Conjugate Hylomorphisms

Or: The Mother of All Structured Recursion Schemes

Ralf Hinze Nicolas Wu Jeremy Gibbons

Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, England

{ralf.hinze, nicolas.wu, jeremy.gibbons}@cs.ox.ac.uk

Abstract

The past decades have witnessed an extensive study of structured recursion schemes. A general scheme is the hylomorphism, which captures the essence of divide-and-conquer: a problem is broken into sub-problems by a coalgebra; sub-problems are solved recursively; the sub-solutions are combined by an algebra to form a solution. In this paper we develop a simple toolbox for assembling recursive coalgebras, which by definition ensure that their hylo equations have unique solutions, whatever the algebra. Our main tool is the conjugate rule, a generic rule parametrized by an adjunction and a conjugate pair of natural transformations. We show that many basic adjunctions induce useful recursion schemes. In fact, almost every structured recursion scheme seems to arise as an instance of the conjugate rule. Further, we adapt our toolbox to the more expressive setting of parametrically recursive coalgebras, where the original input is also passed to the algebra. The formal development is complemented by a series of worked-out examples in Haskell.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.2 [*Programming Languages*]: Language Classifications—applicative (functional) languages; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—algebraic approaches to semantics

General Terms Languages, Theory, Verification

Keywords recursion schemes, hylomorphisms, adjunctions

1. Introduction

Over forty years ago, Hoare [14] observed that "there are certain close analogies between the methods used for structuring data and the methods for structuring a program which processes that data". These days, the relationship between data structure and program structure is expressed in the form of structured recursion schemes, and are widely used in functional languages such Haskell.

Canonical examples are provided by *folds* (catamorphisms), which consume data structures, and *unfolds* (anamorphisms), which produce them, solutions h and k of the following recursion equations:

$$h \cdot in = a \cdot \mathsf{F}\, h \qquad\qquad out \cdot k = \mathsf{F}\, k \cdot c \ \ ,$$

which are shaped by a *base functor* F. Categorically, these recursion schemes arise from constructions in which *in* and *out* are *initial algebras* and *final coalgebras* respectively. Initiality and finality each correspond to both existence and uniqueness of solutions to the equations above, traditionally written $h = (\!|a|\!)$ and $k = [\!(c)\!]$.

Folds and unfolds have been thoroughly studied and are now well understood and widely used, but they are rather restrictive patterns of computation. Thus, there have been many generalizations, extensions, and combinations in an attempt to improve expressivity. Ultimately, most of these are variations on a single theme, the *hylomorphism* [23], which arises as a solution to the equation:

$$h = a \cdot \mathsf{F}\, h \cdot c \ \ .$$

This generalization comes with a catch: there is no guarantee that a solution exists for a hylo equation with a given coalgebra c and algebra a, let alone that such a solution is uniquely determined. A candidate solution is $(\!|a|\!) \cdot [\!(c)\!]$, but this only works if the carriers of initial algebras and final coalgebras coincide.

A number of workarounds to the catch have been proposed over the years. One could work in an algebraically compact category such as **Cpo** [23]; then the carriers of initial algebras and final coalgebras do coincide, but one has to relax uniqueness of solutions to ordering them (for example, picking the least solution). Similarly, one could work in a self-dual category such as **Rel** [4]; again, the carriers coincide, this time because final coalgebras are exactly the converses of initial algebras, but again one has to resort to least solutions and deal with inequalities rather than equalities.

Alternatively, rather than restricting the setting in such ways, we can instead constrain the programs under consideration. In particular, we can restrict ourselves to so-called recursive coalgebras [6]—those that by definition yield unique solutions to the hylo equation, whatever the algebra. Dually, we can work with corecursive algebras—those that yield unique solutions whatever the coalgebra. This is a trade-off: only certain coalgebras (or dually, algebras) are admissible, but in return there need be no restriction on the setting, and no resort to inequalities rather than equalities.

The cost side of the trade-off is mitigated by providing a toolbox of techniques for assembling the admissible components, rather than requiring their admissibility to be proved from first principles. For example, because an initial algebra *in* is an isomorphism, it has an inverse in°, and this inverse turns out to be a recursive coalgebra. Hylomorphisms with $c = in^\circ$ are just folds, so by itself this is not very interesting; it becomes more interesting when we develop a powerful theory of constructions on recursive coalgebras, which is what we set out to do. Our theory is based on adjunctions, and, of course, it all dualizes nicely to corecursive algebras.

Perhaps surprisingly, the entire theory is based on only two tools, which we call the *rolling rule* and the *conjugate rule*. The former applies when the base functor shaping the recursion is composed from two other functors, and allows the two functors to be swapped.

The latter allows us to shunt functors between the producer and consumer parts of the hylo. The conjugate rule is parametrized by an adjunction and a conjugate pair of natural transformations. In a sense, it can be considered the *mother of all recursion schemes*, as almost every recursion scheme can be framed as an instance of this rule: folds and unfolds [10, 21, 23], folds with parameters [26], mutu- and zygomorphisms [8, 20], histo- and futumorphisms [29], generalized folds [5], recursion schemes from comonads [31], and adjoint folds [11].

This is not the first time this topic has been visited, and indeed most of the results in Section 3 appear in Capretta et al. [6], albeit using different machinery—comonads and distributive laws—and with substantially longer proofs. We demonstrate that each of their comonadic schemes also arises by instantiating the conjugate rule to an adjunction that generates the comonad. Indeed, in many ways the present work extends the unification of structured recursion schemes presented in [13], which showed that recursion schemes from comonads [31] are subsumed by adjoint folds [11]. Rather pleasingly, the generalization from initial algebras to recursive coalgebras streamlines many of the proofs. A distinct advantage of adjunctions over comonads is that the former compose nicely: we can build advanced recursion schemes out of simple ones by composing the underlying adjunctions (Section 3.5).

Folds are equivalent to paramorphisms [22], the Squiggol rendering of primitive recursion, where the input is also passed to the algebra. However, when we generalize initial algebras to recursive coalgebras this equivalence no longer holds. Practical considerations suggest that providing the inputs is useful, and this motivates us to extend the theory to include so-called parametrically recursive coalgebras. (There is also a dual variation [24] corresponding to apomorphisms [32], which allow the coalgebra to stop the recursion early.) Here is where we feel the benefit of a unified approach: we show that the rolling and the conjugate rule can be generalized to this more expressive setting, generalizing all of their instances in one go. (There is only one exception; see Section 5.6.)

In summary, this paper makes the following novel contributions:

- we unify and substantially simplify prior work on structured recursion schemes;
- we identify two central adjunction-like techniques for proving the uniqueness of a wide range of recursion equations;
- we generalize these techniques to the more expressive setting of parametrically recursive coalgebras.

The rest of the paper is structured as follows. Section 2 provides some background category theory, with an overview of important adjunctions. In Section 3 we discuss hylomorphisms, and introduce the two cornerstones of our theory: the rolling rule and the conjugate rule. We provide all of the proofs, as they serve as templates for the development in Section 5. We support our claim that almost all known structured recursion schemes can be expressed as instances of the conjugate rule. We illustrate the schemes through a number of Haskell examples as we go. More elaborate examples based on dynamic programming are given in Section 4. We then extend the theory in Section 5 to cover parametrically recursive coalgebras; in particular, we adapt the rolling and the conjugate rule to the new setting. We follow this with further examples in Section 6. Finally, Section 7 discusses related work, and Section 8 concludes.

2. Background

This paper assumes a basic knowledge of category theory: the reader should be familiar with the notions of functors and natural transformations. In this section we fix the notation and establish the concepts that will be used in the remainder of the paper. For the most part this material is standard, and can safely be glossed over on an initial reading, except perhaps the material on conjugates.

2.1 Algebras and Coalgebras

Algebras and coalgebras form the basis for the categorical description of structured recursion schemes.

Given an endofunctor $\mathsf{F} : \mathscr{C} \to \mathscr{C}$, a so-called *base functor*, 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 respectively as the *action* and *carrier* of the algebra. (We deviate a little from the standard notation (A,a), in order to have a syntax that distinguishes algebras from coalgebras.) Since the action determines its carrier, it is often used by itself to refer to the F-algebra. An F-homomorphism between algebras (a,A) and (b,B) is an arrow $h : A \to B : \mathscr{C}$ such that $h \cdot a = b \cdot \mathsf{F}\,h$.

$$\begin{array}{ccc} \mathsf{F}\,A & \xrightarrow{\mathsf{F}\,h} & \mathsf{F}\,B \\ {\scriptstyle a}\downarrow & & \downarrow{\scriptstyle b} \\ A & \xrightarrow[h]{} & B \end{array}$$

F-homomorphisms compose, and there is an identity, so F-algebras and F-homomorphisms form a category, which we call F-**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*. Initiality implies that to each F-algebra, (a,A), there corresponds 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 (in a certain sense, the least fixed-point); this fact is known as Lambek's Lemma [18].

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-**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 $[\![c]\!] : C \to \nu\mathsf{G}$.

Throughout, we use Haskell as a lingua franca for exemplifying categorical constructions. However, we are careful to distinguish between inductive and coinductive types, which Haskell conflates.

Example 2.1. The semantics of the inductive datatype *Tree* defined

$$\textbf{data}\ \mathit{Tree} = \mathit{Empty} \mid \mathit{Node}\ \mathit{Tree}\ \mathbb{Z}\ \mathit{Tree}$$

is given by the initial algebra $(in, \mu\mathsf{Tree})$, where the *base functor*

$$\textbf{data}\ \mathsf{Tree}\ \mathit{tree} = \mathsf{Empty} \mid \mathsf{Node}\ \mathit{tree}\ \mathbb{Z}\ \mathit{tree}$$

abstracts away from the recursive occurrences of *Tree*. The Haskell rendering of the isomorphism *in*, the action of the initial algebra, amounts to a simple renaming of constructors. □

The category F-**Alg**$(\mathscr{C})$ has more structure than $\mathscr{C}$. The forgetful or underlying functor $\mathsf{U}^{\mathsf{F}} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathscr{C}$ 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: $\mathsf{U}_{\mathsf{G}} : \mathsf{G}\text{-}\mathbf{Coalg}(\mathscr{C}) \to \mathscr{C}$.

Liftings and coliftings A functor $\overline{\mathsf{H}} : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathsf{G}\text{-}\mathbf{Alg}(\mathscr{D})$ is called a *lifting* of $\mathsf{H} : \mathscr{C} \to \mathscr{D}$ iff $\mathsf{H} \circ \mathsf{U}^{\mathsf{F}} = \mathsf{U}^{\mathsf{G}} \circ \overline{\mathsf{H}}$. For liftings, the action on the carrier and on homomorphisms is fixed; the action on the algebra can be specified using a natural transformation $\lambda : \mathsf{H} \circ \mathsf{F} \leftarrow \mathsf{G} \circ \mathsf{H}$, allowing us to define H^{λ} as a lifting:

$$\mathsf{H}^{\lambda}\ (a,A) = (\mathsf{H}\,a \cdot \lambda\ A, \mathsf{H}\,A) \qquad \mathsf{H}^{\lambda}\ h = \mathsf{H}\,h\ . \tag{2.1}$$

Since we use the action of an algebra to refer to the algebra itself, we often abbreviate $\mathsf{H}\,a \cdot \lambda\ A$ by $\mathsf{H}^{\lambda}\ a$.

Dually, $\underline{\mathsf{H}} : \mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C}) \to \mathsf{G}\text{-}\mathbf{Coalg}(\mathscr{D})$ is a colifting of $\mathsf{H} : \mathscr{C} \to \mathscr{D}$ iff $\mathsf{U}_{\mathsf{G}} \circ \underline{\mathsf{H}} = \mathsf{H} \circ \mathsf{U}_{\mathsf{F}}$. Given $\lambda : \mathsf{H} \circ \mathsf{F} \dot{\to} \mathsf{G} \circ \mathsf{H}$ we can define a colifting as follows:

$$\mathsf{H}_{\lambda}\ (C,c) = (\mathsf{H}\,C, \lambda\ C \cdot \mathsf{H}\,c) \qquad \mathsf{H}_{\lambda}\ h = \mathsf{H}\,h\ . \tag{2.2}$$

2.2 Adjunctions

Adjunctions were introduced by Kan [17] and are so pervasive in the study of category theory that Mac Lane [19, p.vii] noted "Adjoint functors arise everywhere." Our work supports this view: adjunctions provide a unified framework for program transformation.

Given categories $\mathscr{C},\mathscr{D}$, we say that functors $\mathsf{L} : \mathscr{C} \leftarrow \mathscr{D}$ and $\mathsf{R} : \mathscr{C} \to \mathscr{D}$ form an adjunction, written $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \rightharpoonup \mathscr{D}$ or

$$\mathscr{C} \underset{\mathsf{R}}{\overset{\mathsf{L}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{D} \quad ,$$

iff there is a bijection between the sets of arrows

$$\lfloor - \rfloor : \mathscr{C}(\mathsf{L}\,A, B) \cong \mathscr{D}(A, \mathsf{R}\,B) : \lceil - \rceil$$

that is natural both in A and B. We say that L is a *left adjoint* for R, and R a *right adjoint* for L; the isomorphism $\lfloor - \rfloor$ is called the *left adjunct*, and its inverse $\lceil - \rceil$ the *right adjunct*.

That the adjuncts $\lfloor - \rfloor$ and $\lceil - \rceil$ are mutually inverse can be captured using an equivalence:

$$f = \lceil g \rceil \iff \lfloor f \rfloor = g \ , \tag{2.3}$$

for all $f : \mathsf{L}\,A \to B : \mathscr{C}$ and $g : A \to \mathsf{R}\,B : \mathscr{D}$. The naturality properties of the adjuncts are alternatively expressed as fusion laws.

$$\mathsf{R}\,k \cdot \lfloor f \rfloor \cdot h = \lfloor k \cdot f \cdot \mathsf{L}\,h \rfloor \tag{2.4a}$$

$$k \cdot \lceil g \rceil \cdot \mathsf{L}\,h = \lceil \mathsf{R}\,k \cdot g \cdot h \rceil \tag{2.4b}$$

An alternative definition of adjunctions is based on the natural transformations $\epsilon = \lceil id \rceil$ and $\eta = \lfloor id \rfloor$, which are called the *counit* $\epsilon : \mathsf{L} \circ \mathsf{R} \dot{\to} \mathsf{Id}$ and the *unit* $\eta : \mathsf{Id} \dot{\to} \mathsf{R} \circ \mathsf{L}$ of the adjunction. All in all, an adjunction consists of six entities: two functors, two adjuncts, and two units. Each can be defined in terms of the others:

$$\begin{array}{llll} \lceil g \rceil = \epsilon\,B \cdot \mathsf{L}\,g & \epsilon = \lceil id \rceil & \mathsf{L}\,h = \lceil \eta\,B \cdot h \rceil \\ \lfloor f \rfloor = \mathsf{R}f \cdot \eta\,A & \eta = \lfloor id \rfloor & \mathsf{R}\,k = \lfloor k \cdot \epsilon\,A \rfloor \ . \end{array} \tag{2.5}$$

Example 2.2. Coproducts and products arise as left and right adjoints $(+) \dashv \Delta \dashv (\times)$ of the diagonal functor $\Delta : \mathscr{C} \to \mathscr{C} \times \mathscr{C}$ defined by $\Delta\,A = (A,A)$ and $\Delta f = (f,f)$.

$$\mathscr{C} \underset{\Delta}{\overset{(+)}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{C} \times \mathscr{C} \qquad \mathscr{C} \times \mathscr{C} \underset{(\times)}{\overset{\Delta}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{C}$$

The bijections express that pairs of arrows with the same target (respectively, source) are in 1-1 correspondence with arrows from a coproduct (respectively, to a product). In the case of products, the left adjunct $\lfloor (f_1, f_2) \rfloor = f_1 \vartriangle f_2$ is known as the 'split' combinator, and the counit $\epsilon = (outl, outr)$ arises from the projections.

$$(f_1, f_2) = (outl, outr) \cdot \Delta\,g \iff f_1 \vartriangle f_2 = g \tag{2.6a}$$

Note that the equation on the left lives in a product category. For products the fusion law (2.4a) specializes to

$$(k_1 \times k_2) \cdot (f_1 \vartriangle f_2) \cdot h = (k_1 \cdot f_1 \cdot h) \vartriangle (k_2 \cdot f_2 \cdot h) \ . \tag{2.6b}$$

Both laws will be frequently used in calculations. □

Example 2.3. Perhaps the best-known example of an adjunction is currying: a function of two arguments can be treated as a function of the first argument whose values are functions of the second.

$$\mathscr{C} \underset{(-)^P}{\overset{-\times P}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathscr{C}$$

The right adjoint of pairing with P is the exponential from P. The function Λ (pronounce: curry) corresponds to the left adjunct:

$$\Lambda : \mathscr{C}(A \times P, B) \cong \mathscr{C}(A, B^P) \ .$$

In **Set**, the object B^P is the set of total functions from P to B. □

Left adjoints preserve initial objects, $\mathsf{L}\,0 \cong 0$. Dually, right adjoints preserve final objects, $\mathsf{R}\,1 \cong 1$. In general, left adjoints preserve colimits (LAPC) and right adjoints preserve limits (RAPL).

Example 2.4. Possibly infinite trees with branching structure determined by G and labels drawn from A constitute the *cofree* G*-coalgebra* $\mathsf{Cofree}_{\mathsf{G}}\,A$, which arises as the right adjoint of the underlying functor U_{G}.

$$\mathscr{C} \underset{\mathsf{Cofree}_{\mathsf{G}}}{\overset{\mathsf{U}_{\mathsf{G}}}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathsf{G}\text{-}\mathbf{Coalg}(\mathscr{C})$$

The action of $\mathsf{Cofree}_{\mathsf{G}}\,A = (\mathsf{G}_\infty\,A, \mathsf{tail}\,A)$ maps a tree to a G-structure of subtrees. The counit $\epsilon = \mathsf{head}$ extracts the label of the root of a tree, and $\eta\,(A,a) = \llbracket a \rrbracket$ constructs a tree. The bijection instantiating (2.3) expresses that the tree for a given start state and a given state-transition function expressed as a coalgebra is uniquely determined by a mapping from states to observations:

$$f = \mathsf{head}\,B \cdot \mathsf{U}_{\mathsf{G}}\,g \iff \mathsf{Cofree}_{\mathsf{G}}\,f \cdot \llbracket a \rrbracket = g \ , \tag{2.7}$$

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

Since $\mathsf{Cofree}_{\mathsf{G}}$ is a right adjoint, final coalgebras arise as a special case (RAPL): $(\nu\mathsf{G}, out) \cong \mathsf{Cofree}_{\mathsf{G}}\,1$.

Conversely, cofree coalgebras can be implemented in terms of final coalgebras: $(\mathsf{T}\,B, \mathsf{tail}\,B)$ with $\mathsf{head}\,B : \mathsf{T}\,B \to B$ is a cofree G-coalgebra if and only if $(\mathsf{T}\,B, \mathsf{head}\,B \vartriangle \mathsf{tail}\,B)$ is a final G_B-coalgebra where $\mathsf{G}_B\,X = B \times \mathsf{G}\,X$. It suffices to show that $h : A \to \mathsf{T}\,B$ with $\mathsf{head}\,B \cdot h = hd$ is a G-coalgebra homomorphism if and only if it is a G_B-coalgebra homomorphism. The proof is standard, but instructive:

$$\begin{array}{ll} & \mathsf{head}\,B \cdot h = hd \ \wedge \ \mathsf{tail}\,B \cdot h = \mathsf{G}\,h \cdot tl \\ \iff & \{\ \vartriangle \text{ is an isomorphism (2.6a)}\ \} \\ & \mathsf{head}\,B \cdot h \vartriangle \mathsf{tail}\,B \cdot h = hd \vartriangle \mathsf{G}\,h \cdot tl \\ \iff & \{\ \text{product fusion (2.6b)}\ \} \\ & (\mathsf{head}\,B \vartriangle \mathsf{tail}\,B) \cdot h = hd \vartriangle \mathsf{G}\,h \cdot tl \\ \iff & \{\ \text{definition of } \mathsf{G}_B \text{ and product fusion (2.6b)}\ \} \\ & (\mathsf{head}\,B \vartriangle \mathsf{tail}\,B) \cdot h = \mathsf{G}_B\,h \cdot (hd \vartriangle tl) \end{array}$$

Note that every G_B-coalgebra is of the form $hd \vartriangle tl$, since $c = outl \cdot c \vartriangle outr \cdot c$. Consequently,

$$\mathsf{G}_\infty\,B \cong \nu\,X\,.\,B \times \mathsf{G}\,X \ . \tag{2.8}$$

This implies that $\mathsf{head}\,B \vartriangle \mathsf{tail}\,B$ is an isomorphism; we use $\mathsf{cons}\,B$ to denote its inverse. As an aside, (2.8) motivates the nomenclature: $\mathsf{G}_\infty\,B$ can be seen as the type of generalized streams—'generalized' because the 'tail' is a G-structure of 'streams' rather than just a single one; we obtain standard streams for $\mathsf{G} = \mathsf{Id}$. □

Conjugates At the heart of several of our core proofs are conjugate natural transformations [19]. Just as natural transformations relate functors, conjugates relate adjoint pairs of functors. Given the adjunctions $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \rightharpoonup \mathscr{D}$ and $\mathsf{L}' \dashv \mathsf{R}' : \mathscr{C}' \rightharpoonup \mathscr{D}'$, and functors $\mathsf{H} : \mathscr{C} \to \mathscr{C}'$ and $\mathsf{K} : \mathscr{D} \to \mathscr{D}'$, the natural transformations $\sigma : \mathsf{L}' \circ \mathsf{K} \dot{\to} \mathsf{H} \circ \mathsf{L}$ and $\tau : \mathsf{K} \circ \mathsf{R} \dot{\to} \mathsf{R}' \circ \mathsf{H}$ are *conjugates*, written $\sigma \dashv \tau$, if one of

$$\lfloor \mathsf{H}f \cdot \sigma\,A \rfloor' = \tau\,B \cdot \mathsf{K} \lfloor f \rfloor \ , \tag{2.9a}$$

$$\mathsf{H} \lceil g \rceil \cdot \sigma\,A = \lceil \tau\,B \cdot \mathsf{K}\,g \rceil' \ , \tag{2.9b}$$

holds, for all $f : \mathsf{L}\,A \to B : \mathscr{C}$ and $g : A \to \mathsf{R}\,B : \mathscr{D}$. An important property is that each component uniquely determines the other.

3. Recursive Coalgebras

Hylomorphisms, or hylos for short, are solutions to 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; second, sub-problems are recursively and independently turned into sub-solutions; and finally, sub-solutions are combined by an algebra to form a solution. The recursive call structure of the hylo is determined by the common base functor of the coalgebra and the algebra. Depending on its shape, hylomorphisms capture while-programs, binary-subdivision-schemes etc. In fact, practically any program can be cast into the hylo form [15]. However, there is no a priori guarantee that a solution exists for a hylo equation with a given coalgebra and algebra, let alone that such a solution is uniquely determined.

3.1 Basic Definitions

Hylomorphisms Let $\mathsf{F} : \mathscr{C} \to \mathscr{C}$ be an endofunctor, let (a,A) be an F-algebra, and let (C,c) be an F-coalgebra. An arrow $h : A \leftarrow C$ is a *hylomorphism* (or algebra-from-coalgebra homomorphism), $h : (a,A) \leftarrowtail (C,c)$, if it satisfies

$$h = a \cdot \mathsf{F}\, h \cdot c \ . \tag{3.1}$$

The *control functor* F governs the recursive call structure. We use $(a,A) \leftarrowtail (C,c)$ or $(C,c) \rightarrowtail (a,A)$ to denote the set of all hylomorphisms that satisfy (3.1).

Example 3.1. The functor $(A + -)$ can be used to model tail recursion: a loop either terminates producing an A value, or it goes through another iteration. Tail-recursive programs are captured by

$$h = (id \triangledown id) \cdot (A + h) \cdot c \quad \Longleftrightarrow \quad h = (id \triangledown h) \cdot c \ ,$$

where $id \triangledown id : A + A \to A$ is the so-called *codiagonal*. □

Unlike algebra or coalgebra homomorphisms, hylomorphisms do *not* compose, so they do *not* form a category. However, they do compose with algebra and coalgebra homomorphisms.

$$\begin{array}{ccccccc} A' & \xleftarrow{f} & A & \xleftarrow{h} & C & \xleftarrow{g} & C' \\ {\scriptstyle a'}\uparrow & & {\scriptstyle a}\uparrow & & \downarrow{\scriptstyle c} & & \downarrow{\scriptstyle c'} \\ \mathsf{F}\,A' & \xleftarrow[\mathsf{F}f]{} & \mathsf{F}\,A & \xleftarrow[\mathsf{F}\,h]{} & \mathsf{F}\,C & \xleftarrow[\mathsf{F}\,g]{} & \mathsf{F}\,C' \end{array} \quad = \quad \begin{array}{ccc} A' & \xleftarrow{f\cdot h\cdot g} & C' \\ {\scriptstyle a'}\uparrow & & \downarrow{\scriptstyle c'} \\ \mathsf{F}\,A' & \xleftarrow[\mathsf{F}\,(f\cdot h\cdot g)]{} & \mathsf{F}\,C' \end{array}$$

Thus, the assignment $((C,c),(a,A)) \mapsto ((C,c) \rightarrowtail (a,A))$ can be turned into a functor of type $(\mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C}))^{\mathsf{op}} \times \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C}) \to \mathbf{Set}$.

We are particularly interested in situations where (3.1) has a unique solution. In general, the algebra and coalgebra work hand-in-hand to achieve unicity. However, at the extreme, uniqueness can be shown by focusing on either the coalgebra or the algebra.

Recursive Coalgebras A coalgebra (C,c) is *recursive* (or algebra-initial) if for every algebra (a,A) there is a *unique* hylo $(a,A) \leftarrowtail (C,c)$. We denote such hylos by $(\!|a \leftarrow c|\!)$ to emphasize the source of uniqueness. This notation is reminiscent of that of folds, since a recursive coalgebra provides a kind of initiality: post-composing a hylo with any algebra homomorphism $h : (a,A) \to (b,B)$ yields again a unique hylo, $h \cdot (\!|a \leftarrow c|\!) = (\!|b \leftarrow c|\!)$. The full subcategory of $\mathsf{F}\text{-}\mathbf{Coalg}(\mathscr{C})$ of recursive coalgebras is denoted $\mathsf{F}\text{-}\mathbf{Rec}(\mathscr{C})$.

Example 3.2. Lambek's Lemma states that the action of the initial algebra $(in, \mu\mathsf{F})$ has an inverse, which implies

$$h \cdot in = a \cdot \mathsf{F}\, h \quad \Longleftrightarrow \quad h = a \cdot \mathsf{F}\, h \cdot in^\circ \ .$$

Consequently, $(\mu\mathsf{F}, in^\circ)$ is recursive and $(\!|a|\!) = (\!|a \leftarrow in^\circ|\!)$. □

Corecursive Algebras Dually, an algebra (a,A) is *corecursive* (or coalgebra-final) if for every coalgebra (C,c) there is a *unique* hylo $(a,A) \leftarrowtail (C,c)$. We denote such hylos by $[\![a \leftarrow c]\!]$ to emphasize the source of this uniqueness. The full subcategory of $\mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$ of corecursive algebras is denoted $\mathsf{F}\text{-}\mathbf{Corec}(\mathscr{C})$.

For example, the algebras $(out^\circ, \nu\mathsf{G})$ and $(\mathsf{cons}\ B, \mathsf{G}_\infty\ B)$ are corecursive (recall that $\mathsf{cons}\ B = (\mathsf{head}\ B \vartriangle \mathsf{tail}\ B)^\circ$).

With these definitions in place, we proceed by describing two fundamental techniques for establishing uniqueness: Eppendahl's Basic Lemma [7], which we call the *rolling rule*, and a symmetric version of Proposition 12 of [6], which we call the *conjugate rule*. Indeed, we will be careful to emphasize symmetry, even though our main focus is on recursive coalgebras.

3.2 Rolling Rule

We now consider algebras and coalgebras where *two* functors compose to create the base functor. The rolling rule allows us to swap the underlying functors, and establishes a means of deriving new recursive coalgebras and corecursive algebras from old ones. We start with the situation described in the following diagram.

$$\begin{array}{ccc} (\mathsf{L}\circ\mathsf{R})\text{-}\mathbf{Alg}(\mathscr{C}) & \xrightarrow[\overline{\mathsf{R}}]{} & (\mathsf{R}\circ\mathsf{L})\text{-}\mathbf{Alg}(\mathscr{D}) \\ {\scriptstyle \mathsf{U}^{\mathsf{L}\circ\mathsf{R}}}\downarrow & & \downarrow{\scriptstyle \mathsf{U}^{\mathsf{R}\circ\mathsf{L}}} \\ \mathscr{C} & \underset{\mathsf{R}}{\overset{\mathsf{L}}{\rightleftarrows}} & \mathscr{D} \\ {\scriptstyle \mathsf{U}_{\mathsf{L}\circ\mathsf{R}}}\uparrow & & \uparrow{\scriptstyle \mathsf{U}_{\mathsf{R}\circ\mathsf{L}}} \\ (\mathsf{L}\circ\mathsf{R})\text{-}\mathbf{Coalg}(\mathscr{C}) & \xleftarrow[\underline{\mathsf{L}}]{} & (\mathsf{R}\circ\mathsf{L})\text{-}\mathbf{Coalg}(\mathscr{D}) \end{array} \tag{3.2a}$$

Since the base functors are compositions, the functor L has a trivial colifting to categories of coalgebras. Recall that we need a natural transformation of type $\mathsf{L}\circ(\mathsf{R}\circ\mathsf{L}) \dot{\to} (\mathsf{L}\circ\mathsf{R})\circ\mathsf{L}$; the identity will do nicely: $\underline{\mathsf{L}} = \mathsf{L}_{id}$, and, likewise $\overline{\mathsf{R}} = \mathsf{R}^{id}$.

The *rolling rule* establishes an adjunction-like correspondence between two types of hylomorphisms.

Theorem 3.1 (Rolling Rule)**.** *Assuming the data in* (3.2a)*, there is a bijection between the sets of hylos*

$$\underline{\mathsf{L}}\,(C,c) \rightarrowtail (a,A) \ \cong \ (C,c) \rightarrowtail \overline{\mathsf{R}}\,(a,A) \ , \tag{3.2b}$$

natural in $(C,c) : (\mathsf{R}\circ\mathsf{L})\text{-}\mathbf{Coalg}(\mathscr{D})$ *and* $(a,A) : (\mathsf{L}\circ\mathsf{R})\text{-}\mathbf{Alg}(\mathscr{C})$.

Proof. The correspondence is witnessed by the functions $\lfloor - \rfloor$ and $\lceil - \rceil$, defined $\lfloor x \rfloor = \mathsf{R}\, x \cdot c$ and $\lceil y \rceil = a \cdot \mathsf{L}\, y$. These trivially form a bijection between fixed-points of $\lceil \lfloor - \rfloor \rceil$ and fixed-points of $\lfloor \lceil - \rceil \rfloor$:

$$x = \lceil \lfloor x \rfloor \rceil \quad \Longleftrightarrow \quad y = \lfloor \lceil y \rceil \rfloor \ ,$$

where $y = \lfloor x \rfloor$ and $x = \lceil y \rceil$. Unrolling the definitions we obtain

$$x = a \cdot \mathsf{L}\,(\mathsf{R}\, x) \cdot \mathsf{L}\, c \quad \Longleftrightarrow \quad y = \mathsf{R}\, a \cdot \mathsf{R}\,(\mathsf{L}\, y) \cdot c \ ,$$

which shows that $x : \underline{\mathsf{L}}\,(C,c) \rightarrowtail (a,A)$ and $y : (C,c) \rightarrowtail \overline{\mathsf{R}}\,(a,A)$. The proof of naturality is straightforward. □

The 1-1 correspondence (3.2b) allows us to conclude that $\underline{\mathsf{L}}\,(C,c)$ is recursive if (C,c) is. Thus, the colifting $\underline{\mathsf{L}}$ preserves recursiveness, and, dually, the lifting $\overline{\mathsf{R}}$ preserves corecursiveness.

$$\underline{\mathsf{L}} : (\mathsf{L}\circ\mathsf{R})\text{-}\mathbf{Rec}(\mathscr{C}) \leftarrow (\mathsf{R}\circ\mathsf{L})\text{-}\mathbf{Rec}(\mathscr{D}) \tag{3.3a}$$

$$\overline{\mathsf{R}} : (\mathsf{L}\circ\mathsf{R})\text{-}\mathbf{Corec}(\mathscr{C}) \to (\mathsf{R}\circ\mathsf{L})\text{-}\mathbf{Corec}(\mathscr{D}) \tag{3.3b}$$

3.3 Final Recursive Coalgebras

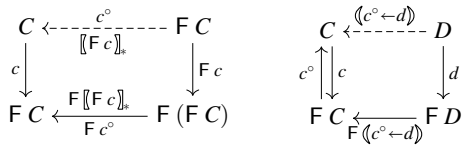
An important special case of the rolling rule is where we substitute $\mathsf{L}, \mathsf{R} := \mathsf{F}, \mathsf{Id}$. We use this to prove the following

Theorem 3.2. *The recursive coalgebra* (C,c) *is final if and only if* c *is an isomorphism.*

The unique F-coalgebra homomorphism from the recursive coalgebra (C,c) to the final recursive coalgebra is written $[\![c]\!]_*$, which emphasizes that this unfolds into a *recursive* coalgebra.

Proof. "$\Longrightarrow$": The inverse of c is the unique coalgebra homomorphism $c^\circ = [\![\mathsf{F}\, c]\!]_*$—using the rolling rule we know that $\mathsf{F}\, c$ is recursive

if c is recursive (3.3a).



Now, $c^\circ \cdot c = id$ because (C,c) is final. Conversely, $c \cdot c^\circ = \mathsf{F}\, c^\circ \cdot \mathsf{F}\, c = id$, see diagram on the left above.

"$\Longleftarrow$": if c is an iso, then $(\!|c^\circ \leftarrow d|\!)$ is the unique coalgebra homomorphism to (C,c) from the recursive coalgebra (D,d), see diagram on the right above. Hence, (C,c) is final. □

Corollary 3.3. *The final recursive* F*-coalgebra is* $(\mu\mathsf{F}, in^\circ)$.

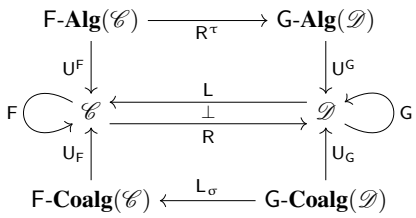
Corollary (3.3) is an important result as it allows us to freely mix folds $(\!|a|\!)$ as consumers with recursive unfolds $[\![c]\!]_*$ as producers.

3.4 Conjugate Rule

The rolling rule can only be applied when a pair of functors form the base functor of the (co)algebra. We can lift this restriction to simple endo-functors $\mathsf{F} : \mathscr{C} \to \mathscr{C}$ and $\mathsf{G} : \mathscr{D} \to \mathscr{D}$ when there is an adjunction between $\mathscr{C}$ and $\mathscr{D}$. Given $\mathsf{L} \dashv \mathsf{R} : \mathscr{C} \rightharpoonup \mathscr{D}$, and conjugates

$$\sigma : \mathsf{L} \circ \mathsf{G} \dot{\to} \mathsf{F} \circ \mathsf{L} \dashv \tau : \mathsf{G} \circ \mathsf{R} \dot{\to} \mathsf{R} \circ \mathsf{F} \ ,$$

we have the following situation.



(3.4a)

Using the conjugates we colift L to categories of coalgebras (2.2) and lift R to categories of algebras (2.1).

Like the rolling rule, the *conjugate rule* establishes an adjunction-like correspondence between two types of hylomorphism.

Theorem 3.4 (Conjugate Rule)**.** *Assuming the data in* (3.4a)*, there is a bijection between the sets of conjugate hylomorphisms*

$$\mathsf{L}_\sigma\ (C,c) \rightarrowtail (a,A) \ \cong\ (C,c) \rightarrowtail \mathsf{R}^\tau\ (a,A) \ , \tag{3.4b}$$

that is natural in $(C,c) : \mathsf{G}\text{-}\mathbf{Coalg}(\mathscr{D})$ *and* $(a,A) : \mathsf{F}\text{-}\mathbf{Alg}(\mathscr{C})$.

Using the conjugates is a natural choice: just as adjoint functors determine one another (up to $\cong$), so too do conjugates. It is this property that allows us to move between algebras and coalgebras.

Proof. We have to show

$$x = a \cdot \mathsf{F}\, x \cdot \mathsf{L}_\sigma\, c \iff \lfloor x \rfloor = \mathsf{R}^\tau\, a \cdot \mathsf{G}\, \lfloor x \rfloor \cdot c \ , \tag{3.4c}$$

where $\lfloor - \rfloor$ is the left adjunct of $\mathsf{L} \dashv \mathsf{R}$.

$$\begin{array}{ll} & x = a \cdot \mathsf{F}\, x \cdot \mathsf{L}_\sigma\, c \ : \ A \leftarrow \mathsf{L}\, C \\ \iff & \{ \text{ definition of colifting (2.2) } \} \\ & x = a \cdot \mathsf{F}\, x \cdot \sigma\, C \cdot \mathsf{L}\, c \\ \iff & \{ \lfloor - \rfloor \text{ and } \lceil - \rceil \text{ are isomorphisms (2.3) } \} \\ & \lfloor x \rfloor = \lfloor a \cdot \mathsf{F}\, x \cdot \sigma\, C \cdot \mathsf{L}\, c \rfloor \\ \iff & \{ \lfloor - \rfloor \text{ is natural (2.4a) } \} \\ & \lfloor x \rfloor = \mathsf{R}\, a \cdot \lfloor \mathsf{F}\, x \cdot \sigma\, C \rfloor \cdot c \\ \iff & \{ \sigma \dashv \tau \text{ conjugates (2.9a) } \} \\ & \lfloor x \rfloor = \mathsf{R}\, a \cdot \tau\, A \cdot \mathsf{G}\, \lfloor x \rfloor \cdot c \\ \iff & \{ \text{ definition of lifting (2.1) } \} \end{array}$$

$$\lfloor x \rfloor = \mathsf{R}^\tau\, a \cdot \mathsf{G}\, \lfloor x \rfloor \cdot c \ : \ \mathsf{R}\, A \leftarrow C \ .$$

Naturality of (3.4b) is inherited from the underling adjunction. □

The 1-1 correspondence (3.4b) between what we call *conjugate hylomorphisms* implies that $\mathsf{L}_\sigma\ (C,c)$ is recursive if (C,c) is. Thus, L_σ preserves recursiveness; dually, R^τ preserves corecursiveness.

$$\mathsf{L}_\sigma : \mathsf{F}\text{-}\mathbf{Rec}(\mathscr{C}) \leftarrow \mathsf{G}\text{-}\mathbf{Rec}(\mathscr{D}) \tag{3.5a}$$

$$\mathsf{R}^\tau : \mathsf{F}\text{-}\mathbf{Corec}(\mathscr{C}) \to \mathsf{G}\text{-}\mathbf{Corec}(\mathscr{D}) \tag{3.5b}$$

Moreover, using our notation for hylos, (3.4c) gives:

$$\lfloor (\!| a \leftarrow \mathsf{L}_\sigma\, c |\!) \rfloor = (\!| \mathsf{R}^\tau\, a \leftarrow c |\!) \ , \tag{3.6a}$$

$$\lceil [\![\mathsf{R}^\tau\, a \leftarrow c]\!] \rceil = [\![a \leftarrow \mathsf{L}_\sigma\, c]\!] \ . \tag{3.6b}$$

3.4.1 Data and Control Functors

The conjugate rule involves six entities—four functors and two natural transformations—which can be a little daunting. When the adjunction $\mathsf{L} \dashv \mathsf{R}$ is fixed, what remains is the choice of functors F and G, and the conjugate $\sigma \dashv \tau$. Often one of the functors can be identified as a *data functor* as it is either part of the input or output type of a hylo. Then there is a canonical choice for the other functor, the *control functor* that governs the recursive call structure.

Focusing on the input, the framework of the conjugate rule can be instantiated with the data functor $\mathsf{G} := \mathsf{D}$, and the control functor $\mathsf{F} := \mathsf{C} = \mathsf{L} \circ \mathsf{D} \circ \mathsf{R}$ with the following conjugate pair:

$$\sigma = \mathsf{L} \circ \mathsf{D} \circ \eta : \mathsf{L} \circ \mathsf{D} \dot{\to} \mathsf{C} \circ \mathsf{L} \ \dashv \ \tau = \eta \circ \mathsf{D} \circ \mathsf{R} : \mathsf{D} \circ \mathsf{R} \dot{\to} \mathsf{R} \circ \mathsf{C} \ .$$

The control functor C is obtained by going round in a circle, see Diagram (3.4a). It is canonical in the following sense: any hylo equation $x = a \cdot \mathsf{C}'\, x \cdot \mathsf{L}_{\sigma'}\, c$ with $\sigma' : \mathsf{L} \circ \mathsf{D} \dot{\to} \mathsf{C}' \circ \mathsf{L}$ is equivalent to one that uses the canonical control functor [13].

For this particular instance the conjugate rule can be simplified: using (2.5) it is straightforward to show that $\mathsf{R}^\tau\ a = \lfloor a \rfloor$ and $\mathsf{C}\, x \cdot \mathsf{L}_\sigma\, c = \mathsf{L}\, (\mathsf{D}\, \lfloor x \rfloor \cdot c)$. Thus, (3.4c) becomes:

$$x = a \cdot \mathsf{L}\, (\mathsf{D}\, \lfloor x \rfloor \cdot c) \iff \lfloor x \rfloor = \lfloor a \rfloor \cdot \mathsf{D}\, \lfloor x \rfloor \cdot c \ . \tag{3.7}$$

Dually, we can focus on the output. Then the roles of F and G are interchanged: we instantiate the conjugate rule to $\mathsf{F}, \mathsf{G} := \mathsf{D}, \mathsf{C}$ with the canonical control functor $\mathsf{C} = \mathsf{R} \circ \mathsf{D} \circ \mathsf{L}$, and conjugates:

$$\sigma = \epsilon \circ \mathsf{D} \circ \mathsf{L} : \mathsf{L} \circ \mathsf{C} \dot{\to} \mathsf{D} \circ \mathsf{L} \ \dashv \ \tau = \mathsf{R} \circ \mathsf{D} \circ \epsilon : \mathsf{C} \circ \mathsf{R} \dot{\to} \mathsf{R} \circ \mathsf{D} \ .$$

To illustrate the versatility of the conjugate rule we instantiate the rule to the adjunctions discussed in Section 2.2. Since we focus on recursive coalgebras, we consider instances of the equation

$$x = a \cdot \mathsf{F}\, x \cdot \mathsf{L}_\sigma\, c \qquad \begin{array}{ccc} A & \xleftarrow{\ x\ } & \mathsf{L}\, C \\ {\scriptstyle a}\uparrow & & \downarrow{\scriptstyle \mathsf{L}_\sigma\, c} \\ \mathsf{F}\, A & \xleftarrow[\mathsf{F}\, x]{} & \mathsf{F}\, (\mathsf{L}\, C) \end{array} \ , \tag{3.8}$$

and of the equation that builds on the canonical control functor:

$$x = a \cdot \mathsf{L}\, (\mathsf{D}\, \lfloor x \rfloor \cdot c) \qquad \begin{array}{ccc} A & \xleftarrow{\ x\ } & \mathsf{L}\, C \\ {\scriptstyle a}\uparrow & & \downarrow{\scriptstyle \mathsf{L}\, c} \\ \mathsf{L}\, (\mathsf{D}\, (\mathsf{R}\, A)) & \xleftarrow[\mathsf{L}\, (\mathsf{D}\, \lfloor x \rfloor)]{} & \mathsf{L}\, (\mathsf{D}\, C) \end{array} \ . \tag{3.9}$$

We already know that both equations have unique solutions if c is recursive. We shall see that each of the adjunctions induces a useful recursion scheme. In fact, almost every structured recursion scheme seems to arise this way (an obvious exception being schemes that make use of the rolling rule, but that do not involve adjoint functors). The reader is invited to dualize.

3.4.2 Instance: Hylo-shift Law: $\mathsf{Id} \dashv \mathsf{Id}$

As a warm-up, consider the adjunction $\mathsf{Id} \dashv \mathsf{Id}$. For this trivial case, a conjugate pair is given by a natural transformation $\alpha : \mathsf{G} \dot{\to} \mathsf{F}$.

$$\alpha : \mathsf{Id} \circ \mathsf{G} \dot{\to} \mathsf{F} \circ \mathsf{Id} \quad \dashv \quad \alpha : \mathsf{G} \circ \mathsf{Id} \dot{\to} \mathsf{Id} \circ \mathsf{F}$$

Then (3.6a) specializes to the so-called *hylo-shift law* [23]:

$$(\!|a \leftarrow \alpha\, C \cdot c|\!) = (\!|a \cdot \alpha\, A \leftarrow c|\!) \ , \tag{3.10}$$

which allows us to shift a natural transformation between the algebra and coalgebra part of a hylomorphism.

3.4.3 Instance: Mutual Recursion: $\Delta \dashv (\times)$

Mutual recursion describes the situation where a number of functions rely on one another in calculating their results.

Example 3.3 (Minimax)**.** An example of mutual recursion is the minimax algorithm. Consider a two-player game where starting at the root of a finite tree, the players take it in turn to choose whether the left or right branch of the tree is taken. The final score is the sum of all the nodes that have been visited before the final leaf. The task of one player is to maximize this score, while the other player tries to minimize it. The two mutually recursive functions are:

$$\begin{array}{l} \mathit{maximize}, \mathit{minimize} :: \mathit{Tree} \to \mathbb{Z} \\ \mathit{maximize}\ (\mathit{Empty}) \quad\ = 0 \\ \mathit{maximize}\ (\mathit{Node}\ l\ v\ r) = v + (\mathit{minimize}\ l\ `\mathit{max}`\ \mathit{minimize}\ r) \\ \mathit{minimize}\ \ (\mathit{Empty}) \quad\ = 0 \\ \mathit{minimize}\ \ (\mathit{Node}\ l\ v\ r) = v + (\mathit{maximize}\ l\ `\mathit{min}`\ \mathit{maximize}\ r) \ . \end{array}$$

These functions rely symmetrically on each other's results before proceeding to the next step. □

Example 3.4 (Perfect trees)**.** Another example of mutual recursion is where the dependence is asymmetric. Consider the function *perfect*, which checks to see whether a tree is perfectly balanced.

$$\begin{array}{l} \mathit{perfect} :: \mathit{Tree} \to \mathbb{B} \\ \mathit{perfect}\ (\mathit{Empty}) \quad\ = \mathit{True} \\ \mathit{perfect}\ (\mathit{Node}\ l\ v\ r) = \mathit{perfect}\ l \wedge \mathit{perfect}\ r \wedge \mathit{height}\ l == \mathit{height}\ r \\ \mathit{height} :: \mathit{Tree} \to \mathbb{Z} \\ \mathit{height}\ (\mathit{Empty}) \quad\ = 0 \\ \mathit{height}\ (\mathit{Node}\ l\ v\ r) = 1 + (\mathit{height}\ l)\ `\mathit{max}`\ (\mathit{height}\ r) \end{array}$$

The calculation of *perfect* depends on *height*, but the result of *height* is independent of *perfect*. □

In both examples, the mutually recursive functions have the same source type and both use the coalgebra in°, which suggests instantiating the conjugate rule to the adjunction $\Delta \dashv (\times)$.

Mutu-hylo When we pick the canonical control functor, we obtain mutually recursive hylos, or *mutu-hylos* for short. Given $x = (x_1, x_2)$ and $a = (a_1, a_2)$, Equation (3.9) specializes to

$$x = a \cdot \Delta\,(\mathsf{D}\ \lfloor x \rfloor \cdot c) \Longleftrightarrow \left\{ \begin{array}{l} x_1 = a_1 \cdot \mathsf{D}\ (x_1 \vartriangle x_2) \cdot c \ :\ A_1 \leftarrow C \\ x_2 = a_2 \cdot \mathsf{D}\ (x_1 \vartriangle x_2) \cdot c \ :\ A_2 \leftarrow C \ . \end{array} \right.$$

The algebras $a_1 : \mathsf{D}\ (A_1 \times A_2) \to A_1$ and $a_2 : \mathsf{D}\ (A_1 \times A_2) \to A_2$ can avail themselves of the results of both recursive calls. The coalgebra has to be the same, and it must be recursive.

A special case of the mutu-hylo is the *zygo-hylo*, where only one of the algebras is dependent on both results: $a_2 := a_2' \cdot \mathsf{D}\ \mathit{outr}$.

Banana-split Here is an amusing variation of the theme. The diagonal functor Δ satisfies a simple property: $\Delta \circ \mathsf{D} = (\mathsf{D} \times \mathsf{D}) \circ \Delta$. Let us instantiate the conjugate rule to $\mathsf{F}, \mathsf{G} := \mathsf{D} \times \mathsf{D}, \mathsf{D}$ with

$$\mathit{id} : \Delta \circ \mathsf{D} = (\mathsf{D} \times \mathsf{D}) \circ \Delta \quad \dashv \quad \tau : \mathsf{D} \circ (\times) \dot{\to} (\times) \circ (\mathsf{D} \times \mathsf{D}) \ .$$

Note that $\tau = \mathsf{D}\ \mathit{outl} \vartriangle \mathsf{D}\ \mathit{outr}$. Then Equation (3.8) specializes to

$$x = a \cdot (\mathsf{D} \times \mathsf{D})\ x \cdot \Delta_{id}\ c \Longleftrightarrow \left\{ \begin{array}{l} x_1 = a_1 \cdot \mathsf{D}\ x_1 \cdot c \ :\ A_1 \leftarrow C \\ x_2 = a_2 \cdot \mathsf{D}\ x_2 \cdot c \ :\ A_2 \leftarrow C \ . \end{array} \right.$$

We obtain a system of two independent functions, which appears unexciting. However, we can invoke (3.6a), which gives us

$$(\!|a_1 \leftarrow c|\!) \vartriangle (\!|a_2 \leftarrow c|\!) = (\!|a_1 \cdot \mathsf{D}\ \mathit{outl} \vartriangle a_2 \cdot \mathsf{D}\ \mathit{outr} \leftarrow c|\!) \ ,$$

as $(\times)^\tau\ (a_1, a_2) = a_1 \cdot \mathsf{D}\ \mathit{outl} \vartriangle a_2 \cdot \mathsf{D}\ \mathit{outr}$. We obtain a generalization of the-called *banana-split law* [4], an important program optimization that replaces a double recursion by a single one. (The law is called 'banana-split', because the fold brackets look like bananas and $\vartriangle$ is pronounced 'split'.)

3.4.4 Instance: Accumulators: $- \times P \dashv (-)^P$

A useful form of recursion is where an additional parameter, an accumulator, is passed in for use by the algebra.

Example 3.5 (Append)**.** Our first example of recursion with a parameter is in the definition of *cat*, which takes two lists and appends them together by structural induction on the first list.

$$\begin{array}{l} \mathit{cat} :: ([a], [a]) \to [a] \\ \mathit{cat}\ ([\,], \quad ys) = ys \\ \mathit{cat}\ (x : xs, ys) = x : \mathit{cat}\ (xs, ys) \end{array}$$

The second parameter is not changed during the recursion, and is simply passed through until the base case is reached. □

Example 3.6 (Tree flattening)**.** A slightly more elaborate example is an efficient version of turning a tree of values into a list. The naive version flattens both subtrees independently and appends the results. Since appending is of linear complexity in its first argument, the following variant is more desirable.

$$\begin{array}{l} \mathit{flattenCat} :: (\mathit{Tree}, [\mathbb{Z}]) \to [\mathbb{Z}] \\ \mathit{flattenCat}\ (\mathit{Empty}, \quad xs) = xs \\ \mathit{flattenCat}\ (\mathit{Node}\ l\ x\ r, xs) = \mathit{flattenCat}\ (l, x : \mathit{flattenCat}\ (r, xs)) \end{array}$$

The function spawns off two new invocations of itself, but only in the second does the parameter *xs* remain unchanged. □

The handling of parameters is a case for the curry adjunction.

Varying parameters Using the canonical control functor, Equation (3.9) specializes to the *accu-hylo*

$$x = a \cdot ((\mathsf{D}\ (\wedge\, x) \cdot c) \times P) \quad :\quad A \leftarrow C \times P \ .$$

This models the situation where the parameter varies during the recursion; $a : \mathsf{D}\ (A^P) \times P \to A$ receives a D-structure of curried variants of x, and can supply each of them with a different argument.

Constant parameters The case where the parameter is passed unchanged can be modelled using the so-called *strength* of a functor:

$$\sigma : (\times P) \circ \mathsf{D} \dot{\to} \mathsf{D} \circ (\times P) \quad \dashv \quad \tau : \mathsf{D} \circ (-)^P \dot{\to} (-)^P \circ \mathsf{D} \ .$$

The strength $\sigma\, A : \mathsf{D}\, A \times P \to \mathsf{D}\ (A \times P)$ broadcasts a value across a structure. (Every endofunctor on **Set** has a canonical strength.) For this data, Equation (3.8) specializes to

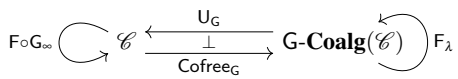
$$x = a \cdot \mathsf{D}\, x \cdot \sigma\, C \cdot (c \times P) \quad :\quad A \leftarrow C \times P \ .$$

The strength copies the parameter to each of the recursive calls.

3.4.5 Instance: Course-of-Values Recursion: $\mathsf{U}_\mathsf{G} \dashv \mathsf{Cofree}_\mathsf{G}$

The one remaining adjunction introduced in Section 2.2 that we have not yet discussed is $\mathsf{U}_\mathsf{G} \dashv \mathsf{Cofree}_\mathsf{G}$. It turns out that this instance gives us *course-of-values recursion*. We postpone a discussion of interesting programming examples until Section 4, where we look at various dynamic programming algorithms.

This instance is more challenging as it involves coalgebras over coalgebras! Consider the categories involved.



The data functor is now a functor over G-**Coalg**$(\mathscr{C})$, for example, the colifting F_λ where $\lambda : \mathsf{F}\circ\mathsf{G} \dot{\to} \mathsf{G}\circ\mathsf{F}$; the corresponding canonical control functor is $\mathsf{U}_\mathsf{G}\circ\mathsf{F}_\lambda\circ\mathsf{Cofree}_\mathsf{G} = \mathsf{F}\circ\mathsf{U}_\mathsf{G}\circ\mathsf{Cofree}_\mathsf{G} = \mathsf{F}\circ\mathsf{G}_\infty$. For this data, Equation (3.9) specializes to

$$\begin{array}{ll} & x = a\cdot \mathsf{U}_\mathsf{G}\,(\mathsf{F}_\lambda\,\lfloor x\rfloor\cdot c)\ :\ A\leftarrow \mathsf{U}_\mathsf{G}\,(C,d)\\ \iff & \{\ \mathsf{U}_\mathsf{G}\circ\mathsf{F}_\lambda = \mathsf{F}\circ\mathsf{U}_\mathsf{G}\ \}\\ & x = a\cdot \mathsf{F}\,(\mathsf{U}_\mathsf{G}\,\lfloor x\rfloor)\cdot \mathsf{U}_\mathsf{G}\,c\ :\ A\leftarrow \mathsf{U}_\mathsf{G}\,(C,d)\\ \iff & \{\ \lfloor x\rfloor = \mathsf{Cofree}_\mathsf{G}\,x\cdot\llbracket d\rrbracket\,(2.7) \text{ and definition of } \mathsf{U}_\mathsf{G}\ \}\\ & x = a\cdot \mathsf{F}\,(\mathsf{G}_\infty\,x\cdot\llbracket d\rrbracket)\cdot c\ :\ A\leftarrow C\ , \end{array}$$

where $a : \mathsf{F}\,(\mathsf{G}_\infty\,A)\to A$ and $c : (C,d)\to \mathsf{F}_\lambda\,(C,d)$. The coalgebra c lives in F_λ-**Coalg**$(\mathsf{G}$-**Coalg**$(\mathscr{C}))$; it is an F-coalgebra that is simultaneously a G-coalgebra homomorphism. If we wish to apply this instance of the conjugate rule, we have to ensure that $((C,d),c)$ is a recursive F_λ-coalgebra. The following lemma shows that F_λ-recursiveness is actually a weaker notion than F-recursiveness.

Lemma 3.5. *Let $\mathscr{D}$ be a full subcategory of G-**Coalg**$(\mathscr{C})$, let $\mathsf{F}_\lambda : \mathscr{D}\to\mathscr{D}$, and let $((C,d),c) : \mathsf{F}_\lambda$-**Coalg**$(\mathscr{D})$. Then*

$$(C,c) : \mathsf{F}\text{-}\mathbf{Rec}(\mathscr{C}) \implies ((C,d),c) : \mathsf{F}_\lambda\text{-}\mathbf{Rec}(\mathscr{D})\ .$$

Proof. We have to show that the equation in h

$$h = a\cdot \mathsf{F}\,h\cdot c\ :\ (A,b)\leftarrow(C,d) \tag{3.11}$$

has a unique solution for each F_λ-algebra $(a,(A,b))$. (An F_λ-algebra is also known as a λ-bialgebra; F_λ-coalgebras have been called λ-dicoalgebras [6].) If (C,c) is a recursive F-coalgebra, we know that (3.11) has a unique solution in $\mathscr{C}$. Since $\mathscr{D}$ is a full subcategory of G-**Coalg**$(\mathscr{C})$, it suffices to show that h is a G-coalgebra homomorphism of type $(A,b)\leftarrow(C,d)$.

$$\begin{array}{ll} & \mathsf{G}\,h\cdot d\\ b\cdot h & = \ \{\ h \text{ solves (3.11)}\ \}\\ = \ \{\ h \text{ solves (3.11)}\ \} & \mathsf{G}\,a\cdot\mathsf{G}\,(\mathsf{F}\,h)\cdot\mathsf{G}\,c\cdot d\\ b\cdot a\cdot \mathsf{F}\,h\cdot c & = \ \{\ c : (C,d)\to\mathsf{F}_\lambda\,(C,d)\ \}\\ = \ \{\ a : \mathsf{F}_\lambda\,(A,b)\to(A,b)\ \} & \mathsf{G}\,a\cdot\mathsf{G}\,(\mathsf{F}\,h)\cdot\lambda\,C\cdot\mathsf{F}\,d\cdot c\\ \mathsf{G}\,a\cdot\lambda\,A\cdot\mathsf{F}\,b\cdot\mathsf{F}\,h\cdot c & = \ \{\ \lambda \text{ is natural}\ \}\\ & \mathsf{G}\,a\cdot\lambda\,A\cdot\mathsf{F}\,(\mathsf{G}\,h)\cdot\mathsf{F}\,d\cdot c\ . \end{array}$$

Since $x = \mathsf{G}\,a\cdot\lambda\,A\cdot\mathsf{F}\,x\cdot c$ has a unique solution as c is recursive, we can conclude that $b\cdot h = \mathsf{G}\,h\cdot d$. □

Combining the conjugate rule and Lemma 3.5 we obtain

Corollary 3.6. *Let $((C,d),c)$ be a F_λ-coalgebra, then*

$$(C,c) : \mathsf{F}\text{-}\mathbf{Rec}(\mathscr{C}) \implies (C,\mathsf{F}\llbracket d\rrbracket\cdot c) : (\mathsf{F}\circ\mathsf{G}_\infty)\text{-}\mathbf{Rec}(\mathscr{C})\ .$$

An important special case arises if we identify F and G, setting $\mathsf{F} := \mathsf{G}$ and $\lambda = id : \mathsf{F}\circ\mathsf{F}\dot{\to}\mathsf{F}\circ\mathsf{F}$. Since a G-coalgebra $c : C\to\mathsf{G}\,C$ is also a G-coalgebra homomorphism $c : (C,c)\to\mathsf{G}_{id}\,(C,c)$, the nested coalgebra $((C,c),c)$ is a G_{id}-coalgebra. For this special case we can further simplify Equation (3.9) to

$$x = a\cdot\mathsf{G}\,(\mathsf{G}_\infty\,x\cdot\llbracket c\rrbracket)\cdot c\ :\ A\leftarrow C\ . \tag{3.12}$$

Operationally, $\mathsf{G}\llbracket c\rrbracket\cdot c = \mathsf{tail}\,C\cdot\llbracket c\rrbracket$ builds a table of all reachable arguments of x, excluding the current one. The hylomorphism x is then recursively applied to all of these, making the results available to the algebra $a : \mathsf{G}\,(\mathsf{G}_\infty\,A)\to A$. In other words, a can access the entire computational history, implementing *course-of-values recursion*. Incidentally, it may seem somewhat odd that a potentially infinite structure is used for tabulation, even though we know that c is recursive; we come back to this point in Section 5.4.

Efficiency Equation (3.12) specifies the notion of course-of-values recursion. However, it does not serve as a blue-print for an implementation, as it realizes the naive recursive definition, which typically leads to an exponential running time. A first step towards an efficient implementation is to define x in terms of its conjugate: $x = \mathsf{head}\,A\cdot\lfloor x\rfloor$ (2.7) where $\lfloor x\rfloor = \lfloor a\rfloor\cdot\mathsf{G}\,\lfloor x\rfloor\cdot c$ (3.7). The conjugate $\lfloor x\rfloor : (C,c)\to(\mathsf{G}_\infty\,A,\mathsf{tail}\,A)$ builds an entire memo-table bottom-up using $\lfloor a\rfloor = \mathsf{G}_\infty\,a\cdot\llbracket\mathsf{G}\,(\mathsf{tail}\,A)\rrbracket$. This is a vast improvement, but we can do better still, avoiding the costly $\mathsf{G}_\infty\,a$. Using the fact that $\lfloor x\rfloor$ is a G-coalgebra homomorphism, we reason

$$\begin{array}{ll} & \lfloor x\rfloor = \lfloor a\rfloor\cdot\mathsf{G}\,\lfloor x\rfloor\cdot c\ \wedge\ \mathsf{tail}\,A\cdot\lfloor x\rfloor = \mathsf{G}\,\lfloor x\rfloor\cdot c\\ \implies & \{\ \text{Leibniz's law and } \mathsf{head}\,A\cdot\lfloor a\rfloor = a\ (2.7)\ \}\\ & \mathsf{head}\,A\cdot\lfloor x\rfloor = a\cdot\mathsf{G}\,\lfloor x\rfloor\cdot c\ \wedge\ \mathsf{tail}\,A\cdot\lfloor x\rfloor = \mathsf{G}\,\lfloor x\rfloor\cdot c\\ \iff & \{\ \triangle \text{ is an isomorphism (2.6a)}\ \}\\ & \mathsf{head}\,A\cdot\lfloor x\rfloor\triangle\mathsf{tail}\,A\cdot\lfloor x\rfloor = a\cdot\mathsf{G}\,\lfloor x\rfloor\cdot c\triangle\mathsf{G}\,\lfloor x\rfloor\cdot c\\ \iff & \{\ \text{product fusion (2.6b)}\ \}\\ & (\mathsf{head}\,A\triangle\mathsf{tail}\,A)\cdot\lfloor x\rfloor = (a\triangle id)\cdot\mathsf{G}\,\lfloor x\rfloor\cdot c\\ \iff & \{\ \mathsf{head}\,A\triangle\mathsf{tail}\,A \text{ is an isomorphism}\ \}\\ & \lfloor x\rfloor = \mathsf{cons}\,A\cdot(a\triangle id)\cdot\mathsf{G}\,\lfloor x\rfloor\cdot c\ . \end{array}$$

Thus, $\lfloor x\rfloor = (\!|\mathsf{cons}\,A\cdot(a\triangle id)\leftarrow c|\!)$ and consequently

$$x = \mathsf{head}\,A\cdot(\!|\mathsf{cons}\,A\cdot(a\triangle id)\leftarrow c|\!)\ . \tag{3.13}$$

This implementation builds an entire memo-table, invoking a exactly once per node. Note that this does not depend on laziness.

3.5 Combining Adjunctions

Up to now we have considered various recursion schemes in isolation. Of course, in practice more complex algorithms are composed by combining different schemes. Consider as an example two mutually recursive functions, of which one has a parameter.

$$\begin{array}{l} x_1 = a_1\cdot\mathsf{D}\,(x_1\triangle\bigwedge x_2)\cdot c\\ x_2 = a_2\cdot(\mathsf{D}\,(x_1\triangle\bigwedge x_2)\times P)\cdot(c\times P) \end{array}$$

One of the attractive features of adjunctions is that we can easily combine simple adjunctions to form more complex ones. The recursion scheme above, for instance, is given by

$$(\mathsf{Id}\times(-\times P))\circ\Delta\ \dashv\ (\times)\circ(\mathsf{Id}\times(-)^P)\ :\ \mathscr{C}\times\mathscr{C}\to\mathscr{C}\ ,$$

which combines pairing, $\mathsf{L}_1\times\mathsf{L}_2\dashv\mathsf{R}_1\times\mathsf{R}_2$, and composition of adjunctions, $\mathsf{L}_2\circ\mathsf{L}_1\dashv\mathsf{R}_1\circ\mathsf{R}_2$. (Do not be confused by the four occurrences of $\times$: the formula involves products both in **Cat**, eg $\mathsf{Id}\times(-)^P$, and in $\mathscr{C}$, eg $-\times P$.) Indeed, all of the machinery introduced for *adjoint folds* [11] is immediately applicable here.

4. Application: Dynamic Programming I

We now discuss two examples of course-of-values recursion, introduced in Section 3.4.5, via dynamic programming algorithms.

The knapsack problem A classic use of dynamic programming is to solve the *unbounded knapsack problem*. The goal is to maximize the total value of elements that are placed into a knapsack with a fixed weight capacity c. The elements are represented as a list of pairs (w,v) where w is the weight and v is the value; elements are unbounded in multiplicity. The problem can be solved with the following recursion, where *wvs* is implicitly provided:

$$\begin{array}{l} \mathit{knapsack} :: \mathbb{N}\to\mathbb{R}\\ \mathit{knapsack}\ c = \mathit{maximum}_0\\ \quad [v + \mathit{knapsack}\ (c-w)\mid(w,v)\leftarrow \mathit{wvs}, 0<w\wedge w\leqslant c]\ . \end{array}$$

The value of a knapsack is determined by finding the item in *wvs* that maximizes the total value while also decreasing the capacity of the knapsack. The function $maximum_0$ returns 0 when given an empty list, and otherwise returns the maximum value in the list.

Dynamic programming optimizes equations like *knapsack* by replacing the recursive calls with look-ups into a memo-table. Since the guard $0 < w \wedge w \leqslant c$ ensures that the recursive calls are always on smaller arguments, this fits perfectly into the framework of Equation (3.12). The base functor is $\mathsf{G} := \mathsf{Nat}$ defined

$$\mathbf{data}\ \mathsf{Nat}\ nat = Zero \mid Succ\ nat\ \ ,$$

and c is simply the final recursive coalgebra $in^\circ : \mathbb{N} \to \mathsf{Nat}\ \mathbb{N}$. The memo-table Nat_∞ can be seen as the type of *non-empty colists*. The coalgebra $\mathsf{Nat}\ \llbracket in^\circ \rrbracket \cdot in^\circ$ of (3.12) generates the colist of all predecessors in decreasing order, for example, for 3 we obtain the colist $Succ\ (Cons\ 2\ (Succ\ (Cons\ 1\ (Succ\ (Cons\ 0\ Zero)))))$. To retrieve values from the memo-table we use

$$\begin{array}{l} lookup_\infty :: (\mathbb{N}, \mathsf{Nat}_\infty\ v) \to Maybe\ v \\ lookup_\infty\ (0,\ \ \ Cons\ a\ \_) \qquad\quad = Just\ a \\ lookup_\infty\ (n+1, Cons\ a\ (Zero)) \quad = Nothing \\ lookup_\infty\ (n+1, Cons\ a\ (Succ\ as)) = lookup_\infty\ (n, as)\ \ . \end{array}$$

Table look-up amounts to an accu-hylo where the memo-table is the accumulator and the recursive coalgebra is $in^\circ : \mathbb{N} \to \mathsf{Nat}\ \mathbb{N}$.

From the specification of *knapsack* we extract an algebra (named the same to emphasise their relationship, but in a different font) $\mathsf{knapsack}$ by making a case distinction on the structure of the naturals, and replacing boundary guards with pattern matching.

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

The effect of $lookup_\infty\ (i, table)$ is to return the result that was computed $i+1$ 'steps before' the current point of call. Thus, the key difference to the specification is that recursive calls with *absolute* indexing, *knapsack i*, are replaced by *relative* table look-ups, $lookup_\infty\ (c-1-i, table)$ where c is the current input. The call $knapsack\ (c-w)$ is turned into $lookup_\infty\ (c-1-(c-w), table) = lookup_\infty\ (w-1, table)$, which works out nicely as the original input c is actually not available to the algebra!

Now, if we capture Equation (3.13) as a higher-order function

$$\begin{array}{l} dyna :: (Functor\ f) \Rightarrow (f\ (f_\infty\ a) \to a) \to (c \to f\ c) \to (c \to a) \\ dyna\ a\ c = \mathsf{head} \cdot h\ \mathbf{where}\ h = \mathsf{cons} \cdot (a \vartriangle id) \cdot fmap\ h \cdot c\ \ , \end{array}$$

then the efficient implementation of knapsack is a breeze.

$$knapsack = dyna\ \mathsf{knapsack}\ in^\circ$$

From now on we just present the algebra and omit this obvious step.

Catalan numbers The Catalan numbers determine how many distinct well-formed arrangements can be made with a given number of pairs of matching parentheses, and are given by a simple recursive equation:

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

This works by summing the results of splitting the parentheses at all possible points, and multiplying the results of independent solutions—an operation technically known as a convolution.

This is an instructive example since it illustrates a deficiency of the recursion scheme: the upper bound for i depends on the input parameter, but the input is not available to the algebra. We present a work-around here, and approach the problem in a more principled way in Section 5. The basic idea is to store the arguments in the memo-table: we use $(\mathsf{Nat}_\mathbb{N})_\infty$, where F_C is defined in Haskell

$$\mathbf{type}\ f_c\ x = (c, f\ x)\ \ .$$

Instead of the coalgebra in° we use $id \vartriangle in^\circ : \mathbb{N} \to \mathsf{Nat}_\mathbb{N}\ \mathbb{N}$. We show in Section 5 that $id \vartriangle in^\circ$ is indeed recursive. The algebra $\mathsf{catalan}$ can then be written as follows:

$$\begin{array}{l} \mathsf{catalan} :: \mathsf{Nat}_\mathbb{N}\ ((\mathsf{Nat}_\mathbb{N})_\infty\ \mathbb{N}) \to \mathbb{N} \\ \mathsf{catalan}\ (n, Zero) \qquad\ = 1 \\ \mathsf{catalan}\ (n, Succ\ table) = sum\ (zipWith\ (*)\ xs\ (reverse\ xs)) \\ \quad \mathbf{where}\ xs = take_\infty\ (n, table)\ \ . \end{array}$$

The key here is that the algebra knows about the current depth of its application, which is held in n. Thus, the appropriate number of values can be extracted from the memo-table, and convolved to form a solution. The function $take_\infty :: (\mathbb{N}, (\mathsf{Nat}_c)_\infty\ v) \to [v]$, which accomplishes the first step, has a straightforward implementation.

We have shown that the examples can be implemented using course-of-values recursion, but there are some remaining issues. In particular, the lack of an absolute reference point in the recursion has made the expression of the algorithms somewhat inelegant. In *knapsack*, we were fortunate that the indexing was essentially relative to begin with, but this is not always the case. The root of the problem lies in the fact that the original argument of the hylo is not available to the algebra. The next section introduces a cure, which will also allow us to replace infinite by finite memo-tables.

5. Para-recursive Coalgebras

As we have seen, hylos are restrictive, since in that the algebra can in general not access the argument of the hylo. It can reconstruct the argument only in the special case that the coalgebra has a left inverse. And indeed, for initial algebras, folds and so-called paramorphisms are interdefinable [22]. For practical considerations, we ensure that inputs can always be reconstructed and focus on coalgebras of the form $id \vartriangle c$, as these have the trivial left inverse *outl*.

5.1 Basic Definitions

Para-hylomorphisms ("have your cake and eat it too") Recall that $\mathsf{F}_C\ X = C \times \mathsf{F}\ X$. Let (a, A) be an F_C-algebra, and let (C, c) be an F-coalgebra. An arrow $h : A \leftarrow C$ is a *para-hylomorphism*, or para-hylo for short, if it satisfies

$$h = a \cdot (id \vartriangle \mathsf{F}\ h \cdot c) \quad (\Longleftrightarrow \quad h = a \cdot \mathsf{F}_C\ h \cdot (id \vartriangle c))\ \ . \qquad (5.1)$$

The argument is passed both to the coalgebra c and the algebra a. (This is closely related to Haskell's @-patterns.) But note the asymmetry: c is a standard F-coalgebra, whereas a is an F_C-algebra.

Para-recursive coalgebras The F-coalgebra (C, c) is parametrically recursive, or *para-recursive* for short, if Equation (5.1) has a unique solution for each F_C-algebra (a, A). In other words, (C, c) is a para-recursive F-coalgebra if and only if $(C, id \vartriangle c)$ is a recursive F_C-coalgebra. The full subcategory of F-**Coalg**$(\mathscr{C})$ of para-recursive coalgebras is denoted F-**PRec**$(\mathscr{C})$.

A coalgebra that is para-recursive is also recursive—this is a simple application of the hylo-shift law (3.10) with $outr : \mathsf{F}_C \xrightarrow{\cdot} \mathsf{F}$. The converse holds if c has a left inverse. Then the para-hylo is equivalent to a zygo-hylo. Consequently, the recursive coalgebra $(\mu\mathsf{F}, in^\circ)$ is also para-recursive.

Apo-hylomorphisms ("stop'n'go") Dualizing the notion of para-hylos, we replace the product in the source type of the algebra by a coproduct in the target type of the coalgebra. This allows the coalgebra to stop the recursion early (cf Example 3.1). Dual to F_C we define $\mathsf{F}^A\ X = A + \mathsf{F}\ X$. Let (a, A) be an F-algebra, and let (C, c) be an F^A-coalgebra. An arrow $h : A \leftarrow C$ is an *apo-hylomorphism*,

or apo-hylo for short, if it satisfies

$$h = (id \triangledown a \cdot \mathsf{F}\ h) \cdot c \quad (\Longleftrightarrow \quad h = (id \triangledown a) \cdot \mathsf{F}^A\ h \cdot c) \qquad (5.2)$$

Apo-corecursive algebras The F-algebra (a,A) is *apo-corecursive* (aka completely iterative), if Equation (5.2) has a unique solution for each F^A-coalgebra (C,c).

Next we adapt the rolling and conjugate rules to the new setting. Recall that both rules capture perfectly symmetric situations, showing that certain coliftings preserve recursiveness and, dually, certain liftings preserve corecursiveness. Now the symmetry is broken, in that we need two separate proofs. For reasons of space, we concentrate on para-recursive coalgebras.

5.2 Para-Rolling Rule

Our goal is to show that $x = a \cdot (id \vartriangle \mathsf{L}\ (\mathsf{R}\ x) \cdot \mathsf{L}\ c) : A \leftarrow \mathsf{L}\ C$, where $a : \mathsf{L}\ C \times \mathsf{L}\ (\mathsf{R}\ A) \to A$ and $c : C \to \mathsf{R}\ (\mathsf{L}\ C)$, has a unique solution if c is para-recursive. We take the same approach as in Section 3.2 and write the right-hand side of the equation as a composition of two transformations: $x = \lceil \lfloor x \rfloor \rceil$ where

$$\lfloor x \rfloor = \mathsf{R}\ x \cdot c \qquad\qquad \lceil y \rceil = a \cdot (id \vartriangle \mathsf{L}\ y)\ . \qquad (5.3)$$

Clearly, solutions of $x = \lceil \lfloor x \rfloor \rceil$ are in 1-1 correspondence to solutions of $y = \lfloor \lceil y \rceil \rfloor$. However, it is not immediate that the latter equation is actually a para-hylo equation. Let's calculate.

$$\begin{array}{ll} & y = \lfloor \lceil y \rceil \rfloor \\ \Longleftrightarrow & \{\ \text{definition of } \lfloor - \rfloor \text{ and } \lceil - \rceil\ (5.3)\ \} \\ & y = \mathsf{R}\ (a \cdot (id \vartriangle \mathsf{L}\ y)) \cdot c \\ \Longleftrightarrow & \{\ \textit{assumption:}\ \mathsf{R} \text{ is zippable, see (5.4) below}\ \} \\ & y = \mathsf{R}\ a \cdot \mathsf{zip} \cdot (\mathsf{R}\ id \vartriangle \mathsf{R}\ (\mathsf{L}\ y)) \cdot c \\ \Longleftrightarrow & \{\ \text{product fusion (2.6b)}\ \} \\ & y = \mathsf{R}\ a \cdot \mathsf{zip} \cdot (c \times id) \cdot (id \vartriangle \mathsf{R}\ (\mathsf{L}\ y) \cdot c) \end{array}$$

The last equation is indeed a para-hylo equation. However, we have to assume that R-structures are *zippable*: that there exists a natural transformation $\mathsf{zip} : \mathsf{R}\ A \times \mathsf{R}\ B \dot{\to} \mathsf{R}\ (A \times B)$ such that $\mathsf{zip} \cdot (id \vartriangle id) = \mathsf{R}\ (id \vartriangle id)$. Because of naturality this implies

$$\mathsf{zip} \cdot (\mathsf{R}\ f \vartriangle \mathsf{R}\ g) = \mathsf{R}\ (f \vartriangle g)\ . \qquad (5.4)$$

Most but not all functors are zippable, for instance, Id, $P+$, and $P\times$ are zippable but $+$ is not. We have established

Theorem 5.1. *Assume the data in* (3.2a)*. If* R *is zippable, then*

$$\underline{\mathsf{L}} : (\mathsf{L} \circ \mathsf{R})\text{-}\mathbf{PRec}(\mathscr{C}) \leftarrow (\mathsf{R} \circ \mathsf{L})\text{-}\mathbf{PRec}(\mathscr{D})\ . \qquad (5.5)$$

5.3 Final and Cofree Para-recursive Coalgebras

Theorem 5.2. *The para-recursive coalgebra* (C,c) *is final if and only if* c *is an isomorphism.*

Proof. The proof is exactly the same as for recursive coalgebras (Theorem 3.2), except that the para-rolling rule is used to show that $\mathsf{F}\ c$ is para-recursive if c is (Id is trivially zippable). □

Corollary 5.3. *The final para-recursive* F*-coalgebra is* $(\mu\mathsf{F}, in^\circ)$.

Just as final coalgebras can be generalized to cofree coalgebras, final para-recursive coalgebras can be generalized to *cofree para-recursive coalgebras*, which we write $\mathsf{Cofree}_*\ B = (\mathsf{G}_*\ B, \mathsf{tail}_*\ B)$. The functor Cofree_* arises as the right adjoint of the forgetful functor $\mathsf{U}_* : \mathsf{G}\text{-}\mathbf{PRec}(\mathscr{C}) \to \mathscr{C}$.

$$\mathscr{C} \underset{\mathsf{Cofree}_*}{\overset{\mathsf{U}_*}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathsf{G}\text{-}\mathbf{PRec}(\mathscr{C})$$

We have characterized G_∞ as the type of possibly infinite trees whose branching structure is determined by G. Roughly speaking, G_* is the finite counterpart of G_∞. So the counit $\epsilon = \mathsf{head}_*$ extracts the root label of a tree, and $\eta\ (A,a) = \llbracket a \rrbracket_*$ constructs a finite tree.

$$\mathsf{G}_*\ B \cong \mu\ X\ .\ B \times \mathsf{G}\ X \qquad\qquad \mathsf{G}_\infty\ B \cong \nu\ X\ .\ B \times \mathsf{G}\ X$$

The proof of the isomorphism on the left is almost the same as the one for the isomorphism on the right (Example 2.4). We prove that $(\mathsf{T}\ B, \mathsf{tail}_*\ B)$ with $\mathsf{head}_*\ B : \mathsf{T}\ B \to B$ is a cofree para-recursive G-coalgebra if and only if $(\mathsf{T}\ B, \mathsf{head}_*\ B \vartriangle \mathsf{tail}_*\ B)$ is a final para-recursive G_B-coalgebra. It suffices to show that $h : A \to \mathsf{T}\ B$ with $\mathsf{head}_*\ B \cdot h = hd$ is a G-coalgebra homomorphism if and only if it is a G_B-coalgebra homomorphism.

$$\begin{array}{ll} & \mathsf{head}_* \cdot h = hd \ \wedge\ \mathsf{tail}_* \cdot h = \mathsf{G}\ h \cdot tl \\ \Longleftrightarrow & \{\ \text{see Example 2.4}\ \} \\ & (\mathsf{head}_* \vartriangle \mathsf{tail}_*) \cdot h = \mathsf{G}_A\ h \cdot (hd \vartriangle tl) \end{array}$$

As the arrows now range over a sub-category, there is one additional proof obligation: we have to show that tl is a para-recursive G-coalgebra if and only if $hd \vartriangle tl$ is a para-recursive G_B-coalgebra.

Lemma 5.4. *Let* $f : C \to D$ *and let* (C,c) *be an* F*-coalgebra. Then*

$$(C,c) : \mathsf{F}\text{-}\mathbf{PRec}(\mathscr{C}) \quad \Longleftrightarrow \quad (C, f \vartriangle c) : \mathsf{F}_D\text{-}\mathbf{PRec}(\mathscr{C})\ . \qquad (5.6)$$

Proof. "$\Longrightarrow$": First of all, note that $\mathsf{F}_C = (C \times -) \circ \mathsf{F}$ and

$$(\mathsf{F}_D)_C = (C \times -) \circ (D \times -) \circ \mathsf{F} \cong ((C \times D) \times -) \circ \mathsf{F} = \mathsf{F}_{C \times D}\ .$$

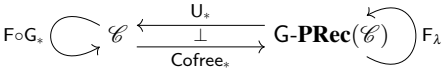
The isomorphism is witnessed by $\alpha : (X \times Y) \times Z \cong X \times (Y \times Z)$, which is natural in Z, that is, $\alpha : \mathsf{F}_{C\times D} \dot{\to} (\mathsf{F}_D)_C$. The proof then amounts to a simple application of the hylo-shift law (3.10) with $\alpha \cdot ((id \vartriangle f) \times id) : (\mathsf{F}_D)_C \dot{\leftarrow} \mathsf{F}_{C\times D} \dot{\leftarrow} \mathsf{F}_C$.

"$\Longleftarrow$": Again, we apply the hylo-shift law (3.10), this time with $C \times outr : \mathsf{F}_C \dot{\leftarrow} (\mathsf{F}_D)_C$. □

The characterization of G_* implies that $\mathsf{head}_*\ B \vartriangle \mathsf{tail}_*\ B$ is an isomorphism; we use $\mathsf{cons}_*\ B$ to denote its inverse.

5.4 Course-of-Values Recursion Revisited: $\mathsf{U}_* \dashv \mathsf{Cofree}_*$

When we discussed course-of-values recursion in Section 3.4.5 we noted the oddity that a potentially infinite structure is used for tabulation, even though only recursive coalgebras are involved. The oddity turned into a nuisance for the examples in Section 4, as many desirable functions such as *reverse* cannot be written for coinductive types. Using G_* we can remedy the problem. As usual, let us first consider the categories involved.



The data functor is now a functor over $\mathsf{G}\text{-}\mathbf{PRec}(\mathscr{C})$. Coliftings F_λ do not automatically preserve para-recursiveness, so this entails a *proof obligation* for the user of the scheme (see also below). The rest is, however, routine. Equation (3.9) specializes to

$$x = a \cdot \mathsf{F}\ (\mathsf{G}_*\ x \cdot \llbracket d \rrbracket_*) \cdot c\ \ :\ \ A \leftarrow C\ ,$$

where $a : \mathsf{F}\ (\mathsf{G}_*\ A) \to A$ and $c : (C,d) \to \mathsf{F}_\lambda\ (C,d)$. As an instance of the conjugate rule, Lemma 3.5 is applicable, and we record

Theorem 5.5. *Let* $((C,d),c)$ *be an* F_λ*-coalgebra, then*

$$(C,c) : \mathsf{F}\text{-}\mathbf{Rec}(\mathscr{C}) \quad \Longrightarrow \quad (C, \mathsf{F}\ \llbracket d \rrbracket_* \cdot c) : (\mathsf{F} \circ \mathsf{G}_*)\text{-}\mathbf{Rec}(\mathscr{C})\ .$$

The special case where we identify F and G also works out nicely. The para-rolling rule instantiated to $\mathsf{L}, \mathsf{R} := \mathsf{G}, \mathsf{Id}$ implies that G_{id} preserves para-recursiveness. Consequently,

$$x = a \cdot \mathsf{G}\ (\mathsf{G}_*\ x \cdot \llbracket c \rrbracket_*) \cdot c\ \ :\ \ A \leftarrow C \qquad (5.7)$$

has a unique solution if $c : C \to \mathsf{G}\ C$ is *para*-recursive. The coalgebra c has to be para-recursive, because it is used as an argument of $\llbracket - \rrbracket_*$. The algebra $a : \mathsf{G}\ (\mathsf{G}_*\ A) \to A$ now receives a G-structure of *finite* memo-tables that record the entire computational history.

5.5 Para-Conjugate Rule

Right adjoints preserve limits, in particular, binary products. Recall that the functor R preserves products if $\mathsf{R}\ (B_1 \times B_2)$ with R *outl* and R *outr* is a product of $\mathsf{R}\ B_1$ and $\mathsf{R}\ B_2$. This implies that there is a natural isomorphism $\iota : \mathsf{R}\ B_1 \times \mathsf{R}\ B_2 \cong \mathsf{R}\ (B_1 \times B_2)$ such that

$$\iota \cdot (\lfloor f_1 \rfloor \vartriangle \lfloor f_2 \rfloor) = \lfloor f_1 \vartriangle f_2 \rfloor \ , \tag{5.8}$$

for all $f_i : A \to \mathsf{R}\ B_i$. Because of RAPL the conjugate rule can be adapted without further ado.

Theorem 5.6. *Assume the data in* (3.4a)*. Then*

$$\mathsf{L}_\sigma : \mathsf{F}\text{-}\mathbf{PRec}(\mathscr{C}) \leftarrow \mathsf{G}\text{-}\mathbf{PRec}(\mathscr{D}) \ . \tag{5.9}$$

Proof. Adopting the calculation for Theorem 3.4 we reason

$$\begin{array}{ll} & x = a \cdot (id \vartriangle \mathsf{F}\ x \cdot \mathsf{L}_\sigma\ c) \ \ : \ \ A \leftarrow \mathsf{L}\ C \\ \iff & \{ \text{ definition of colifting (2.2) } \} \\ & x = a \cdot (id \vartriangle \mathsf{F}\ x \cdot \sigma\ C \cdot \mathsf{L}\ c) \\ \iff & \{ \ \lfloor - \rfloor \text{ and } \lceil - \rceil \text{ are isomorphisms (2.3) } \} \\ & \lfloor x \rfloor = \lfloor a \cdot (id \vartriangle \mathsf{F}\ x \cdot \sigma\ C \cdot \mathsf{L}\ c) \rfloor \\ \iff & \{ \ \lfloor - \rfloor \text{ is natural (2.4a) } \} \\ & \lfloor x \rfloor = \mathsf{R}\ a \cdot \lfloor id \vartriangle \mathsf{F}\ x \cdot \sigma\ C \cdot \mathsf{L}\ c \rfloor \\ \iff & \{ \ \mathsf{R} \text{ preserves products (5.8) } \} \\ & \lfloor x \rfloor = \mathsf{R}\ a \cdot \iota \cdot (\lfloor id \rfloor \vartriangle \lfloor \mathsf{F}\ x \cdot \sigma\ C \cdot \mathsf{L}\ c \rfloor) \\ \iff & \{ \ \lfloor - \rfloor \text{ is natural (2.4a) } \} \\ & \lfloor x \rfloor = \mathsf{R}\ a \cdot \iota \cdot (\lfloor id \rfloor \vartriangle \lfloor \mathsf{F}\ x \cdot \sigma\ C \rfloor \cdot c) \\ \iff & \{ \ \sigma \dashv \tau \text{ conjugates (2.9b) } \} \\ & \lfloor x \rfloor = \mathsf{R}\ a \cdot \iota \cdot (\lfloor id \rfloor \vartriangle \tau\ A \cdot \mathsf{G}\ \lfloor x \rfloor \cdot c) \\ \iff & \{ \text{ product fusion (2.6b) } \} \\ & \lfloor x \rfloor = \mathsf{R}\ a \cdot \iota \cdot (\lfloor id \rfloor \times \tau\ A) \cdot (id \vartriangle \mathsf{G}\ \lfloor x \rfloor \cdot c) \ \ : \ \ \mathsf{R}\ A \leftarrow C \ . \end{array}$$

The proof works by pushing $\lfloor - \rfloor$ to work on the hylo x. □

The conjugate rule allows for a unified treatment of most if not all recursion schemes. Now we feel the benefit of this unified approach: using the para-conjugate rule, generalizing the different schemes to para-hylos is a piece of cake. Table 1 summarizes our findings. There is only one exception: course-of-values recursion. This particular instance of the conjugate rule relates hylos in the ambient category to hylos in the category of coalgebras. However, for para-hylos we need products, but these do not necessarily exist in categories of coalgebras. Fortunately, there is an easy way out. We postpone the treatment of course-of-values recursion until Section 5.6 and first turn our attention to the easy cases.

In their most general form, the recursion schemes build on the canonical control functor. For the induced conjugate pair $\mathsf{L} \circ \mathsf{D} \circ \eta \dashv \eta \circ \mathsf{D} \circ \mathsf{R}$, the para-hylo equation becomes

$$x = a \cdot (id \vartriangle \mathsf{L}\ (\mathsf{D}\ \lfloor x \rfloor \cdot c)) \ \ : \ \ A \leftarrow \mathsf{L}\ C \ ,$$

where $a : \mathsf{C}_C\ A \to A$ and $c : C \to \mathsf{D}\ C$. Unravelling the definitions, $\mathsf{C} = \mathsf{L} \circ \mathsf{D} \circ \mathsf{R}$ and $\mathsf{F}_C\ X = C \times \mathsf{F}\ X$, the algebra a is assigned the impressive type $\mathsf{L}\ C \times \mathsf{L}\ (\mathsf{D}\ (\mathsf{R}\ A)) \to A$. The entries in Table 1 specialize the equation above to various adjunctions. There is a slight twist for accumulators: we use $outl \vartriangle \ldots$ instead of $id \vartriangle \ldots$, otherwise a would receive P twice. (This is a cosmetic change.)

5.6 Course-of-Values Recursion Re-Revisited

It remains to show that

$$x = a \cdot (id \vartriangle \mathsf{G}\ (\mathsf{G}_\infty\ x \cdot \llbracket c \rrbracket) \cdot c) \ \ : \ \ A \leftarrow C \tag{5.10}$$

has a unique solution if c is para-recursive. The idea for the proof is inspired by the treatment of Catalan numbers in Section 4: we integrate the arguments of x into the memo-table using $(\mathsf{G}_C)_\infty$.

To convert between different types of memo-tables we make use of a natural transformation that changes the base functor. Let $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$, then $\alpha_\infty : \mathsf{F}_\infty \dot{\to} \mathsf{G}_\infty$ is given by

$$\alpha_\infty = \mathsf{cons} \cdot (id \times \alpha \circ \alpha_\infty) \cdot (\mathsf{head} \vartriangle \mathsf{tail}) \ .$$

Since cons is corecursive, α_∞ is uniquely defined. (If all the necessary cofree coalgebras exist, $(-)_\infty : \mathscr{C}^{\mathscr{C}} \to \mathscr{C}^{\mathscr{C}}$ can be turned into a *higher-order* functor, whose action on arrows is given by α_∞.) The transformation enjoys an attractive property, termed *functor fusion*:

$$\alpha_\infty\ C \cdot \llbracket c \rrbracket = \llbracket \alpha\ C \cdot c \rrbracket \ . \tag{5.11}$$

The proof is left as the obligatory exercise to the reader. Using functor fusion we can rewrite Equation (5.10) into the standard form for course-of-values recursion (3.12).

$$\begin{array}{ll} & x = a \cdot \mathsf{G}_C\ (\mathsf{G}_\infty\ x \cdot \llbracket c \rrbracket) \cdot (id \vartriangle c) \\ \iff & \{ \text{ functor fusion (5.11) and } outr \cdot (f \vartriangle g) = g \ \} \\ & x = a \cdot \mathsf{G}_C\ (\mathsf{G}_\infty\ x \cdot outr_\infty \cdot \llbracket id \vartriangle c \rrbracket) \cdot (id \vartriangle c) \\ \iff & \{ \ outr_\infty : (\mathsf{G}_C)_\infty \dot{\to} \mathsf{G}_\infty \text{ is natural } \} \\ & x = a \cdot \mathsf{G}_C\ (outr_\infty \cdot (\mathsf{G}_C)_\infty\ x \cdot \llbracket id \vartriangle c \rrbracket) \cdot (id \vartriangle c) \\ \iff & \{ \ \mathsf{G}_C \text{ is a functor } \} \\ & x = a \cdot \mathsf{G}_C\ outr_\infty \cdot \mathsf{G}_C\ ((\mathsf{G}_C)_\infty\ x \cdot \llbracket id \vartriangle c \rrbracket) \cdot (id \vartriangle c) \end{array}$$

The results of Section 3.4.5 imply that the latter equation has a unique solution if $id \vartriangle c$ is recursive, that is, if c is para-recursive.

As an aside, the algebra $a \cdot \mathsf{G}_C\ outr_\infty$ discards a lot of useful information. Noting that $(\mathsf{G}_C)_\infty\ A \cong \mathsf{G}_\infty\ (C \times A)$, the algebra has type $C \times \mathsf{G}\ (\mathsf{G}_\infty\ (C \times A)) \to A$: it receives the original argument and a memo-table that pairs each prior argument with its result, which has potentially useful applications.

Using an entirely analogous argument we can also show that the variant that uses a finite memo-table

$$x = a \cdot (id \vartriangle \mathsf{G}\ (\mathsf{G}_*\ x \cdot \llbracket c \rrbracket_*) \cdot c) \ \ : \ \ A \leftarrow C$$

has a unique solution if c is para-recursive. Only the reason why the equations are uniquely defined changes: we essentially replace the statement "... is unique as the algebra a is corecursive" by "... is unique as the coalgebra c is para-recursive". Consider the base transformer $\alpha_* : \mathsf{F}_* \dot{\to} \mathsf{G}_*$ given by

$$\alpha_* = \mathsf{cons}_* \cdot (id \times \alpha \circ \alpha_*) \cdot (\mathsf{head}_* \vartriangle \mathsf{tail}_*) \ .$$

This equation uniquely defines α_* as tail_* is para-recursive. Using the analogue of functor fusion, $\alpha_*\ C \cdot \llbracket c \rrbracket_* = \llbracket \alpha\ C \cdot c \rrbracket_*$, we have

$$\begin{array}{ll} & x = a \cdot \mathsf{G}_C\ (\mathsf{G}_*\ x \cdot \llbracket c \rrbracket_*) \cdot (id \vartriangle c) \\ \iff & \{ \text{ see above } \} \\ & x = a \cdot \mathsf{G}_C\ outr_* \cdot \mathsf{G}_C\ ((\mathsf{G}_C)_*\ x \cdot \llbracket id \vartriangle c \rrbracket_*) \cdot (id \vartriangle c) \ . \end{array}$$

Note that $id \vartriangle c$ is used as an argument to $\llbracket - \rrbracket_*$. By Lemma (5.4) we know that $id \vartriangle c$ is para-recursive if c is.

6. Application: Dynamic Programming II

Catalan numbers revisited We have laboured hard to make the implementation of dynamic programming algorithms simpler, and now we can reap the rewards. For the Catalan numbers we replace Nat_∞, the type of non-empty colists, by Nat_*, the type of non-empty *lists*. Since Nat_* is an inductive type, we can actually convert memo-tables into standard non-empty lists.

$$\begin{array}{l} nelist :: \mathsf{Nat}_*\ v \to [v] \\ nelist\ (Cons_*\ a\ (Zero)) \quad\ = [a] \\ nelist\ (Cons_*\ a\ (Succ\ as)) = a : nelist\ as \end{array}$$

recursion scheme	*adjunction*	*conjugates*	*para-hylo equation*	*algebra*
(hylo-shift law)	$\mathsf{Id} \dashv \mathsf{Id}$	$\alpha \dashv \alpha$	$x = a \cdot (id \vartriangle \mathsf{D}\, x \cdot \alpha\, C \cdot c)\ :\ A \leftarrow C$	$a : C \times \mathsf{D}\, A \to A$
mutual recursion	$\Delta \dashv (\times)$	ccf	$x_1 = a_1 \cdot (id \vartriangle \mathsf{D}\, (x_1 \vartriangle x_2) \cdot c)\ :\ A_1 \leftarrow C$ $x_2 = a_2 \cdot (id \vartriangle \mathsf{D}\, (x_1 \vartriangle x_2) \cdot c)\ :\ A_2 \leftarrow C$	$a_1 : C \times \mathsf{D}\, (A_1 \times A_2) \to A_1$ $a_2 : C \times \mathsf{D}\, (A_1 \times A_2) \to A_2$
accumulator	$- \times P \dashv (-)^P$	ccf	$x = a \cdot (outl \vartriangle ((\mathsf{D}\, (\curlywedge\, x) \cdot c) \times P))\ :\ A \leftarrow C \times P$	$a : C \times \mathsf{D}\, (A^P) \times P \to A$
course-of-values (§5.6)	$\mathsf{U}_\mathsf{D} \dashv \mathsf{Cofree}_\mathsf{D}$	ccf	$x = a \cdot (id \vartriangle \mathsf{D}\, (\mathsf{D}_\infty\, x \cdot \llbracket c \rrbracket) \cdot c)\ :\ A \leftarrow C$	$a : C \times \mathsf{D}\, (\mathsf{D}_\infty\, A) \to A$
finite memo-table (§5.6)	$\mathsf{U}_* \dashv \mathsf{Cofree}_*$	ccf	$x = a \cdot (id \vartriangle \mathsf{D}\, (\mathsf{D}_*\, x \cdot \llbracket c \rrbracket_*) \cdot c)\ :\ A \leftarrow C$	$a : C \times \mathsf{D}\, (\mathsf{D}_*\, A) \to A$

Table 1. Different types of para-hylos building on the canonical control functor (ccf); the coalgebra is $c : C \to \mathsf{D}\, C$ in each case.

This allows us to frame *catalan* as an instance of (5.7), where the coalgebra is simply $in^\circ : \mathbb{N} \to \mathsf{Nat}\, \mathbb{N}$, and the algebra implements a convolution. Using the hylo-shift law with $\mathsf{Nat} \circ nelist$ the algebra can actually work directly on lists.

$$\begin{aligned}
&\mathsf{catalan} :: \mathsf{Nat}\, [\mathbb{N}] \to \mathbb{N}\\
&\mathsf{catalan}\ (Zero) \quad\ = 1\\
&\mathsf{catalan}\ (Succ\ xs) = sum\ (zipWith\ (*)\ xs\ (reverse\ xs))
\end{aligned}$$

Chain matrix multiplication In *chain matrix multiplication* we must find the minimum number of scalar operations required to multiply a chain of matrices $A_0 \ldots A_n$, where each matrix A_k has dimensions given by $a_k \times a_{k+1}$. Multiplying a $p \times q$ matrix by a $q \times r$ matrix yields a $p \times r$ matrix, costing (we assume) pqr scalar operations. Matrix multiplication is associative, of course, but different parenthesisations can lead to different costs.

The recurrence equation that solves this problem works by considering all the different splits, and minimizing the combined cost:

$$\begin{aligned}
&chain :: (\mathbb{N}, \mathbb{N}) \to \mathbb{N}\\
&chain\ (i,j) \mid i == j = 0\\
&\qquad\quad \mid i < j\ = minimum\ [a_i * a_{k+1} * a_{j+1} +\\
&\qquad\qquad\qquad chain\ (i,k) + chain\ (k+1,j) \mid k \leftarrow [i\,..\,j-1]]\ .
\end{aligned}$$

The final answer for this is held in $chain\ (0, n-1)$, where n is the number of matrices that are being multiplied.

This is quite unlike previous examples, since the input type is not immediately inductive. To work around this, we can show that it is isomorphic to an inductive type. Some bounds checking reveals that the domain of *chain* is actually a subset of $(\mathbb{N}, \mathbb{N})$, since the function is only defined when $0 \leqslant i \leqslant j$. Thus, the algebra needs access to only a triangle of previous values that can be represented as a set of pairs $\mathbb{T} = \{(i,j) \mid 0 \leqslant i \leqslant j\}$. It is easy to show that there is an isomorphism $tri : \mathbb{N} \cong \mathbb{T} : tri^\circ$ between the set of triangle pairs and the natural numbers, and this gives us that $(tin, \mathbb{T})$ is initial, where $tin = tri \cdot in \cdot Nat\ tri^\circ$. Thus, the coalgebra tin° is corecursive, and with appropriate choice of *tri*, can be given by:

$$\begin{aligned}
&tin^\circ :: \mathbb{T} \to \mathsf{Nat}\, \mathbb{T}\\
&tin^\circ\ (0,0) \qquad\quad\ = Zero\\
&tin^\circ\ (i,j) \mid i == j \quad = Succ\ (0, j-1)\\
&\qquad\quad\ \mid otherwise = Succ\ (i+1, j)\ .
\end{aligned}$$

Here we record an efficient version of tri° that is based on the formula for triangle numbers, $T(n) = \sum_{i=1}^{n} i = n(n+1)/2$:

$$\begin{aligned}
&tri^\circ :: \mathbb{T} \to \mathbb{N}\\
&tri^\circ\ (i,j) = j * (j+1)\ `div`\ 2 + j - i\ .
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{chain} :: (\mathbb{T}, \mathsf{Nat}\, (\mathsf{Nat}_*\, \mathbb{N})) \to \mathbb{N}\\
&\mathsf{chain}\ ((i,j), Zero) = 0\\
&\mathsf{chain}\ ((i,j), Succ\ table)\\
&\quad \mid i == j = 0\\
&\quad \mid i < j\ = minimum\ [a_i * a_{k+1} * a_{j+1} + u + v \mid k \leftarrow [i\,..\,j-1],\\
&\qquad Just\ u \leftarrow [extract\ (i,k)], Just\ v \leftarrow [extract\ (k+1,j)]]\\
&\quad \textbf{where}\ extract\ (r,s) = lookup_*\ (tri^\circ\ (i,j) - tri^\circ\ (r,s) - 1, table)
\end{aligned}$$

This definition closely mirrors the specification given by *chain*, except that rather than being recalculated, results are now extracted from the lookup table using $lookup_* :: (\mathbb{N}, \mathsf{Nat}_*\ a) \to Maybe\ a$, which, like its coinductive counterpart, is an accu-hylo. Although we are certain to have a unique solution, not all proof obligations are discharged: naturally, the correctness relies on whether the appropriate elements are indexed in the intermediate table.

The two examples have shown that the arguments provided to the algebra by para-recursive coalgebras are particularly convenient for dynamic programming algorithms with tricky indices.

7. Related Work

Recursive coalgebras The study of recursive coalgebras goes back to the work of Osius [25] on categorical set theory, where he showed that every well-founded coalgebra of the powerset functor is recursive. Taylor [27] generalized this result to set functors that preserve inverse images. Adámek et al. [1] further demonstrated that for finitary set functors preserving inverse images, recursive coalgebras are equivalent to both parametrically recursive coalgebras and to the existence of homomorphisms into the initial algebra. Completely iterative algebras are dual to parametrically recursive coalgebras, and were investigated by Milius [24], where we can glean the dual of some of the technical material in Section 5. Backhouse and Doornbos [2] worked on reductivity of recursive *relational* coalgebras, including applications to hylo equations.

Rolling rule The origins of the rolling rule can be traced back at least to the work of Freyd [9] on algebraically complete categories, which was later extended by Backhouse et al. [3] to form the categorical fixed-point calculus. However, they only considered algebras and algebra-homomorphisms. Eppendahl [7] analyzed Freyd's proof of the Iterated Square Lemma and noticed that an adjunction-like correspondence formed a core part of the proof. (He calls the correspondence a *pro-adjunction*, as hylomorphisms form a profunctor.) He further generalized (a weak form of) the Square Lemma to recursive coalgebras.

Conjugate rule An early instance of the conjugate rule can be found in the work of Bird and Paterson [5]. In order to show that generalized folds are uniquely defined, they discuss conditions to ensure that the equation $x \cdot \mathsf{L}\, in = \Psi\, x$ (or equivalently, $x = \Psi\, x \cdot$

$\mathsf{L}\ in^\circ$) uniquely defines x. Two solutions are provided to this problem, the second of which requires L to have a right adjoint.

Technically our work is closest to the seminal paper by Capretta et al. [6] on using recursive coalgebras as a tool for structured recursion, which is essentially a generalization of results in [28, 29, 31]. Ironically, they introduce the conjugate rule, but do not use it to full effect, relying on comonads and distributive laws instead. We apply the conjugate rule more liberally, leading to substantially shorter proofs. We can do so because we use adjunctions rather than comonads as our building blocks: adjunctions are a more flexible concept as they can be easily composed.

Capretta et al. instantiate their theory to two main examples: the product comonad, which gives rise to zygo-hylos, and the cofree comonad, which gives rise to course-of-values recursion. We saw in Section 3.4.3 that zygo-hylos are a special case of mutual recursion from the adjunction $\Delta \dashv (\times)$. Section 3.4.5 demonstrated that course-of-values recursion arises out of the adjunction $\mathsf{U}_\mathsf{G} \dashv \mathsf{Cofree}_\mathsf{G}$, based on F_λ-recursiveness, which is more general. They also discuss the so-called co-solution theorem, which corresponds to our development of para-recursive course-of-values recursion in Section 5.6. Similarly, the work of Uustalu and Vene [30] discusses the comonad G_*, which we covered in detail in Section 5.4. As an aside, zygo-hylos precisely fall out of the adjunction $\mathsf{U}_Z \dashv \mathsf{P}_Z : \mathscr{C} \rightharpoonup \mathscr{C} \downarrow Z$, between $\mathscr{C}$ and the slice category $\mathscr{C} \downarrow Z$, where the pairing functor P_Z is right adjoint to the forgetful functor U_Z. In fact, whatever the comonad N, we can apply the machinery of the Eilenberg–Moore construction and use the adjunction $\mathsf{U}_\mathsf{N} \dashv \mathsf{Cofree}_\mathsf{N} : \mathscr{C} \rightharpoonup \mathscr{C}_\mathsf{N}$ to integrate the ensuing comonadic recursion scheme into our framework. Such an approach mimics the techniques of Hinze et al. [13], which further shows that our theory based on adjunctions subsumes the use of comonads, but not the other way round.

For brevity, we have not shown all interesting adjunctions. For example, *natural* hylomorphisms between parametric datatypes arise from the adjunction $(- \circ \mathsf{P}) \dashv \mathsf{Ran}_\mathsf{P}$ between pre-composition and the right Kan extension, covering generalized folds [5].

Recursion schemes In this paper we have paid particular attention to course-of-values recursion, which was first captured as *histomorphisms* [29]. This scheme was identified as an instance of comonadic recursion in [31]. The adaption of histomorphisms for use in dynamic programming was worked out in detail by [16], although in an algebraically compact setting. This was later modified to the setting of recursive coalgebras in [12]. Both of these works focus on the efficiency of these schemes.

8. Conclusion

We believe that this work is a significant step towards a unifying theory of datatype-generic programming. It is quite amazing that all of the well-studied structured recursion schemes can be framed as instances of a single construction, conjugate hylomorphisms, aptly justifying the title "mother of all structured recursion schemes".

References

[1] J. Adámek, D. Lücke, and S. Milius, "Recursive coalgebras of finitary functors," *RAIRO-Theoretical Informatics and Applications*, vol. 41, no. 04, pp. 447–462, 2007.

[2] R. Backhouse and H. Doornbos, "Datatype-generic termination proofs," *Theory of Comput. Sys.*, vol. 43, no. 3-4, pp. 362–393, 2008.

[3] R. Backhouse, M. Bijsterveld, R. van Geldrop, and J. van der Woude, "Categorical fixed point calculus," in *Category Theory and Computer Science*, ser. Lect. Notes Comput. Sc., vol. 953. Springer, Aug. 1995, pp. 159–179.

[4] R. Bird and O. de Moor, *Algebra of Programming*. Prentice Hall Europe, 1997.

[5] R. Bird and R. Paterson, "Generalised folds for nested datatypes," *Form. Asp. Comput.*, vol. 11, no. 2, pp. 200–222, 1999.

[6] V. Capretta, T. Uustalu, and V. Vene, "Recursive coalgebras from comonads," *Inform. Comput.*, vol. 204, no. 4, pp. 437–468, 2006.

[7] A. Eppendahl, "Coalgebra-to-algebra morphisms," *Electronic Notes in Theoretical Computer Science*, vol. 29, no. 0, pp. 42–49, 1999.

[8] M. M. Fokkinga, "Tupling and mutumorphisms," *The Squiggolist*, vol. 1, no. 4, pp. 81–82, Jun. 1990.

[9] P. Freyd, "Algebraically complete categories," in *Category Theory*, ser. Lect. Notes Math. Springer, 1991, vol. 1488, pp. 95–104.

[10] T. Hagino, "Category theoretic approach to data types," Ph.D. dissertation, University of Edinburgh, 1987.

[11] R. Hinze, "Adjoint folds and unfolds—an extended study," *Sci. Comput. Program.*, Aug. 2012.

[12] R. Hinze and N. Wu, "Histo- and dynamorphisms revisited," in *Workshop on Generic Program.* ACM, 2013, pp. 1–12.

[13] R. Hinze, N. Wu, and J. Gibbons, "Unifying structured recursion schemes," in *International Conference on Functional Programming*. ACM, 2013, pp. 209–220.

[14] C. A. R. Hoare, "Notes on data structuring," in *Structured Programming*, ser. APIC studies in data processing. Academic Press, 1972, pp. 83–174.

[15] Z. Hu, H. Iwasaki, and M. Takeichi, "Deriving structural hylomorphisms from recursive definitions," in *International Conference on Functional Programming*. ACM, 1996, pp. 73–82.

[16] J. Kabanov and V. Vene, "Recursion schemes for dynamic programming," in *Math. of Program Construction*. Springer, 2006, pp. 235–252.

[17] D. M. Kan, "Adjoint functors," *Trans. Am. Math. Soc.*, vol. 87, no. 2, pp. 294–329, 1958.

[18] J. Lambek, "A fixpoint theorem for complete categories," *Math. Z.*, vol. 103, pp. 151–161, 1968.

[19] S. Mac Lane, *Categories for the Working Mathematician*, 2nd ed., ser. Graduate Texts in Mathematics. Springer, 1998.

[20] G. Malcolm, "Algebraic data types and program transformation," Ph.D. dissertation, University of Groningen, 1990.

[21] ——, "Data structures and program transformation," *Sci. Comput. Program.*, vol. 14, no. 2–3, pp. 255–280, 1990.

[22] L. Meertens, "Paramorphisms," *Form. Asp. Comput.*, vol. 4, pp. 413–424, 1992.

[23] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," in *Functional Programming Languages and Computer Architecture*, ser. Lect. Notes Comput. Sc., vol. 523. Springer, 1991, pp. 124–144.

[24] S. Milius, "Completely iterative algebras and completely iterative monads," *Inform. Comput.*, vol. 196, no. 1, pp. 1–41, 2005.

[25] G. Osius, "Categorical set theory: A characterization of the category of sets," *J. Pure Appl. Algebra*, vol. 4, no. 1, pp. 79–119, 1974.

[26] A. Pardo, "Generic accumulations," in *IFIP TC2 Working Conference on Generic Program.*, vol. 243. Kluwer Academic Publishers, Jul. 2002, pp. 49–78.

[27] P. Taylor, *Practical Foundations of Mathematics*, ser. Cambridge Studies in Adv. Math. Cambridge University Press, 1999, no. 59.

[28] T. Uustalu and V. Vene, "Coding recursion à la Mendler," in *Workshop on Generic Program.*, Jul. 2000, pp. 69–85.

[29] ——, "Primitive (co)recursion and course-of-value (co)iteration, categorically," *Informatica, Lith. Acad. Sci.*, vol. 10, no. 1, pp. 5–26, 1999.

[30] ——, "The recursion scheme from the cofree recursive comonad," *Electronic Notes in Theoretical Computer Science*, vol. 229, no. 5, pp. 135–157, 2011, Math. Structured Functional Program. 2008.

[31] T. Uustalu, V. Vene, and A. Pardo, "Recursion schemes from comonads," *Nordic J. Comput.*, vol. 8, pp. 366–390, Sep. 2001.

[32] V. Vene and T. Uustalu, "Functional programming with apomorphisms (corecursion)," *Proceedings of the Estonian Academy of Sciences: Physics, Mathematics*, vol. 47, no. 3, pp. 147–161, 1998.