduces a number of dynamic programming algorithms that will be revisited throughout the paper. A brief overview of some of the basic concepts we use is given in Section 3. We introduce histomorphisms in Section 4 and relate these to recursion schemes from comonads in Section 5. We then introduce dynamorphisms in Section 6 and relate these to recursion schemes from recursive coalgebras in Section 7. Finally, we present related work in Section 8, and conclude in Section 9.

2. Dynamic Programming

Before looking at the construction of histo- and dynamorphisms we first take a look at at the different kinds of algorithms that fit under the umbrella of dynamic programming. Dynamic programming relies on the *principle of optimality*, where the optimal solution to a problem can be determined by first breaking the problem into subproblems, optimally solving those subproblems, and combining the ensuing subsolutions into a final answer.

The knapsack problem A classic example of of a dynamic algorithm is the *unbounded knapsack problem*. Suppose we are interested in maximizing the total value of elements that are placed into a knapsack with a fixed weight capacity. The elements are chosen from a set of items that are assigned a particular weight and value, each item being unbounded in number. We might represent the set of items as a list of pairs (w,v), where w is the weight and v is the value. For instance, consider the following list:

$$\begin{aligned}&wvs :: [(\mathbb{N},\mathbb{R})]\\ &wvs = [(12,4),(1,2),(2,2),(1,1),(4,10)]\ .\end{aligned}$$

With a knapsack of capacity 15 the optimal solution is to choose three elements from the 2nd and 5th items, for a total value of 36.

This problem can be solved using a recursive function, that forms the basis of a specification:

$$\begin{aligned}&knapsack_1 :: \mathbb{N} \to \mathbb{R}\\ &knapsack_1\ c = maximum_0\\ &\quad [v + knapsack_1\ (c-w) \mid (w,v) \leftarrow wvs, 0 < w \wedge w \leqslant c]\ .\end{aligned}$$

The value of a knapsack with capacity c is determined by finding the item in wvs that maximizes the value of the knapsack when it has been added: an element of weight w and value v increases the value of a knapsack by v, and decreases its capacity by w. Only items with positive weight that can fit into the knapsack are considered. The function $maximum_0$ returns 0 when given an empty list, and otherwise returns the maximum value in the list.

This specification makes no attempt to be efficient, and naively recomputes the values of knapsacks with capacities that have already been explored. In order to avoid these recomputations, the intermediate results can be stored in a table that is populated by the recursion itself, and used to lookup previously visited values.

$$\begin{aligned}&knapsack_2 :: \mathbb{N} \to \mathbb{R}\\ &knapsack_2\ n = table\ !\ n\ \mathbf{where}\\ &\quad table \qquad\ \ = tabulate\ (0,n)\ knapsack\\ &\quad knapsack\ c = maximum_0\\ &\qquad [v + table\ !\ (c-w) \mid (w,v) \leftarrow wvs, 0 < w \wedge w \leqslant c]\end{aligned}$$

This definition closely mirrors the specification, where the main body of *knapsack* is almost identical to $knapsack_1$. The key difference is that the recursive calls have been replaced by looking up values in a table. This table is constructed by the function *tabulate*, which takes as its arguments the bounds of the array that is to be constructed and a function that produces values for given indices.

$$\begin{aligned}&tabulate :: (Ix\ i) \Rightarrow (i,i) \to (i \to a) \to \mathsf{Array}\ i\ a\\ &tabulate\ ixs\ f = array\ ixs\ [(i, f\ i) \mid i \leftarrow range\ ixs]\end{aligned}$$

Thus the function *knapsack* and the array *table* are mutually recursive: values are initially tabulated by using the function *knapsack*, and *knapsack* makes use of the table to find values that have already been computed. This relies on lazy evaluation, where values are generated only as they are demanded: it is the calling convention that determines which values are calculated next. We shall see how this sets the pattern of how dynamic programming algorithms can be solved using arrays in the problems that follow. Note that there is an unfulfilled proof obligation here: one must show that the recursion is well-founded; it is precisely this proof that histomorphisms and dynamorphisms provide, and, as we shall see, we will have to work hard to transmogrify the original formulation into the form required by these schemes.

Catalan numbers A simple example of a course-of-values program that makes use of all its subcomponents is the evaluation of the *Catalan* numbers. Amongst other things, the Catalan numbers can be used to find the number of distinct well-formed arrangements that can be made with a set of n matching parentheses. This can be expressed by a simple recursive definition.

$$\begin{aligned}&catalan_1 :: \mathbb{N} \to \mathbb{N}\\ &catalan_1\ 0 \qquad\quad = 1\\ &catalan_1\ (n+1) = sum\ [catalan_1\ i * catalan_1\ (n-i) \mid i \leftarrow [0\,..\,n]]\end{aligned}$$

For example, the value of $catalan_1$ 3 is 5, which can be seen through a simple enumeration of the possibilities:

$$()\,()\,(),()\,(()),(())\,(),(()\,()),((()))\ .$$

The recursive solution works by considering all of the different ways of splitting an expression with parentheses.

Strictly speaking this is not a dynamic programming problem, since we are not seeking an optimal solution. However, it does exhibit the same hallmarks: common subproblems are encountered time and again, and there is scope to share solutions between recursive calls to increase the efficiency of this algorithm. As before, we apply the technique dynamic programming, where array-based memoization is used to store and share results.

$$\begin{aligned}&catalan_2 :: \mathbb{N} \to \mathbb{N}\\ &catalan_2\ p = table\ !\ p\ \mathbf{where}\\ &\quad table = tabulate\ (0,p)\ catalan\\ &\quad catalan\ 0 \qquad\quad = 1\\ &\quad catalan\ (n+1) = sum\ [table\ !\ i * table\ !\ (n-i) \mid i \leftarrow [0\,..\,n]]\end{aligned}$$

Again this implementation builds an array that is indexed in place of recursive calls.

Chain matrix multiplication The *chain matrix multiplication* problem concerns finding the minimal number of operations required to multiply a chain of matrices of arbitrary length. The multiplication of a $p \times q$ matrix by a $q \times r$ matrix yields a matrix of size

$p \times r$ in pqr scalar operations. This multiplication is associative, yielding the same result regardless of the order in which more than two matrices are multiplied. However it is easy to show that different parenthesizations can lead to different costs. For example, consider multiplying a chain of three matrices of sizes 2×3, 3×5, and 5×7. There are two solutions, where multiplying the first two matrices and then the third costs 100 operations, whereas multiplying the last two matrices and then the first costs 147 operations.

The naive solution to this problem is to compute the cost of all possible parenthesizations. This algorithm takes time proportional to the Catalan numbers to generate all the different sequences, each of which is checked in isolation. We can improve upon this solution by using dynamic programming, where the result for any (sub)-parenthesization is calculated only once and reused where appropriate. As usual, we start with a recurrence equation that solves the problem. We assume that the matrices $A_1 \ldots A_n$ are given to be multiplied, and matrix A_k has dimensions given by $a_{k-1} \times a_k$.

```
$chain_1 :: (\mathbb{N}, \mathbb{N}) \to \mathbb{N}$
$chain_1\ (i,j)$
    $\mid i == j = 0$
    $\mid i < j\ = minimum\ [a_i * a_{k+1} * a_{j+1} +$
      $chain_1\ (i,k) + chain_1\ (k+1,j) \mid k \leftarrow [i\,..\,j-1]]$
```

(In a sense, the specification is geared towards an imperative array-based solution: the argument to the recursion is represented as a pair, which is an efficient representation of a contiguous segment when the data is globally stored in an array.) This solution makes use of the principle of optimality, by noting that the optimal solutions to subproblems can be combined to form the final solution. In this case, the optimal chain for multiplying matrices $A_i \ldots A_j$ is given by finding the value k that minimizes the number of scalar operations, when the optimal values for chaining matrices $A_i \ldots A_k$ and $A_{k+1} \ldots A_j$ are known. The final answer for this is held in $chain_1\ (1,n)$, where n is the number of matrices that are being multiplied.

To turn this into a more efficient array-based version, we employ the usual technique and memoize the results of the recursion:

```
$chain_2 :: (\mathbb{N}, \mathbb{N}) \to \mathbb{N}$
$chain_2\ (m,n) = table\,!\,(m,n)$ **where**
  $table = tabulate\ ((0,0),(n,n))\ chain$
  $chain\ (i,j)$
    $\mid i == j = 0$
    $\mid i < j\ = minimum\ [a_i * a_{k+1} * a_{j+1} +$
      $table\,!\,(i,k) + table\,!\,(k+1,j) \mid k \leftarrow [i\,..\,j-1]]$ .
```

Although the recursive definition is in two variables, we can capture this quite simply by creating a multi-dimensional array. Note that all the complexity is hidden in the function *tabulate*, which is overloaded on the type of indices.

The bitonic travelling-salesman problem The bitonic travelling-salesman problem is a simple variation of the classic NP-hard travelling-salesman problem: given a set of points and distances between each pair of points, the task is to find the shortest route that visits each point exactly once before returning back to the start. The variation is that the points are assumed to be on a plane, and share no x coordinate. Furthermore, the solutions are restricted to consider only *bitonic tours*: paths that start at the leftmost point, then move strictly towards the rightmost point, and then move strictly left back to the start, having covered all points. For convenience we assume that there are m points, $p_0\,..\,p_m$, ordered by their x coordinate. We denote the distance between the points p_i and p_j by $\overline{p_i\,p_j}$. Under these circumstances, it can be shown that the optimal path is found in $O(m^2)$ time.

First we present a solution that uses strong induction over the naturals. We ensure that the value of $bitonic_1\ n$ is the length of the

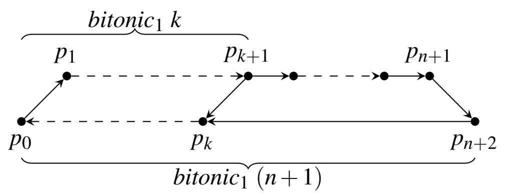

Figure 1. Adding p_{n+2} to the tour given by $bitonic_1\ k$.

shortest tour that includes all the points $p_0\,..\,p_{n+1}$. The base case is $bitonic_1\ 0$, which is simply twice the distance between the first two points. For the inductive case, we assume that the result of $bitonic_1\ i$ is the shortest tour for all $i \leqslant n$, and show how to find the shortest tour for $bitonic_1\ (n+1)$. Consider the diagram in Figure 1. If we add the point p_{n+2}, at the far right of the tour, then this must be connected to the point p_{n+1}, and also to some other point p_k where $k \leqslant n$. Since the tour is bitonic, this implies that there must be a path that connects the points $p_{k+1}\,..\,p_{n+1}$ in succession. Therefore, a bitonic tour with p_{n+2} at its rightmost point is given by *bitonic k* plus these connections, and minus the path between p_k and p_{k+1}. We are obviously interested in finding the shortest tour given by some k. This is expressed precisely in the following recursive function:

```
$bitonic_1 :: \mathbb{N} \to \mathbb{R}$
$bitonic_1\ 0\qquad\quad = 2 * \overline{p_0\,p_1}$
$bitonic_1\ (n+1) = minimum\ [bitonic_1\ k - \overline{p_k\,p_{k+1}} + \overline{p_k\,p_{n+2}}$
    $+\, sum\ [\overline{p_i\,p_{i+1}} \mid i \leftarrow [k+1\,..\,n+1]] \mid k \leftarrow [0\,..\,n]]$ .
```

Using this recurrence, the solution is found in $bitonic_1\ (m-1)$. To turn this into an efficient version we must memoize the results of $bitonic_1\ k$ to avoid the recomputation of subsolutions. We omit this definition, since it is similar to the array-based solutions of the previous examples.

A second solution to the bitonic travelling-salesman problem can be formulated that has quite a different invariant. The recursion equation for $bitonic'_1\ i\ j$ expresses the minimal traversal of the points that starts at p_i, travels strictly left to p_0, and then strictly right to p_j. We assume that $i \leqslant j$, and that all points smaller than j are in the path. Clearly, when $i == j$ we have a cycle: the final answer is to be found in *bitonic m m*.

```
$bitonic'_1 :: (\mathbb{N}, \mathbb{N}) \to \mathbb{R}$
$bitonic'_1\ (0,0) = 0$
$bitonic'_1\ (0,1) = \overline{p_0\,p_1}$
$bitonic'_1\ (i,j)$
    $\mid i < j-1\quad = bitonic'_1\ (i,j-1) + \overline{p_{j-1}\,p_j}$
    $\mid otherwise = minimum\ [bitonic'_1\ (k,i) + \overline{p_k\,p_j} \mid k \leftarrow [0\,..\,i-1]]$
```

There are two base cases for this recursion. The first case is the tour that contains only the point p_0, which has a distance of 0. The second base case is a path that connects p_0 and p_1, which has length $\overline{p_0\,p_1}$. Now we consider a path from p_i to p_j. When $i < j-1$, then we must connect p_{j-1} and p_j, since our invariant is that all points less than j are in the path, so this distance is added to the result of $bitonic'_1\ i\ (j-1)$. Otherwise, either $i == j-1$ or $i == j$, and in both cases we find the minimal path that has one end at i, and the other end going through some k and immediately to j. To turn this into an efficient version we must construct a table in two dimensions. Again, we omit this definition, since as with the previous examples, it follows quite naturally from the recursive specification.

In each of these examples, we turn a recursive definition into a more efficient array-based solution that solves the problem. However, this is unsatisfactory in the sense that we have no guarantee that the recursion is well-defined. In the remainder of the paper we will focus on recursion schemes where the structure of the lookup table comes from the data itself.

3. Background

In this section we introduce some of the basic concepts that will be used in the remainder of the paper, and show how these notions can be implemented in Haskell.

Inductive types Algebras and coalgebras form the basis for the categorical description of structured recursion schemes. Given an endofunctor $\mathsf{F} : \mathscr{C} \to \mathscr{C}$, an F-algebra is a pair (a, A), where $a : \mathsf{F}\,A \to A$ is an arrow and $A : \mathscr{C}$ is an object, which are known as the *action* and *carrier* of the algebra. (This deviates a little from the standard notation (A, a), since it gives us syntax that distinguishes algebras from coalgebras.) An F-homomorphism between algebras (a, A) and (b, B) is an arrow $h : A \to B : \mathscr{C}$ such that $h \cdot a = b \cdot \mathsf{F}\,h$.

$$\begin{tikzcd} \mathsf{F}\,A \arrow[r, "\mathsf{F}\,h"] \arrow[d, "a"'] & \mathsf{F}\,B \arrow[d, "b"] \\ A \arrow[r, "h"'] & B \end{tikzcd}$$

Since F-homomorphisms compose and have an identity, it follows that F-algebras and F-homomorphisms form a category, which we call F-$\mathbf{Alg}(\mathscr{C})$. The initial object of this category, if it exists, is given by $(in, \mu\mathsf{F})$ and called the *initial* F*-algebra*. The initiality implies that to each F-algebra, (a, A), there exists a unique F-homomorphism, $(\!|a|\!) : (in, \mu\mathsf{F}) \to (a, A)$, called a *fold*. The algebra *in* is, in fact, an isomorphism, so $\mu\mathsf{F}$ is a fixed-point of F (the least fixed-point), a fact known as Lambek's lemma [12]. We call $\mu\mathsf{F}$ an *inductive type*.

Dually, given an endofunctor $\mathsf{G} : \mathscr{C} \to \mathscr{C}$, a G-coalgebra is a pair (C, c), where $C : \mathscr{C}$ is the carrier and $c : C \to \mathsf{G}\,C$ is the action of the coalgebra. A G-homomorphism between coalgebras (C, c) and (D, d) is an arrow $h : C \to D : \mathscr{C}$ that satisfies $\mathsf{G}\,h \cdot c = d \cdot h$. Just as before, a category G-$\mathbf{Coalg}(\mathscr{C})$ can be formed from G-coalgebras and G-homomorphisms. The final object of this category, if it exists, is given by $(\nu\mathsf{G}, out)$ and called the *final* G*-coalgebra*. The unique homomorphism to the G-algebra (C, c), called an *unfold*, is written $[\![c]\!] : C \to \nu\mathsf{G}$.

The category F-$\mathbf{Alg}(\mathscr{C})$ has more structure than $\mathscr{C}$. The forgetful or underlying functor $\mathsf{U}^{\mathsf{F}} : \mathsf{F}$-$\mathbf{Alg}(\mathscr{C}) \to \mathscr{C}$ forgets about the additional structure: $\mathsf{U}^{\mathsf{F}}\,(a, A) = A$ and $\mathsf{U}^{\mathsf{F}}\,h = h$. An analogous functor can be defined for coalgebras: $\mathsf{U}_{\mathsf{G}} : \mathsf{G}$-$\mathbf{Coalg}(\mathscr{C}) \to \mathscr{C}$.

Inductive types in Haskell The standard approach to implementing this machinery in Haskell is to make *in* a data constructor of a generic fixed-point constructor: **data** $\mu\mathsf{F} = In\ \{in^{\circ} :: \mathsf{F}\,(\mu\mathsf{F})\}$. We depart from this approach, and instead use Haskell's type classes and family synonyms [4] to witness the isomorphism $in : \mathsf{F}\,(\mu\mathsf{F}) \cong \mu\mathsf{F} : in^{\circ}$. Thus, for inductive types, we introduce the following class:

```
class (Functor F) ⇒ Inductive F where
  type μF :: ∗
  in  :: F (μF) → μF
  in° :: μF → F (μF)
  (|-|) :: (F a → a) → (μF → a)  .
```

Here, *in* is implemented as a function rather than a data constructor, and there is an obligation on the implementer to ensure that *in* and in° are indeed inverses. However, it is possible to define in° in terms of a fold, and vice versa, so these form suitable default implementations:

```
in°  = (|fmap in|)
(|a|) = a · fmap (|a|) · in°  .
```

This allows us to keep the implementation of the fixed point abstract, and in turn, gives us the freedom to associate efficient representations to base functors.

This is particularly useful for primitive types such as natural numbers. The base functor for these is expressed by Nat:

```
data Nat n = Zero | Succ n
instance Functor Nat where
  fmap f Zero     = Zero
  fmap f (Succ x) = Succ (f x)  .
```

The *Inductive* instance for this datatype is simply the natural numbers, which are implemented efficiently as integers in Haskell:

```
instance Inductive Nat where
  type μNat = ℕ
  in Zero       = 0
  in (Succ n)   = n + 1
  in° 0         = Zero
  in° (n + 1)   = Succ n  .
```

This lets us freely use properties of the structure of natural numbers without being too heavily penalized.

Comonads Functional programmers have embraced monads, and to a lesser extent, comonads, to capture effectful and context-sensitive computations. We shall use comonads to model 'recursive calls in context'. A comonad is a functor $\mathsf{N} : \mathscr{C} \to \mathscr{C}$ equipped with a natural transformation $\epsilon : \mathsf{N} \overset{.}{\to} \mathsf{Id}$ (counit), that extracts a value from a context, and a second natural transformation $\delta : \mathsf{N} \overset{.}{\to} \mathsf{N} \circ \mathsf{N}$ (comultiplication) that duplicates a context. These functions are subject to the comonad laws:

$$(\epsilon \circ id_{\mathsf{N}}) \cdot \delta = id_{\mathsf{N}}\ , \tag{1a}$$

$$(id_{\mathsf{N}} \circ \epsilon) \cdot \delta = id_{\mathsf{N}}\ , \tag{1b}$$

$$(\delta \circ id_{\mathsf{N}}) \cdot \delta = (id_{\mathsf{N}} \circ \delta) \cdot \delta\ . \tag{1c}$$

The first two properties, the counit laws, state that duplicating a context and then discarding a duplicate is the same as doing nothing. The third property, the coassociative law, equates the two ways of duplicating a context twice. Here we use categorical notation, where natural transformations can be composed horizontally ($\circ$), and vertically ($\cdot$). (Recall that given functors $\mathsf{F}, \mathsf{F}' : \mathscr{C} \to \mathscr{D}$ and $\mathsf{G}, \mathsf{G}' : \mathscr{D} \to \mathscr{E}$, and natural transformations $\alpha : \mathsf{F} \overset{.}{\to} \mathsf{F}'$ and $\beta : \mathsf{G} \overset{.}{\to} \mathsf{G}'$, the horizontal composition $\beta \circ \alpha : \mathsf{G} \circ \mathsf{F} \overset{.}{\to} \mathsf{G}' \circ \mathsf{F}'$ has components $(\beta \circ \alpha)\,A = \mathsf{G}'\,(\alpha\,A) \cdot \beta\,(\mathsf{F}\,A) = \beta\,(\mathsf{F}'\,A) \cdot \mathsf{G}\,(\alpha\,A)$, for all A.)

4. Histomorphisms

Dynamic programming algorithms make use of solutions to previously visited subproblems to compute the values of new ones. In other words, values that are computed are placed in some context, and then extracted from that context when needed. As a first approximation, this pattern is captured by a *histomorphism*, which has access to the whole history of a computation. Recall that a fold is made available the result of the recursive calls on the *immediate* substructures. By contrast, a histomorphism can make use of the results of the recursive calls on *all* substructures.

Before we go into the details of how a histomorphism is defined, we first introduce the so-called *cofree comonad* of a functor F, which we write as F_{∞}. This comonad serves to provide the context in which results are placed during recursive calls. Loosely speaking, it serves as a generic counterpart of the memo tables implemented by arrays above.

Cofree comonad Categorically speaking, the cofree comonad comes from the following relationship, called an *adjunction*, between the category of coalgebras and its underlying category.

$$\begin{tikzcd} \mathscr{C} \arrow[r, bend right, "\mathsf{Cofree}_\mathsf{F}"'] \arrow[r, phantom, "\perp"] & \mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C}) \arrow[l, bend right, "\mathsf{U}_\mathsf{F}"'] \end{tikzcd} \qquad (2)$$

The forgetful functor U_F has a right adjoint $\mathsf{Cofree}_\mathsf{F}$ that maps an object A to the cofree coalgebra $\mathsf{Cofree}_\mathsf{F}\ A = (\mathsf{F}_\infty\ A, \mathit{tail}_\infty\ A)$. Very generally speaking, $\mathsf{Cofree}_\mathsf{F}$ can be used to capture the behaviour of 'systems'. It may help to think of the functor F as a static description of all possible transitions for a class of different systems, and of the object A as a type of system states. The elements of $\mathsf{F}_\infty\ A$ then capture the entire behaviour of a system as the infinite unfolding of all possible transitions. The action of the cofree coalgebra $\mathit{tail}_\infty : \mathsf{F}_\infty\ A \to \mathsf{F}\ (\mathsf{F}_\infty\ A)$ maps such a description to the F-structure of all possible successor systems.

The adjunction provides further infrastructure: the so-called unit and counit, which are natural transformation that obey certain laws. Given a state-transition function expressed as an F-coalgebra $a : A \to \mathsf{F}\ A$, the unit η which we write $\eta\ (A,a) = [\![a]\!] : A \to \mathsf{F}_\infty\ A$ constructs the infinite unfolding from a given initial state. The counit ϵ which we denote $\mathit{head}_\infty : \mathsf{F}_\infty\ A \to A$ extracts the initial state of a system. This data satisfies an important property, which establishes a bijection between certain arrows in $\mathscr{C}$ and certain arrows in F-$\mathbf{Coalg}(\mathscr{C})$. Specifically, an F-coalgebra homomorphism $g : (A,a) \to \mathsf{Cofree}_\mathsf{F}\ B$ is uniquely determined by a mapping $f : A \to B$ from states of type A to observations of type B. This *universal property* can be neatly expressed as an equivalence:

$$f = \mathit{head}_\infty\ B \cdot g \iff \mathsf{F}_\infty f \cdot [\![a]\!] = g\ , \qquad (3)$$

for all arrows $f : A \to B$ and homomorphisms $g : (A,a) \to \mathsf{Cofree}_\mathsf{F}\ B$.

Every adjunction induces a comonad [9]. The adjunction (2) gives rise to the cofree comonad $\mathsf{F}_\infty = \mathsf{U}_\mathsf{F} \circ \mathsf{Cofree}_\mathsf{F}$.

The cofree comonad in Haskell One can show that final coalgebras and cofree coalgebras are interdefinable. In one direction we have $\nu\mathsf{F} \cong \mathsf{F}_\infty\ 1$ where 1 is the final object. In the other direction we have $\mathsf{F}_\infty\ A \cong \nu\ X\ .\ A \times \mathsf{F}\ X$, which forms the basis for an implementation in Haskell. Using Haskell's higher-kinded datatypes, $\mathsf{F}_\infty\ A$ can be readily implemented as follows.

$$\begin{array}{l} \mathbf{data}\ \mathsf{F}_\infty\ a = \mathit{Cons}_\infty\ \{\mathit{head}_\infty :: a, \mathit{tail}_\infty :: \mathsf{F}\ (\mathsf{F}_\infty\ a)\} \\ \mathbf{instance}\ (\mathit{Functor}\ \mathsf{F}) \Rightarrow \mathit{Functor}\ (\mathsf{F}_\infty)\ \mathbf{where} \\ \quad \mathit{fmap}\ f\ (\mathit{Cons}_\infty\ a\ ts) = \mathit{Cons}_\infty\ (f\ a)\ (\mathit{fmap}\ (\mathit{fmap}\ f)\ ts) \end{array}$$

Here, Cons_∞ is the inverse of the isomorphism $\mathit{head}_\infty \vartriangle \mathit{tail}_\infty$, where $(\vartriangle)$ is the *split* operator:

$$\begin{array}{l} (\vartriangle) :: (a \to b_1) \to (a \to b_2) \to a \to (b_1, b_2) \\ f \vartriangle g = \lambda x \to (f\ x, g\ x)\ . \end{array}$$

The type F_∞ can be seen as the type of generalized streams of observations—it behaves as a 'stream' because each successive layer has a head_∞ that contains a value, and 'generalized' because the 'tail' is an F-structure of 'streams' rather than just a single one. A generalized stream is, in fact, very similar to a generalized rose tree, except that the latter is usually seen as an element of an inductive type, whereas this construction is patently coinductive. If we instantiate the base functor of the cofree comonad to Id, we obtain the type of simple streams.

Given a coalgebra $a : A \to \mathsf{F}\ A$, the implementation of the unit $[\![a]\!] : a \to \mathsf{F}_\infty\ a$ is fairly straightforward, where the function $\mathit{cons}_\infty :: (a, \mathsf{F}\ (\mathsf{F}_\infty\ a)) \to \mathsf{F}_\infty\ a$ is the uncurried version of Cons_∞:

$$\begin{array}{l} [\![\text{-}]\!] :: (\mathit{Functor}\ \mathsf{F}) \Rightarrow (a \to \mathsf{F}\ a) \to (a \to \mathsf{F}_\infty\ a) \\ [\![a]\!] = h\ \mathbf{where}\ h = \mathit{cons}_\infty \cdot (\mathit{id} \vartriangle \mathit{fmap}\ h \cdot a)\ . \end{array}$$

This takes an initial seed that is used to create the head of the structure, and is also combined with the algebra to recursively grow the next level of values in the tails.

As its name suggests, the cofree comonad is comonadic, and so comes equipped with a means of extracting a value from its context, $\epsilon = \mathit{head}_\infty$, and a means of duplicating a context, $\delta = [\![\mathit{tail}_\infty]\!]$, which uses an entire F_∞-structure as the state!

We have noted above that Id_∞ yields the type of streams. A more interesting base functor is Nat which gives rise to the type Nat_∞ of non-empty colists. For example, the call $[\![\mathit{in}^\circ]\!]\ 2$ generates the colist

$$\mathit{Cons}_\infty\ 2\ (\mathit{Succ}\ (\mathit{Cons}_\infty\ 1\ (\mathit{Succ}\ (\mathit{Cons}_\infty\ 0\ \mathit{Zero}))))\ .$$

This corresponds to the list containing 2 and all of its predecessors.

Histomorphisms With these basics in place, we are now ready to give the original formulation of histomorphisms [14].

The argument to a histomorphism is a 'context-sensitive' algebra $a : \mathsf{F}\ (\mathsf{F}_\infty\ A) \to A$ that works on a structure that contains all the recursive subsolutions, and combines these to form a new solution of type A. Informally, $\mathsf{F}_\infty\ A$ is a hierarchical memo-table where each successive level contains a subsolution of type A together with an F-substructure; the deepest level is the base case of the datatype.

A *histomorphism* is defined to be the unique solution $x : \mu\mathsf{F} \to A$ of the equation:

$$x \cdot \mathit{in} = a \cdot \mathsf{F}\ (\mathsf{F}_\infty\ x \cdot [\![\mathit{in}^\circ]\!])\ . \qquad (4)$$

The coalgebra $[\![\mathit{in}^\circ]\!] : \mu\mathsf{F} \to \mathsf{F}_\infty\ (\mu\mathsf{F})$ turns an element of an inductive type into a table of all substructures. The histomorphism x is recursively applied to each of the substructures, making the results of the recursive calls readily available to the algebra a.

Remark. There are a number of different variations of this definition, depending on how the argument of F is expressed. In the definition above we have used a formulation that is based on the unit of the adjunction (2): $h = \mathsf{F}_\infty\ x \cdot [\![\mathit{in}^\circ]\!]$. The original characterization by Uustalu and Vene [14] is recovered if we identify $\mathsf{F}_\infty\ A$ and $\nu\ X\ .\ A \times \mathsf{F}\ X$, thus giving us $h = [\![x \vartriangle \mathit{in}^\circ]\!]$. □

Equation (4) specifies the notion of an histomorphism, however, it does *not* serve as a blue-print for an efficient implementation. Indeed, as an implementation it is typically exponential in the sense that the annotated tree is recomputed at every recursive call. Also, at the outset it is not clear that the equation (4) has a unique solution. We postpone both issues until Section 5 where we attack them in a more general setting.

Histomorphisms in Haskell We can turn the naive definition of a histomorphism into a recursion scheme in Haskell by making the most of the fact that *in* has an inverse, in°, which is provided by the *Inductive* typeclass:

$$\begin{array}{l} \mathit{histo}_1 :: (\mathit{Inductive}\ \mathsf{F}) \Rightarrow (\mathsf{F}\ (\mathsf{F}_\infty\ a) \to a) \to (\mu\mathsf{F} \to a) \\ \mathit{histo}_1\ a = x\ \mathbf{where}\ x = a \cdot \mathit{fmap}\ (\mathit{fmap}\ x \cdot [\![\mathit{in}^\circ]\!]) \cdot \mathit{in}^\circ\ . \end{array}$$

The histomorphism first deconstructs a recursive type to expose one level of its base functor. The function $[\![\mathit{in}^\circ]\!] :: \mu\mathsf{F} \to \mathsf{F}_\infty\ (\mu\mathsf{F})$ is then applied to the functor arguments, turning them into a table of predecessors, before the histomorphism is recursively applied with $\mathit{fmap}\ (\mathit{histo}_1\ a)$ to produce subsolutions as labels to each level. Finally, the algebra a takes this structure that contains all subsolutions, and combines them to form a final solution.

5. Recursion Schemes From Comonads

Histomorphisms involve both an algebra and a coalgebra, and combine them in an interesting way. We have noted above that $[\![\mathit{in}^\circ]\!]$ is a coalgebra, but it is actually a bit more: it is a coalgebra *for the comonad* F_∞. Furthermore, the algebra *in* and the coalgebra $[\![\mathit{in}^\circ]\!]$ go hand-in-hand. They are related by a so-called *distributive law* $\lambda : \mathsf{F} \circ \mathsf{F}_\infty \overset{\cdot}{\to} \mathsf{F}_\infty \circ \mathsf{F}$ and form what is known as a λ-bialgebra, a combination of an algebra and a coalgebra with a common carrier. In particular, *in* and $[\![in^\circ]\!]$ satisfy the so-called pentagonal law.

$$\begin{tikzcd}
\mathsf{F}\,(\mu\mathsf{F}) \arrow[d, "in"'] \arrow[r, "\mathsf{F}\,[\![in^\circ]\!]"] & \mathsf{F}\,(\mathsf{F}_\infty\,(\mu\mathsf{F})) \arrow[d, "\lambda\,(\mu\mathsf{F})"] \\
\mu\mathsf{F} \arrow[d, "{[\![in^\circ]\!]}"'] & \mathsf{F}_\infty\,(\mathsf{F}\,(\mu\mathsf{F})) \arrow[dl, "\mathsf{F}_\infty\,in"] \\
\mathsf{F}_\infty\,(\mu\mathsf{F}) &
\end{tikzcd} \tag{5}$$

Loosely speaking, the distributive law $\lambda : \mathsf{F} \circ \mathsf{F}_\infty \dot{\to} \mathsf{F}_\infty \circ \mathsf{F}$ defined

$$\lambda\,A = \mathsf{F}_\infty\,(\mathsf{F}\,(head_\infty\,A)) \cdot [\![\mathsf{F}\,(tail_\infty\,A)]\!] \tag{6}$$

allows us to swap the functors F and F_∞.

The cofree comonad is by no means special. In fact, histomorphisms are an instance of recursion schemes from comonads [15]. We postpone a formal introduction of the scheme after we have provided the necessary background in the following section, which can be skipped by those already familiar with the material.

5.1 Background

Coalgebras for a comonad A coalgebra for a comonad N is an N-coalgebra (C, c) that respects ϵ and δ:

$$\epsilon\,C \cdot c = id_C \ , \tag{7a}$$
$$\delta\,C \cdot c = \mathsf{N}\,c \cdot c \ . \tag{7b}$$

If we first create a context using c and then focus, we obtain the original value. Creating a nested context is the same as first creating a context and then duplicating it. For example, the so-called cofree coalgebra $(\mathsf{N}\,C, \delta\,C)$ is respectful, which follows directly from (1b) and (1c).

Coalgebras that respect ϵ and δ and N-coalgebra homomorphisms form a category, known as the *(co)-Eilenberg-Moore category* and denoted $\mathscr{C}_\mathsf{N}$.

Eilenberg-Moore construction As noted above, every adjunction generates a comonad. The converse is also true: every comonad N induces an adjunction that generates N—in fact, in two canonical ways. One construction was discovered by Kleisli [11], the other by Eilenberg and Moore [5]. We shall need the latter, which constructs a right adjoint to the forgetful functor $\mathsf{U}_\mathsf{N} : \mathscr{C}_\mathsf{N} \to \mathscr{C}$.

$$\begin{tikzcd}
\mathscr{C} \arrow[r, bend right, "\mathsf{Cofree}_\mathsf{N}"'] \arrow[r, phantom, "\perp"] & \mathscr{C}_\mathsf{N} \arrow[l, bend right, "\mathsf{U}_\mathsf{N}"']
\end{tikzcd}$$

The functor $\mathsf{Cofree}_\mathsf{N}$ maps an object to the cofree coalgebra for N:

$$\mathsf{Cofree}_\mathsf{N}\,B = (\mathsf{N}\,B, \delta\,B) \ , \tag{8a}$$
$$\mathsf{Cofree}_\mathsf{N}\,f = \mathsf{N}\,f \ . \tag{8b}$$

The adjunction establishes a bijection between certain arrows in $\mathscr{C}$ and certain arrows in $\mathscr{C}_\mathsf{N}$. Specifically, an N-coalgebra homomorphism $h : (A, a) \to \mathsf{Cofree}_\mathsf{N}\,B$ is uniquely determined by an arrow $f : A \to B$ in $\mathscr{C}$. As before, this *universal property* can be expressed as an equivalence:

$$f = \epsilon\,B \cdot h \iff \mathsf{N}\,f \cdot a = h \ , \tag{9}$$

for all arrows $f : A \to B$ and homomorphisms $h : (A, a) \to (\mathsf{N}\,B, \delta\,B)$. The homomorphism h is also called the *transpose* of f and is denoted $\lfloor f \rfloor = \mathsf{N}\,f \cdot a$. Conversely, f is the transpose of h, denoted $\lceil h \rceil = \epsilon\,B \cdot h$.

Eilenberg-Moore categories generalize categories of coalgebras: we have $\mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C}) \cong \mathscr{C}_\mathsf{N}$ where $\mathsf{N} = \mathsf{F}_\infty$ is the cofree comonad. In particular, F-coalgebra homomorphisms are in 1-1 correspondence to N-coalgebra homomorphisms:

$$\begin{tikzcd}
A \arrow[r, "h"] \arrow[d, "a"'] & B \arrow[d, "b"] & \arrow[r, phantom, "\iff"] & & A \arrow[r, "h"] \arrow[d, "{[\![a]\!]}"'] & B \arrow[d, "{[\![b]\!]}"] \\
\mathsf{F}\,A \arrow[r, "\mathsf{F}\,h"'] & \mathsf{F}\,B & & & \mathsf{F}_\infty\,A \arrow[r, "\mathsf{F}_\infty\,h"'] & \mathsf{F}_\infty\,B
\end{tikzcd} \ . \tag{10}$$

Note that the isomorphism $\mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C}) \cong \mathscr{C}_\mathsf{N}$ implies that $[\![a]\!]$ is always a coalgebra for the comonad F_∞. Conversely, each respectful F_∞-coalgebra is of this form.

Distributive laws A distributive law $\lambda : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$ of an endofunctor F over a comonad N is a natural transformation satisfying the two coherence conditions:

$$(\epsilon \circ \mathsf{F}) \cdot \lambda = \mathsf{F} \circ \epsilon \ , \tag{11a}$$
$$(\delta \circ \mathsf{F}) \cdot \lambda = (\mathsf{N} \circ \lambda) \cdot (\lambda \circ \mathsf{N}) \cdot (\mathsf{F} \circ \delta) \ . \tag{11b}$$

The first law has type $\mathsf{F} \circ \mathsf{N} \to \mathsf{F}$, and states that there are two equivalent ways of extracting a value from a comonadic context that is nested in a functor: either by first exposing the comonad to the outside by applying a distributive law, and then extracting the functorial value from the comonadic context; or by working directly inside the functor, and extracting a value from the comonadic context that is there. The second law has type $\mathsf{F} \circ \mathsf{N} \to \mathsf{N} \circ \mathsf{N} \circ \mathsf{F}$ and states that pushing a functorial value into a context and then duplicating the context is equivalent to first duplicating the context embedded in a functor, and then shifting the functorial value inside the contexts.

One can show that the distributive law $\lambda : \mathsf{F} \circ \mathsf{F}_\infty \dot{\to} \mathsf{F}_\infty \circ \mathsf{F}$ defined in equation (6) obeys these laws; the proof is beyond the scope of this paper.

Bialgebras A bialgebra combines an algebra and a coalgebra with a common carrier. Bialgebras come in many flavours; we need the variant that combines F-algebras and coalgebras for a comonad N. The two functors have to interact coherently, described by a distributive law.

Let $\lambda : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$ be a distributive law for the endofunctor F over the comonad N. A λ-*bialgebra* (a, X, c) consists of an F-algebra a and a coalgebra c for the comonad N such that the *pentagonal law* holds:

$$c \cdot a = \mathsf{N}\,a \cdot \lambda\,X \cdot \mathsf{F}\,c \ . \tag{12}$$

Loosely speaking, this law allows us to swap the algebra a and the coalgebra c. A λ-bialgebra homomorphism is both simultaneously an F-algebra and an N-coalgebra homomorphism.

The pentagonal law (12) has two asymmetric renderings, which identify the algebra a and the coalgebra c as homomorphisms.

$$\begin{tikzcd}
\mathsf{F}\,X \arrow[r, "\mathsf{F}\,c"] \arrow[d, "a"'] & \mathsf{F}\,(\mathsf{N}\,X) \arrow[d, "\mathsf{N}\,a \cdot \lambda\,X"] \\
X \arrow[r, "c"'] & \mathsf{N}\,X
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{F}\,X \arrow[d, "a"'] \arrow[r, "\mathsf{F}\,c"] & \mathsf{F}\,(\mathsf{N}\,X) \arrow[d, "\lambda\,X"] \\
X \arrow[d, "c"'] & \mathsf{N}\,(\mathsf{F}\,X) \arrow[dl, "\mathsf{N}\,a"] \\
\mathsf{N}\,X &
\end{tikzcd}
\qquad
\begin{tikzcd}
X \arrow[d, "c"'] & \mathsf{F}\,X \arrow[l, "a"'] \arrow[d, "\lambda\,X \cdot \mathsf{F}\,c"] \\
\mathsf{N}\,X & \mathsf{N}\,(\mathsf{F}\,X) \arrow[l, "\mathsf{N}\,a"]
\end{tikzcd}$$

The diagram on the left shows that c is an F-algebra homomorphism. Dually, the diagram on the right identifies a as an N-coalgebra homomorphism.

5.2 Recursion Schemes from Comonads

Now that the terminology is in place, we are in a position to generalize histomorphisms to recursion schemes from comonads [15].

These form a general recursion principle that makes use of a comonad N to provide 'contextual information' to the algebra of the to-be-defined function.

Let $\lambda : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$ be a distributive law, and let $(in, \mu\mathsf{F}, c)$ be a λ-bialgebra, where $c = \llparenthesis \mathsf{N}\ in \cdot \lambda\ (\mu\mathsf{F}) \rrparenthesis$. For any $(\mathsf{F} \circ \mathsf{N})$-algebra (b, B) there is a unique arrow $f : \mu\mathsf{F} \to B$ such that

$$f \cdot in = b \cdot \mathsf{F}\ (\mathsf{N} f \cdot c)\ . \tag{13}$$

The composition $\mathsf{N} f \cdot c$ creates a context that makes the results of recursive calls available to the algebra b. Note that b is a 'context-sensitive' algebra—an $(\mathsf{F} \circ \mathsf{N})$-algebra, rather than merely an F-algebra.

One way to prove uniqueness is to use the Eilenberg-Moore adjunction (9) to relate solutions of equation (13) to certain λ-bialgebra homomorphisms. Abstracting away from in and identifying $\mathsf{N} f \cdot c$ as the transpose of f, one can establish the following equivalence

$$f \cdot a = b \cdot \mathsf{F}\ h \iff h \cdot a = \lfloor b \rfloor \cdot \mathsf{F}\ h\ , \tag{14}$$

where $h = \lfloor f \rfloor = \mathsf{N} f \cdot c$ is the transpose of f. The diagrammatical rendering makes explicit that h is not only an F-algebra homomorphism but also a λ-bialgebra homomorphism.

$$\begin{tikzcd}
\mathsf{F}\,A \arrow[r, "\mathsf{F}\,h"] \arrow[d, "a"'] & \mathsf{F}\,(\mathsf{N}\,B) \arrow[d, "b"] \\
A \arrow[r, "f"'] & B
\end{tikzcd}
\iff
\begin{tikzcd}
\mathsf{F}\,A \arrow[r, "\mathsf{F}\,h"] \arrow[d, "a"'] & \mathsf{F}\,(\mathsf{N}\,B) \arrow[d, "\lfloor b \rfloor"] \\
A \arrow[r, "h"'] \arrow[d, "c"'] & \mathsf{N}\,B \arrow[d, "\delta\,B"] \\
\mathsf{N}\,A \arrow[r, "\mathsf{N}\,h"'] & \mathsf{N}\,(\mathsf{N}\,B)
\end{tikzcd} \tag{15}$$

For the proof of this fact we refer to [8]. Now, if a is in, the action of the initial algebra, the homomorphism h is uniquely defined since it is a fold, and hence f is uniquely defined as well.

Histomorphisms revisited To show that the original formulation of histomorphisms is an instance of this scheme, we must show that in and $\llbracket in^{\circ} \rrbracket$ form a λ-bialgebra, where $\lambda : \mathsf{F} \circ \mathsf{F}_{\infty} \dot{\to} \mathsf{F}_{\infty} \circ \mathsf{F}$. To this end we make use of the following 1-1 correspondence between id-bialgebras and λ-bialgebras, which is a consequence of the fact that $\mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C}) \cong \mathscr{C}_{\mathsf{N}}$.

$$\begin{tikzcd}
\mathsf{F}\,X \arrow[dr, "\mathsf{F}\,c"] \arrow[dd, "a"'] & \\
 & \mathsf{F}\,(\mathsf{F}\,X) \arrow[dd, "id\,X"] \\
X \arrow[dd, "c"'] & \\
 & \mathsf{F}\,(\mathsf{F}\,X) \arrow[dl, "\mathsf{F}\,a"] \\
\mathsf{F}\,X &
\end{tikzcd}
\iff
\begin{tikzcd}
\mathsf{F}\,X \arrow[dr, "\mathsf{F}\llbracket c \rrbracket"] \arrow[dd, "a"'] & \\
 & \mathsf{F}\,(\mathsf{F}_{\infty}\,X) \arrow[dd, "\lambda\,X"] \\
X \arrow[dd, "\llbracket c \rrbracket"'] & \\
 & \mathsf{F}_{\infty}\,(\mathsf{F}\,X) \arrow[dl, "\mathsf{F}_{\infty}\,a"] \\
\mathsf{F}_{\infty}\,X &
\end{tikzcd} \tag{16}$$

Note that for id-bialgebras there is no coherence requirement on the F-coalgebra as F is just a functor. Furthermore, recall that $\llbracket c \rrbracket$ is always a coalgebra for the comonad F_{∞} and that each respectful F_{∞}-coalgebra is of this form.

The proof obligation that $(in, \mu\mathsf{F}, \llbracket in^{\circ} \rrbracket)$ is a λ-bialgebra is now easy to discharge.

$$\begin{aligned}
& \llbracket in^{\circ} \rrbracket \cdot in = \mathsf{F}_{\infty}\ in \cdot \lambda\ (\mu\mathsf{F}) \cdot \mathsf{F}\,\llbracket in^{\circ} \rrbracket \\
\iff & \quad \{\ (16)\ \} \\
& in^{\circ} \cdot in = \mathsf{F}\ in \cdot id\ (\mu\mathsf{F}) \cdot \mathsf{F}\ in^{\circ}\ .
\end{aligned}$$

The latter equation holds trivially.

Efficiency improvements We have noted before that equation (4) is merely a specification of a histomorphism. Even though it is executable, it is not fit for public consumption as it implements the naive recursive definition, which often leads to an exponential running time. In a sense, the original definitions of *knapsack* and friends suffer from two problems: First, it is not clear that the recursion equations have a solution—framing the algorithm as an instance of equation (4) solves this problem; and second, as a program the recursion equations are horribly inefficient—we tackle this problem next.

Because of the 1-1 correspondence (14) we can implement $f : \mu\mathsf{F} \to B$ in terms of $h : \mu\mathsf{F} \to \mathsf{F}_{\infty}\ B$, which constructs an entire table of answers: $f = \lceil h \rceil = head_{\infty}\ B \cdot h = head_{\infty}\ B \cdot \llparenthesis \lfloor b \rfloor \rrparenthesis$. So it remains to derive an efficient implementation of $\lfloor b \rfloor$. To this end observe that $(\lfloor b \rfloor, \mathsf{F}_{\infty}\ B, \delta\ B)$ forms a λ-bialgebra, which is also related to an id-bialgebra.

$$\begin{tikzcd}
\mathsf{F}\,(\mathsf{F}_{\infty}\,B) \arrow[dr, "\mathsf{F}\,(tail_{\infty}\,B)"] \arrow[dd, "\lfloor b \rfloor"'] & \\
 & \mathsf{F}\,(\mathsf{F}\,(\mathsf{F}_{\infty}\,B)) \arrow[dd, "id\,(\mathsf{F}_{\infty}\,B)"] \\
\mathsf{F}_{\infty}\,B \arrow[dd, "tail_{\infty}\,B"'] & \\
 & \mathsf{F}\,(\mathsf{F}\,(\mathsf{F}_{\infty}\,B)) \arrow[dl, "\mathsf{F}\,\lfloor b \rfloor"] \\
\mathsf{F}\,(\mathsf{F}_{\infty}\,B) &
\end{tikzcd}
\iff
\begin{tikzcd}
\mathsf{F}\,(\mathsf{F}_{\infty}\,B) \arrow[dr, "\mathsf{F}\llbracket tail_{\infty}\,B \rrbracket"] \arrow[dd, "\lfloor b \rfloor"'] & \\
 & \mathsf{F}\,(\mathsf{F}_{\infty}\,(\mathsf{F}_{\infty}\,B)) \arrow[dd, "\lambda\,(\mathsf{F}_{\infty}\,B)"] \\
\mathsf{F}_{\infty}\,B \arrow[dd, "\llbracket tail_{\infty}\,B \rrbracket"'] & \\
 & \mathsf{F}_{\infty}\,(\mathsf{F}\,(\mathsf{F}_{\infty}\,B)) \arrow[dl, "\mathsf{F}_{\infty}\,\lfloor b \rfloor"] \\
\mathsf{F}_{\infty}\,(\mathsf{F}_{\infty}\,B) &
\end{tikzcd}$$

The diagram on the left identifies $\lfloor b \rfloor$ as an F-coalgebra homomorphism: $tail_{\infty}\ B \cdot \lfloor b \rfloor = \mathsf{F}\ \lfloor b \rfloor \cdot \mathsf{F}\ (tail_{\infty}\ B)$. Since furthermore $head_{\infty}\ B \cdot \lfloor b \rfloor = b$ using (9), we can invoke the universal property of cofree coalgebras (3) and conclude

$$\lfloor b \rfloor = \mathsf{F}_{\infty}\ b \cdot \llbracket \mathsf{F}\ (tail_{\infty}\ B) \rrbracket\ .$$

Consequently, the histomorphism f is given by

$$f = head_{\infty}\ B \cdot \llparenthesis \mathsf{F}_{\infty}\ b \cdot \llbracket \mathsf{F}\ (tail_{\infty}\ B) \rrbracket \rrparenthesis\ .$$

Loosely speaking, we have managed to turn the exponential specification into an implementation with a quadratic running-time. The fold makes a single sweep through the input structure; for each level the context-sensitive algebra b is mapped over the table to create a table for the next level of recursion. (Of course, all of this depends on the particulars of F and b, which is why we said "loosely".)

Ideally, we would like b to be invoked only once per level. Interestingly, we can achieve this goal if we make use of the fact that h is a λ-bialgebra homomorphism; a blend of both an F-algebra and an F_{∞}-coalgebra homomorphism. Furthermore recall that F-coalgebra homomorphisms are in 1-1 correspondence to F_{∞}-coalgebra homomorphisms (10).

$$\begin{tikzcd}
\mathsf{F}\,(\mu\mathsf{F}) \arrow[r, "\mathsf{F}\,h"] \arrow[d, "in"'] & \mathsf{F}\,(\mathsf{F}_{\infty}\,B) \arrow[d, "\lfloor b \rfloor"] \\
\mu\mathsf{F} \arrow[r, "h"'] \arrow[d, "in^{\circ}"'] & \mathsf{F}_{\infty}\,B \arrow[d, "tail_{\infty}\,B"] \\
\mathsf{F}\,(\mu\mathsf{F}) \arrow[r, "\mathsf{F}\,h"'] & \mathsf{F}\,(\mathsf{F}_{\infty}\,B)
\end{tikzcd}
\iff
\begin{tikzcd}
\mathsf{F}\,(\mu\mathsf{F}) \arrow[r, "\mathsf{F}\,h"] \arrow[d, "in"'] & \mathsf{F}\,(\mathsf{F}_{\infty}\,B) \arrow[d, "\lfloor b \rfloor"] \\
\mu\mathsf{F} \arrow[r, "h"'] \arrow[d, "\llbracket in^{\circ} \rrbracket"'] & \mathsf{F}_{\infty}\,B \arrow[d, "\llbracket tail_{\infty}\,B \rrbracket"] \\
\mathsf{F}_{\infty}\,(\mu\mathsf{F}) \arrow[r, "\mathsf{F}_{\infty}\,h"'] & \mathsf{F}_{\infty}\,(\mathsf{F}_{\infty}\,B)
\end{tikzcd}$$

Now we use the fact that h is both an F-algebra and an F-coalgebra homomorphism.

$$
\begin{array}{ll}
& h \cdot in = \lfloor b \rfloor \cdot \mathsf{F}\, h \;\wedge\; \mathsf{F}\, h \cdot in^{\circ} = tail_{\infty}\, B \cdot h \\
\Longrightarrow & \{ \text{ Leibniz } \} \\
& head_{\infty}\, B \cdot h \cdot in = head_{\infty}\, B \cdot \lfloor b \rfloor \cdot \mathsf{F}\, h \;\wedge\; tail_{\infty}\, B \cdot h = \mathsf{F}\, h \cdot in^{\circ} \\
\Longleftrightarrow & \{ \; head_{\infty}\, B \cdot \lfloor b \rfloor = b \text{ (9) and } in \text{ isomorphism } \} \\
& head_{\infty}\, B \cdot h \cdot in = b \cdot \mathsf{F}\, h \;\wedge\; tail_{\infty}\, B \cdot h \cdot in = \mathsf{F}\, h \\
\Longleftrightarrow & \{ \text{ products } \} \\
& head_{\infty}\, B \cdot h \cdot in \vartriangle tail_{\infty}\, B \cdot h \cdot in = b \cdot \mathsf{F}\, h \vartriangle \mathsf{F}\, h \\
\Longleftrightarrow & \{ \text{ fusion } \} \\
& (head_{\infty}\, B \vartriangle tail_{\infty}\, B) \cdot h \cdot in = (b \vartriangle id) \cdot \mathsf{F}\, h \\
\Longleftrightarrow & \{ \; head_{\infty}\, B \vartriangle tail_{\infty}\, B \text{ isomorphism } \} \\
& h \cdot in = cons_{\infty} \cdot (b \vartriangle id) \cdot \mathsf{F}\, h
\end{array}
$$

Thus, $h = (\!|cons_{\infty} \cdot (b \vartriangle id)|\!)$ and consequently

$$f = head_{\infty}\, B \cdot (\!|cons_{\infty} \cdot (b \vartriangle id)|\!) \ .$$

Finally, we have arrived at an efficient formulation, which is in fact equivalent to the definition of histomorphisms by Uustalu and Vene [14], though through a much shorter proof. This final solution folds over the input in a single sweep with h, which returns the entire history in a context. Once this is done, the head of this context is extracted. Note also that this definition does not depend on laziness.

Histomorphisms in Haskell revisited The implementation of more efficient histomorphisms in Haskell translates easily from the categorical notation.

$$
\begin{array}{l}
histo_2 :: (Inductive\ \mathsf{F}) \Rightarrow (\mathsf{F}\ (\mathsf{F}_{\infty}\ b) \to b) \to (\mu\mathsf{F} \to b) \\
histo_2\ b = head_{\infty} \cdot (\!|cons_{\infty} \cdot (b \vartriangle id)|\!)
\end{array}
$$

This works by ensuring that the input value has an inductive type, and folds this value in a single sweep into a structure of type $\mathsf{F}_{\infty}\ b$, where the head contains the final solution.

5.3 Examples

Let us now apply the framework of histomorphisms to the examples presented in Section 2. As we have seen, histomorphisms can only be applied when the input types are initial algebras. This is certainly the case for $knapsack_1$, $catalan_1$, and $bitonic_1$ since the initial algebra in question is simply the natural numbers. On the other hand, $bitonic'_1$ and $chain_1$ cannot be expressed as histomorphisms, since the input to these functions, $(\mathbb{N}, \mathbb{N})$, is not an initial algebra.

As we shall see, there is one important modification that needs to be made when translating the specifications to this version of the algorithms: indices in both the specifications and the tabular versions are absolute in the sense that they are indexed from some fixed origin that is taken as a reference point. With histomorphisms, the reference point is the 'current' point of call in the recursion, and so indices that refer to subsolutions are relative.

Knapsack problem To express the knapsack problem as a histomorphism, we will need to consider the input parameter as an initial algebra. This is more easily seen when we specialize the recursive specification to expose the structure of natural numbers in c:

$$
\begin{array}{l}
knapsack_3 :: \mathbb{N} \to \mathbb{R} \\
knapsack_3\ 0 \qquad\quad = 0 \\
knapsack_3\ (c+1) = maximum_0 \\
\quad [v + knapsack_3\ (c - w) \mid (w+1, v) \leftarrow wvs, w \leqslant c] \ .
\end{array}
$$

Just as in the array-based version of $knapsack_2$, we will create an algebra that replaces the recursive call in the body of the algorithm with a lookup. This time, however, we will be looking up values in the cofree structure of the naturals, $\mathsf{Nat}_{\infty}\ v$, rather than an array.

To turn this into a histomorphism, we provide a context-sensitive algebra *knapsack* that uses the results of previous subsolutions found in the $\mathsf{Nat}_{\infty}\ v$ structure and returns the solution. This solution is then used by $histo_2$ which embeds this value at the top of the context that is used in the next round of the recursion.

$$
\begin{array}{l}
knapsack_4 :: \mathbb{N} \to \mathbb{R} \\
knapsack_4 = histo_2\ knapsack\ \mathbf{where} \\
\quad knapsack :: \mathsf{Nat}\ (\mathsf{Nat}_{\infty}\ \mathbb{R}) \to \mathbb{R} \\
\quad knapsack\ Zero \qquad\quad = 0 \\
\quad knapsack\ (Succ\ table) = maximum_0 \\
\qquad [v + u \mid (w+1, v) \leftarrow wvs, Just\ u \leftarrow [lookup_{\infty}\ table\ w]]
\end{array}
$$

When *knapsack* is called for the first time, the lookup table is *Zero* and contains no elements, so the result is simply 0. For each successive call, *knapsack* has access to previous computed values in *table*, one for each smaller knapsack capacity than the one currently under consideration.

Note that we have adjusted the indices of the lookup: in the original version, the recursive call is performed with $knapsack\ (c - w)$, which is the absolute position of the knapsack capacity minus the weight of a given item. In the histomorphism version, we replace the lookup with the value u, which is the result of a relative indexing, where $lookup_{\infty}\ t\ (c - (c - w)) = lookup_{\infty}\ t\ w$. This works out nicely because the 'current' capacity c is not available. Another difference is that the out-of-bounds guard $w \leqslant c$ has been replaced by $Just\ u \leftarrow [\ldots]$. The maximum is then calculated just as in the recursive version.

In order to find values in the $\mathsf{Nat}_{\infty}\ v$ structure, we introduce function $lookup_{\infty}$, which provides access to the results stored in the head:

$$
\begin{array}{lll}
lookup_{\infty} :: \mathsf{Nat}_{\infty}\ a \to \mathbb{N} \to Maybe\ a & & \\
lookup_{\infty}\ (Cons_{\infty}\ a\ \_) & 0 & = Just\ a \\
lookup_{\infty}\ (Cons_{\infty}\ a\ Zero) & (n+1) & = Nothing \\
lookup_{\infty}\ (Cons_{\infty}\ a\ (Succ\ as)) & (n+1) & = lookup_{\infty}\ as\ n \ .
\end{array}
$$

The effect of $lookup_{\infty}\ table\ n$ is to return the result that was computed n 'steps' before the current point of call, since more recent values are found at the head of a $Cons_{\infty}$ constructor.

Catalan numbers The generation of Catalan numbers proves to be an instructive example, since it is *not* expressible as a histomorphism. First notice that each successive value makes use of *all* of the subsolutions: this can be seen in the definition of $catalan_2$, where computing the value of $catalan_2\ (n+1)$ must access all of the values with indices in the range $[0\,..\,n]$. It may be tempting to admit the following *bogus* definition as a histomorphism, where a Catalan number is simply the result of the convolution, that is, summing the multiplication of a list of prior elements with its reversal.

$$
\begin{array}{l}
catalan_3 :: \mathbb{N} \to \mathbb{N} \qquad\qquad \text{-- BOGUS!} \\
catalan_3 = histo_2\ catalan\ \mathbf{where} \\
\quad catalan :: \mathsf{Nat}\ (\mathsf{Nat}_{\infty}\ \mathbb{N}) \to \mathbb{N} \\
\quad catalan\ Zero \qquad\quad = 1 \\
\quad catalan\ (Succ\ table) = sum\ (zipWith\ (*)\ xs\ (reverse\ xs)) \\
\qquad \mathbf{where}\ xs = elems_{\infty}\ table
\end{array}
$$

At each step of the histomorphism the table in scope contains only the solutions to subproblems, so instead of pulling out values from the table using an index, we can instead select all of the elements at once, returning a list of all the previously computed solutions. So, why is this function not well-defined?

The function at fault is $elems_{\infty}$, that extracts all of the values:

$$
\begin{array}{ll}
elems_{\infty} :: \mathsf{Nat}_{\infty}\ v \to [v] & \text{-- BOGUS!} \\
elems_{\infty}\ (Cons_{\infty}\ a\ Zero) & = [a] \\
elems_{\infty}\ (Cons_{\infty}\ a\ (Succ\ as)) & = a : elems_{\infty}\ as \ .
\end{array}
$$

This definition is bogus because we are converting the coinductive $\mathsf{Nat}_{\infty}\ v$ to a simple inductive list. Of course, this does not work in general. We shall revisit the Catalan numbers later, and show how they can be expressed as a dynamorphism.

The bitonic travelling-salesman problem The specification expressed by $bitonic_1$ is another example where histomorphisms cannot be used in an algorithm where the input type is an initial algebra. The closest we can get is the following definition, which follows directly from the specification. As before, the lookup is performed on a table with a relative index.

$$\begin{array}{l}
bitonic_2 :: \mathbb{N} \to \mathbb{R} \qquad\qquad \text{-- BOGUS!} \\
bitonic_2 = histo_2\ bitonic\ \mathbf{where} \\
\quad bitonic :: \mathsf{Nat}\ (\mathsf{Nat}_\infty\ \mathbb{R}) \to \mathbb{R} \\
\quad bitonic\ Zero \qquad\quad\;\; = 2 * \overline{p_0\, p_1} \\
\quad bitonic\ (Succ\ table) = minimum\ [u - \overline{p_k\, p_{k+1}} + \overline{p_k\, p_{n+2}} \\
\qquad + sum\ [\overline{p_i\, p_{i+1}} \mid i \leftarrow [k+1\,..\,n+1]] \mid k \leftarrow [0\,..\,n], \\
\qquad\quad Just\ u \leftarrow [lookup_\infty\ table\ (n-k)]] \\
\qquad \mathbf{where}\ n = length_\infty\ table
\end{array}$$

Since the input parameter n is not present, this definition makes use of the function $length_\infty$, but this is unfortunately bogus:

$$\begin{array}{l}
length_\infty :: \mathsf{Nat}_\infty\ v \to Int \qquad\qquad \text{-- BOGUS!} \\
length_\infty\ (Cons_\infty\ a\ Zero) \qquad\quad = 0 \\
length_\infty\ (Cons_\infty\ a\ (Succ\ as)) = 1 + length_\infty\ as \enspace .
\end{array}$$

Again, we are handling values with coinductive types incorrectly: this function has no solution when the input is infinite.

The heart of the problem with these examples lies in the fact that we need access to the original parameter to the function, n, but this is not available to us. Our attempts to recover this value resort to using bogus functions that are not well-defined when the input is infinite. We will revisit this problem after we introduce dynamorphisms.

6. Dynamorphisms

Histomorphisms insist that the input is an element of some initial algebra. Looking back at Section 2 we note that this is the case for some but not all of the examples: $chain_1$ and $bitonic'_1$, for instance, take a pair of natural numbers as input. For these examples dynamorphisms come to the rescue. The basic idea is simple, but far reaching:

When we implemented histomorphisms

$$x \cdot in = a \cdot \mathsf{F}\ (\mathsf{F}_\infty\ x \cdot \llbracket in^\circ \rrbracket) \enspace ,$$

we made most of the fact that in has an inverse, turning the algebra on the left into a coalgebra on the right:

$$x = a \cdot \mathsf{F}\ (\mathsf{F}_\infty\ x \cdot \llbracket in^\circ \rrbracket) \cdot in^\circ \enspace .$$

The idea of dynamorphisms is to replace in° by a so-called *recursive* coalgebra c.

$$x = a \cdot \mathsf{F}\ (\mathsf{F}_\infty\ x \cdot \llbracket c \rrbracket) \cdot c \enspace . \tag{17}$$

Loosely speaking, recursiveness guarantees that the equation still has a *unique* solution. We shall say more about recursive coalgebras in Section 7.1.

Remark. Dynamorphisms were originally introduced in the setting of partial orders and continuous functions with no restriction on the coalgebra c [10]. Under these assumptions (17) has only a canonical solution, not a unique solution. We do not wish to go down this route. □

As with histomorphisms, it is useful to abstract away from the cofree comonad F_∞ and develop the recursion scheme in a more general setting. Before we do so, we record an implementation of this (inefficient) version of dynamorphisms in Haskell.

Dynamorphisms in Haskell The implementation is

$$\begin{array}{l}
dyna_1 :: (Functor\ \mathsf{F}) \Rightarrow (\mathsf{F}\ (\mathsf{F}_\infty\ a) \to a) \to (c \to \mathsf{F}\ c) \to (c \to a) \\
dyna_1\ a\ c = x\ \mathbf{where}\ x = a \cdot fmap\ (fmap\ x \cdot \llbracket c \rrbracket) \cdot c \enspace .
\end{array}$$

As with the definition of $histo_1$, this is not efficient, since the intermediary structure is built in exponential time.

7. Recursion Schemes From Recursive Coalgebras

Histomorphisms combine an algebra and a coalgebra. For dynamorphisms we have transmogrified the algebra into a coalgebra. Thus, dynamorphisms combine two coalgebras, of which one is a coalgebra for a comonad. As before, the two ingredients are related by a distributive law $\lambda : \mathsf{F} \circ \mathsf{F}_\infty \dot{\to} \mathsf{F}_\infty \circ \mathsf{F}$.

$$\begin{tikzcd}
\mathsf{F}\ C \arrow[dr, "\mathsf{F}\llbracket c \rrbracket"] & \\
& \mathsf{F}\ (\mathsf{F}_\infty\ C) \arrow[dd, "\lambda\ C"] \\
C \arrow[uu, "c"] \arrow[dd, "\llbracket c \rrbracket"'] & \\
& \mathsf{F}_\infty\ (\mathsf{F}\ C) \\
\mathsf{F}_\infty\ C \arrow[ur, "\mathsf{F}_\infty\ c"'] &
\end{tikzcd}$$

This is essentially the same diagram as in (5), only that the arrows previously labelled with in and $\mathsf{F}_\infty\ in$ have been flipped. There is no established name for the resulting structure. Capretta et al. [3] have coined the combination of two coalgebras a λ-*dicoalgebra*. We adopt the terminology, even though this is likely to cause confusion (there are also dialgebras, which are entirely different beasts).

As usual, we postpone a formal introduction of the scheme after we have provided the necessary background.

7.1 Background

Hylomorphisms and recursive coalgebras A hylomorphism (or algebra-from-coalgebra homomorphism) is a recursion scheme that captures the essence of *divide-and-conquer* algorithms. Such algorithms have three phases: first, a problem is broken into sub-problems by a coalgebra $c : C \to \mathsf{F}\ C$; second, sub-problems are recursively turned into sub-solutions; and finally, sub-solutions are combined by an algebra $a : \mathsf{F}\ A \to A$ to form a solution. An arrow $h : C \to A$ is a *hylomorphism*, $h : (C, c) \to (a, A)$, if it satisfies

$$h = a \cdot \mathsf{F}\ h \cdot c \enspace . \tag{18}$$

A coalgebra (C, c) is *recursive* (or algebra-initial) if for every algebra (a, A) there is a unique hylomorphism $(C, c) \to (a, A)$ satisfying (18). An important recursive coalgebra is $(\mu\mathsf{F}, in^\circ)$, which is also the final recursive coalgebra. Thus, using recursive coalgebras allows us to generalize the development of histomorphisms, which are a special case of dynamorphisms where the coalgebra is in°.

Dicoalgebras Let $\lambda : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$ be a distributive law for the endofunctor F over the comonad N. A λ-*dicoalgebra* (X, c, d) consists of an F-coalgebra c and a coalgebra d for the comonad N such that the *pentagonal law* holds:

$$\mathsf{N}\ c \cdot d = \lambda\ X \cdot \mathsf{F}\ d \cdot c \enspace . \tag{19}$$

The pentagonal law (19) also has an asymmetric rendering, which identifies the coalgebra c as an N-coalgebra homomorphisms.

$$\begin{tikzcd}
\mathsf{F}\ X \arrow[dr, "\mathsf{F}\ d"] & & & \\
& \mathsf{F}\ (\mathsf{N}\ X) \arrow[dd, "\lambda\ X"] & X \arrow[r, "c"] \arrow[d, "d"'] & \mathsf{F}\ X \arrow[d, "\lambda\ X \cdot \mathsf{F}\ d"] \\
X \arrow[uu, "c"] \arrow[dd, "d"'] & & \mathsf{N}\ X \arrow[r, "\mathsf{N}\ c"'] & \mathsf{N}\ (\mathsf{F}\ X) \\
& \mathsf{N}\ (\mathsf{F}\ X) & & \\
\mathsf{N}\ X \arrow[ur, "\mathsf{N}\ c"'] & & &
\end{tikzcd} \tag{20}$$

7.2 Recursion Schemes From Recursive Coalgebras

Let $\lambda : \mathsf{F} \circ \mathsf{N} \dot{\to} \mathsf{N} \circ \mathsf{F}$ be a distributive law, and let (C, c, d) be a λ-dicoalgebra where $c : C \to \mathsf{F}\ C$ is recursive. For any $(\mathsf{F} \circ \mathsf{N})$-algebra (a, A) there is a unique arrow $f : C \to A$ such that

$$f = a \cdot \mathsf{F}\ (\mathsf{N} f \cdot d) \cdot c\ . \qquad (21)$$

Quite amazingly, everything we said about histomorphisms and recursion schemes from comonads generalizes to this more expressive setting. In particular, there is a 1-1 correspondence between two kinds of hylomorphisms:

$$f = a \cdot \mathsf{F}\ h \cdot c \iff h = \lfloor a \rfloor \cdot \mathsf{F}\ h \cdot c\ ,$$

where $h = \lfloor f \rfloor = \mathsf{N} f \cdot d$ is the transpose of f. Since the coalgebra c is recursive, the equation on the right has a unique solution and hence the original equation (21), as shown by Capretta et al. [3].

Dynamorphisms revisited To show that dynamorphisms are an instance of this scheme, we have to prove that c and $\llparenthesis c \rrparenthesis$ form a λ-dicoalgebra, where $\lambda : \mathsf{F} \circ \mathsf{F}_\infty \dot{\to} \mathsf{F}_\infty \circ \mathsf{F}$. Like for bialgebras, there is a 1-1 correspondence between id-dicoalgebras and λ-dicoalgebras.

$$\begin{tikzcd}
\mathsf{F}\ X \arrow[dr, "\mathsf{F}\ g"] & & & \mathsf{F}\ X \arrow[dr, "\mathsf{F}\ \llparenthesis g \rrparenthesis"] & & \\
& \mathsf{F}\ (\mathsf{F}\ X) \arrow[dd, "id\ X"] & & & \mathsf{F}\ (\mathsf{F}_\infty\ X) \arrow[dd, "\lambda\ X"] & \\
X \arrow[uu, "f"] \arrow[dd, "g"'] & & \iff & X \arrow[uu, "f"] \arrow[dd, "\llparenthesis g \rrparenthesis"'] & & (22) \\
& \mathsf{F}\ (\mathsf{F}\ X) & & & \mathsf{F}_\infty\ (\mathsf{F}\ X) & \\
\mathsf{F}\ X \arrow[ur, "\mathsf{F} f"'] & & & \mathsf{F}_\infty\ X \arrow[ur, "\mathsf{F}_\infty f"'] & &
\end{tikzcd}$$

Using this property, the proof that $(C, c, \llparenthesis c \rrparenthesis)$ forms a λ-dicoalgebra is a one-liner.

$$\begin{aligned}
& \mathsf{F}_\infty\ c \cdot \llparenthesis c \rrparenthesis = \lambda\ X \cdot \mathsf{F}\ \llparenthesis c \rrparenthesis \cdot c \\
\iff\ & \{\ (22)\ \} \\
& \mathsf{F}\ c \cdot c = id\ X \cdot \mathsf{F}\ c \cdot c
\end{aligned}$$

Thus dynamorphisms are indeed an instance of the scheme above. In addition, histomorphisms are also an instance, since they are simply the case where we specialize further and instantiate $C := \mu\mathsf{F}$ and $c := in^\circ$.

Efficiency improvements First of all, note that $(\lfloor a \rfloor, \mathsf{F}_\infty\ A, \delta\ A)$ still forms a λ-bialgebra and that h can be seen as an arrow from a λ-dicoalgebra to a λ-bialgebra, see the diagram on the right below.

$$\begin{tikzcd}
\mathsf{F}\ C \arrow[r, "\mathsf{F}\ h"] & \mathsf{F}\ (\mathsf{F}_\infty\ A) \arrow[d, "\lfloor a \rfloor"] & & \mathsf{F}\ C \arrow[r, "\mathsf{F}\ h"] & \mathsf{F}\ (\mathsf{F}_\infty\ A) \arrow[d, "\lfloor a \rfloor"] \\
C \arrow[u, "c"] \arrow[r, "h"'] \arrow[d, "c"'] & \mathsf{F}_\infty\ A \arrow[d, "tail_\infty\ A"] & \iff & C \arrow[u, "c"] \arrow[r, "h"'] \arrow[d, "\llparenthesis c \rrparenthesis"'] & \mathsf{F}_\infty\ A \arrow[d, "\llparenthesis tail_\infty\ A \rrparenthesis"] \\
\mathsf{F}\ C \arrow[r, "\mathsf{F}\ h"'] & \mathsf{F}\ (\mathsf{F}_\infty\ A) & & \mathsf{F}_\infty\ C \arrow[r, "\mathsf{F}_\infty\ h"'] & \mathsf{F}_\infty\ (\mathsf{F}_\infty\ A)
\end{tikzcd}$$

Again, we leverage the fact that F-coalgebra homomorphisms are in 1-1 correspondence to F_∞-coalgebra homomorphisms (10). The derivation of an efficient implementation of h follows the template laid out in Section 5.2—the proof below is even somewhat simpler, even though it establishes a more general result.

$$\begin{aligned}
& h = \lfloor a \rfloor \cdot \mathsf{F}\ h \cdot c \ \wedge\ tail_\infty \cdot h = \mathsf{F}\ h \cdot c \\
\implies\ & \{\ \text{Leibniz}\ \} \\
& head_\infty \cdot h = head_\infty \cdot \lfloor a \rfloor \cdot \mathsf{F}\ h \cdot c \ \wedge\ tail_\infty \cdot h = \mathsf{F}\ h \cdot c \\
\iff\ & \{\ head_\infty \cdot \lfloor a \rfloor = a\ (9)\ \} \\
& head_\infty \cdot h = a \cdot \mathsf{F}\ h \cdot c \ \wedge\ tail_\infty \cdot h = \mathsf{F}\ h \cdot c \\
\iff\ & \{\ \text{products}\ \} \\
& head_\infty \cdot h \vartriangle tail_\infty \cdot h = a \cdot \mathsf{F}\ h \cdot c \vartriangle \mathsf{F}\ h \cdot c \\
\iff\ & \{\ \text{fusion}\ \} \\
& (head_\infty \vartriangle tail_\infty) \cdot h = (a \vartriangle id) \cdot \mathsf{F}\ h \cdot c \\
\iff\ & \{\ head_\infty \vartriangle tail_\infty \text{ isomorphism}\ \} \\
& h = cons_\infty \cdot (a \vartriangle id) \cdot \mathsf{F}\ h \cdot c
\end{aligned}$$

Since c is a recursive coalgebra, the last equation has a unique solution.

Dynamorphisms in Haskell revisited The translation of this categorical machinery into Haskell is entirely straightforward:

```
dyna₂ :: (Functor F) ⇒ (F (F∞ a) → a) → (c → F c) → (c → a)
dyna₂ a c = head∞ · h where h = cons∞ · (a △ id) · fmap h · c  .
```

7.3 Examples

Dynamorphisms work by constructing an intermediate structure with a coalgebra that stores the history of all subresults. One way to interpret this is that the intermediate structure holds a call stack of previous values that can be referenced. While more elaborate functors are supported by the scheme, using a linear structure—such as lists—is a particularly versatile option since we can flatten more complex structures by providing a specific traversal. The base functor for polymorphic nonempty lists has two constructors: one for when there is a single element, and the other to add elements to the list.

```
data List v x = Some v | Cons v x
instance Functor (List v) where
  fmap f (Some v)   = Some v
  fmap f (Cons v x) = Cons v (f x)
```

To query values from this structure, we develop a number of operations. First, we provide an indexing operator that takes a natural number and returns the corresponding value, working back through the hierarchy the given number of times. The interface is very similar to the indexing operator for standard lists.

```
(!!∞) :: (Show a, Show v) ⇒ (List v)∞ a → ℕ → a
(Cons∞ a _)              !!∞ 0       = a
(Cons∞ a (Cons v as)) !!∞ (n + 1) = as !!∞ n
```

Of course we could have defined this to be a total function, returning a value of type *Maybe a* in case the indexing is out of bounds. For our purposes, we use this operator since it reduces clutter in the code that follows.

When more than one value is required at once, and assuming they appear in a contiguous section, it is convenient to make use of the function $take_\infty\ n$, which takes n consecutive values from the hierarchy.

```
take∞ :: ℕ → (List v)∞ a → [a]
take∞ 0         _                      = []
take∞ (n + 1) (Cons∞ a (Some v))     = [a]
take∞ (n + 1) (Cons∞ a (Cons v as)) = a : take∞ n as
```

Note that it might be tempting to define a related operator, $drop_\infty :: \mathbb{N} \to (\mathsf{List}\ v)_\infty\ a \to [a]$, which drops a given number of values from the hierarchy before returning the values that remain. However, such a definition would be bogus, since we cannot validly use induction over values of type $(\mathsf{List}\ v)_\infty\ v$. The definition of $take_\infty$, however, is perfectly valid, since are using induction over the natural numbers.

With these basic ingredients in place, we are now ready to define some dynamorphisms.

Catalan numbers revisited We now revisit the Catalan numbers, and show how they can be defined in terms of a dynamorphism. The problem we had previously was that there was no means of knowing where in the recursion a call was being made. To store this information, we can define the coalgebra *natural* as follows:

$$\begin{array}{l}
natural :: \mathbb{N} \to (\mathsf{List}\ \mathbb{N}\ \mathbb{N}) \\
natural\ 0 \qquad\quad = Some\ 0 \\
natural\ (n+1) = Cons\ (n+1)\ n \ \ .
\end{array}$$

To validly apply this in a dynamorphism, we must argue that it is a recursive coalgebra. This amounts to showing that when applied recursively this has the halting property [1]. In this case we observe that in the recursive case, the value $n+1$ is reduced by 1 at each step.

Using this coalgebra, we can form the following definition:

$$\begin{array}{l}
catalan_4 :: \mathbb{N} \to \mathbb{N} \\
catalan_4 = dyna_2\ catalan\ natural\ \mathbf{where} \\
\quad catalan :: \mathsf{List}\ \mathbb{N}\ ((\mathsf{List}\ \mathbb{N})_\infty\ \mathbb{N}) \to \mathbb{N} \\
\quad catalan\ (Some\ 0) \qquad\quad = 1 \\
\quad catalan\ (Cons\ n\ table) = sum\ (zipWith\ (*)\ xs\ (reverse\ xs)) \\
\qquad \mathbf{where}\ xs = take_\infty\ n\ table \ \ .
\end{array}$$

The key here is that the algebra *catalan* knows about the current depth of its application, which is held in n. Thus, the appropriate number of values can be extracted from the table of previous values, and convoluted just as in previous definitions.

The bitonic travelling-salesman problem revisited Clearly, using the same technique as in the definition of $catalan_4$, we can encode the parameter n as a parameter. From here, producing a correct version with a few modifications to $bitonic_2$ is trivial.

For variety, we now discuss a solution that reflects the algorithm described by $bitonic'_1$. First we must consider which coalgebra and algebra should be used. The carrier for both of these is already determined by the type of the recursion, and must be $(\mathbb{N},\mathbb{N})$. What remains to be decided is the base functor for this computation. A crucial part of the solution involves looking up distances between points, and so it is important to keep track of the current point that is being processed during the recursion steps. Therefore, to store these values, we use a base functor of type $\mathsf{List}\ (\mathbb{N},\mathbb{N})$.

Let us consider the coalgebra that constructs the lookup structure. One of the conditions we can impose is that $i \leqslant j$, since this formed part of our invariant. With this in consideration, it makes sense that only a triangle of values needs to be computed.

$$\begin{array}{l}
triangle_1 :: (\mathbb{N},\mathbb{N}) \to \mathsf{List}\ (\mathbb{N},\mathbb{N})\ (\mathbb{N},\mathbb{N}) \\
triangle_1\ (0,0) = Some\ (0,0) \\
triangle_1\ (0,1) = Some\ (0,1) \\
triangle_1\ (0,j)\ \ = Cons\ (0,j)\ (j-1,j-1) \\
triangle_1\ (i,j)\ \ = Cons\ (i,j)\ (i-1,j)
\end{array}$$

To argue that this coalgebra is recursive, we observe that the reverse lexicographic ordering of the pair (i,j) always decreases, where the relationship with a new pair (i',j') is given by $(i,j) > (i',j') \iff j > j' \lor (j == j' \land i > i')$. This strategy of building a triangle stores the coordinates in scope as it goes, since this information becomes vital when applying the algebra. Note that here we have included two

$$\begin{tikzcd}
(0,0) \\
(0,1) \arrow[r] & (1,1) \arrow[u, bend left, to=1-1] \\
(0,2) \arrow[r] & (1,2) \arrow[r] & (2,2) \arrow[to=2-1, bend left] \\
(0,3) \arrow[r] & (1,3) \arrow[r] & (2,3) \arrow[r] & (3,3) \arrow[to=3-1, bend left]
\end{tikzcd}$$

Figure 2. Coordinate ordering given by $triangle_2$.

base cases, both of which follow from the recursive definition of the algorithm.

$$\begin{array}{l}
bitonic'_2 :: (\mathbb{N},\mathbb{N}) \to \mathbb{R} \\
bitonic'_2 = dyna_2\ bitonic\ triangle_1\ \mathbf{where} \\
\quad bitonic :: \mathsf{List}\ (\mathbb{N},\mathbb{N})\ ((\mathsf{List}\ (\mathbb{N},\mathbb{N}))_\infty\ \mathbb{R}) \to \mathbb{R} \\
\quad bitonic\ (Some\ (0,0)) = 0 \\
\quad bitonic\ (Some\ (0,1)) = \overline{p_0\ p_1} \\
\quad bitonic\ (Cons\ (i,j)\ table) \\
\qquad \mid i < j-1 = table\ !!_\infty\ (j-1) + \overline{p_{j-1}\ p_j} \\
\qquad \mid i == j-1 = minimum \\
\qquad\quad [table\ !!_\infty\ (k+j) + \overline{p_{i-k-1}\ p_j} \mid k \leftarrow [0\,..\,i-1]] \\
\qquad \mid i == j \qquad = minimum \\
\qquad\quad [table\ !!_\infty\ k + \overline{p_{i-k-1}\ p_j} \mid k \leftarrow [0\,..\,i-1]]
\end{array}$$

While this definition has some similarities to the specification in $bitonic'_1$, the indices are clearly quite different. As with histomorphisms, this is because in this definition we no longer have the ability to make references to previously computed values by using absolute indices: all the indexing into the t structure is relative to the point of call. This is why the '*otherwise*' clause from the recursive definition has been split into two different cases.

Chain matrix multiplication The chain matrix multiplication problem also involves computing a triangle of values, since in the definition of $chain_1\ i\ j$ we have the invariant that $i \leqslant j$. However, the order in which this triangle is built is different to the definition of $triangle_1$. For one, we must ensure that $i \geqslant 0$, and furthermore, the access pattern for values in the triangle is somewhat different, since values are instead built from the diagonal to an edge.

$$\begin{array}{l}
triangle_2 :: (\mathbb{N},\mathbb{N}) \to \mathsf{List}\ (\mathbb{N},\mathbb{N})\ (\mathbb{N},\mathbb{N}) \\
triangle_2\ (0,0) = Some\ (0,0) \\
triangle_2\ (i,j) \\
\quad \mid i == j \qquad\quad = Cons\ (i,j)\ (0,j-1) \\
\quad \mid otherwise\ \ = Cons\ (i,j)\ (i+1,j)
\end{array}$$

Again, we must argue that this coalgebra is recursive, which is only true when $i \leqslant j$. If $i < j$ then i is increased until $i == j$. When this is the case, i is set to 0 and j is decreased by 1. Thus at any point in the recursion, over the course of at most j steps the value will be reduced to $(0,j-1)$, and eventually this terminates.

The definition of the algebra of $chain_3$ requires particular attention to the relative indices: the base case is straightforward, but when $i < j$ we must calculate the offset carefully.

$$\begin{array}{l}
chain_3 :: (\mathbb{N},\mathbb{N}) \to \mathbb{N} \\
chain_3 = dyna_2\ chain\ triangle_2\ \mathbf{where} \\
\quad chain :: \mathsf{List}\ (\mathbb{N},\mathbb{N})\ ((\mathsf{List}\ (\mathbb{N},\mathbb{N}))_\infty\ \mathbb{N}) \to \mathbb{N} \\
\quad chain\ (Some\ \_) = 0 \\
\quad chain\ (Cons\ (i,j)\ table) \\
\qquad \mid i == j = 0 \\
\qquad \mid i < j = minimum\ (zipWith\ (+)\ [a_i * a_{k+1} * a_{j+1}\ + \\
\qquad\quad table\ !!_\infty\ offset\ k \mid k \leftarrow [i\,..\,j-1]]\ (take_\infty\ (j-i)\ table)) \\
\qquad \mathbf{where}\ offset\ k = ((j*(j+1) - k*(k+1))\ `div`\ 2) - k + j - 1
\end{array}$$

To understand this definition, we first note that a cell with index (i,j) is dependent on all the cells that are 'above' and to the 'right', relative to the ordering imposed in $triangle_2$, as depicted in Figure 2.

Taking values from the right is simple, since these are in order, and can be extracted directly with the $take_\infty$ function. However, indices of values that are picked from an offset above the current node require careful calculation. The definition of *offset k* arises as a consequence of the triangle numbers $T(n) = \sum_{x=1}^{n} x = n(n+1)/2$, where we subtract one triangle number from the other.

8. Related Work

Histo- and dynamorphisms The work we have presented builds on the foundations that were set out in the original paper on histomorphisms [14], where course-of-values iteration was captured as a categorically-inspired recursion scheme. This work was later extended to include dynamorphisms [10], with the specific goal of extending the reach of histomorphisms to cover dynamic programming algorithms. The authors there also present a number of classic dynamic programming algorithms are given in terms of this framework, and the derivation of efficient dynamorphisms rests on the connection with hylomorphisms. The constructions presented there differ in that they are all within CPO, where initial algebras and final coalgebras coincide.

Recursion schemes from comonads The construction of recursion schemes from comonads was first presented by Uustalu et al. [15], where the relationship with histomorphisms is explored in detail. That paper also provides an implementation of various recursion schemes in Haskell although it does not make use of type class synonyms, since it predates that work. The correspondence between histomorphisms and recursion schemes from comonads is a direct application of the unification of structured recursion schemes [8], which further explores material on the relationship between recursion schemes from comonads and the adjoint folds [6].

Recursion schemes from recursive coalgebras The notion of obtaining uniqueness properties through recursive coalgebras comes directly from the seminal paper on the topic [3], and a more gentle introduction to recursive coalgebras can be found in the work of Hinze et al. [7].

9. Conclusion

In this paper we have demonstrated the use of histomorphisms and dynamorphisms through a number of examples, and have shown how these categorically-inspired recursion schemes can be implemented efficiently. The derivation of the efficient versions of histomorphisms relies on their formulation as recursion schemes from comonads and a correspondence between certain bialgebras. Similarly, the derivation of efficient dynamorphisms relies on their formulation as recursion schemes from recursive coalgebras and a correspondence between certain dicoalgebras. In both cases, the Eilenberg-Moore adjunction is at the heart of the development.

Future work There are a number of avenues for future work. One aspect of dynamorphisms which we have not discussed is the choice of base functor. In this paper, we linearized all structures, and this has required us to pay particular attention to the relative indexing schemes. Another option worth exploring are using a more direct approach such as arrays with a focus. A more structured approach would be to change the base functor to one that both deals with sharing, and that also maintains the structure of the recursion. A similar strategy was employed to turn hylomorphisms with nexuses into dynamic algorithms [2].

Recursive coalgebras are modular in the sense that they can be combined to form even more expressive schemes. For example, it will be convenient to also avail the algebra to the original argument of the function: this arises from considering parametrically recursive coalgebras, which we do not explore here.

Acknowledgments

This work has been funded by EPSRC grant number EP/J010995/1, on Unifying Theories of Generic Programming. The authors would like to thank the anonymous reviewers for their helpful and constructive comments. In particular, they identified areas where further explanation of our algorithms was needed, and prompted us to add more diagrams to clarify the material.

References

[1] J. Adámek, D. Lücke, and S. Milius, “Recursive coalgebras of finitary functors,” *Theoret. Informatics Appl.*, vol. 41, no. 4, pp. 447–462, 2007. doi:10.1051/ita:2007028

[2] R. Bird, *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

[3] V. Capretta, T. Uustalu, and V. Vene, “Recursive coalgebras from comonads,” *Information and Computation*, vol. 204, no. 4, pp. 437–468, 2006. doi:10.1016/j.ic.2005.08.005

[4] M. M. T. Chakravarty, G. Keller, and S. P. Jones, “Associated type synonyms,” in *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’05. ACM, 2005, pp. 241–253. doi:10.1145/1086365.1086397

[5] S. Eilenberg and J. C. Moore, “Adjoint functors and triples,” *Illinois J. Math*, vol. 9, no. 3, pp. 381–398, 1965.

[6] R. Hinze, “Adjoint folds and unfolds—an extended study,” *Science of Computer Programming*, Aug. 2012. doi:10.1016/j.scico.2012.07.011

[7] R. Hinze, D. W. James, and T. Harper, “Theory and practice of fusion,” in *Proceedings of the 22nd Symposium on the Implementation and Application of Functional Languages (IFL ’10)*, ser. Lecture Notes in Computer Science, vol. 6647. Springer Berlin / Heidelberg, Sep. 2011, pp. 19–37. doi:10.1007/978-3-642-24276-2_2

[8] R. Hinze, N. Wu, and J. Gibbons, “Unifying structured recursion schemes,” in *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP ’13. ACM, 2013. doi:10.1145/2500365.2500578

[9] P. J. Huber, “Homotopy theory in general categories,” *Mathematische Annalen*, vol. 144, pp. 361–385, 1961. doi:10.1007/BF01396534

[10] J. Kabanov and V. Vene, “Recursion schemes for dynamic programming,” in *Mathematics of Program Construction, 8th International Conference, MPC 2006*. Springer Berlin / Heidelberg, 2006, pp. 235–252. doi:10.1007/11783596_15

[11] H. Kleisli, “Every standard construction is induced by a pair of adjoint functors,” *Proceedings of the American Mathematical Society*, vol. 16, no. 3, pp. 544–546, Jun. 1965. doi:10.1090/S0002-9939-1965-0177024-4

[12] J. Lambek, “A fixpoint theorem for complete categories,” *Math. Zeitschr.*, vol. 103, pp. 151–161, 1968. doi:10.1007/BF01110627

[13] S. Peyton Jones, *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[14] T. Uustalu and V. Vene, “Primitive (co)recursion and course-of-value (co)iteration, categorically,” *Informatica, Lith. Acad. Sci.*, vol. 10, no. 1, pp. 5–26, 1999.

[15] T. Uustalu, V. Vene, and A. Pardo, “Recursion schemes from comonads,” *Nordic J. of Computing*, vol. 8, no. 3, pp. 366–390, Sep. 2001.