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:

\[
\begin{align*}
h \cdot \text{in} &= a \cdot F \cdot h \\
\text{out} \cdot k &= F \cdot k \cdot c
\end{align*}
\]

which are shaped by a base functor \( F \). Categorically, these recursion schemes arise from constructions in which \( \text{in} \) and \( \text{out} \) are initial algebras and final coalgebras respectively. Initiality and finality 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 \textit{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 hylo-morphisms, 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 \( F : C \to C \), a so-called \textit{base functor}, an \( F \)-algebra is a pair \((a, A), \) where \( a : F A \to A \) is an arrow and \( A : C \) is an object, which are known respectively as the \textit{action} and \textit{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 \((a, A)\) and \((h, B)\) is an arrow \( h : A \to B : C \) such that \( h \circ a = b \circ F h \).

\[
\begin{aligned}
F A & \xrightarrow{F b} F B \\
a & \downarrow \quad \downarrow b \\
A & \xrightarrow{h} B
\end{aligned}
\]

\( F \)-homomorphisms compose, and there is an identity, so \( F \)-algebras and \( F \)-homomorphisms form a category, which we call \( \textit{F-Alg}(C) \).

The initial object of this category, if it exists, is given by \((\text{in}, \mu F)\) and called the \textit{initial F-algebra}. Initiality implies that to each \( F \)-algebra \((a, A)\), there corresponds a unique \( F \)-homomorphism, \( \lambda : (\text{in}, \mu F) \to (a, A) \), called a \textit{fold}. The algebra in is, in fact, an isomorphism, so \( \mu F \) 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 \( G : C \to C \), a \( G \)-coalgebra is a pair \((c, C)\), where \( C : C \) is the carrier and \( c : C \to G C \) is the action of the coalgebra. A \( G \)-homomorphism between coalgebras \((c, C)\) and \((d, D)\) is an arrow \( h : C \to D : C \) that satisfies \( G h \cdot c = d \cdot h \). Just as before, a category \( \textit{G-Coalg}(C) \) can be formed from \( G \)-coalgebras and \( G \)-homomorphisms. The final object of this category, if it exists, is given by \((\nu G, \text{out})\) and called the \textit{final G-coalgebra}. The unique homomorphism to the \( G \)-algebra \((c, C)\), called an \textit{unfold}, is written \( \overline{c} : C \to \nu G \).

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.

\textbf{Example 2.1.} The semantics of the inductive datatype \textit{Tree} defined

\[
\text{data Tree} = \text{Empty} \mid \text{Node Tree Z Tree}
\]

is given by the initial algebra \((\text{in}, \mu \text{Tree})\), where the \textit{base functor}

\[
\text{data Tree } = \text{Empty} \mid \text{Node Tree Z Tree}
\]

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

\[ \lambda : H \circ F \overset{\sim}{\rightarrow} G \circ H, \text{allowing us to define } H^\lambda \text{ as a lifting:} \]

\[
H^\lambda (a, A) = (H a \cdot \lambda, A, H A) \\
H^\lambda h = H h . \quad (2.1)
\]

Since we use the action of an algebra to refer to the algebra itself, we often abbreviate \( H a \cdot \lambda \) by \( H^\lambda a \).

\[ \text{Dually, } H : \text{Coalg}(\mathcal{E}) \to \text{Coalg}(\mathcal{D}) \text{ is a colifting of } H : \mathcal{E} \to \mathcal{D} \text{ iff } U G \circ H = H \circ U F . \]

Given \( \lambda : H \circ F \overset{\sim}{\rightarrow} G \circ H \) we can define a colifting as follows:

\[
H^\lambda (c, C) = (H C \cdot \lambda C \cdot H c) \\
H^\lambda h = H h . \quad (2.2)
\]
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 “Adjoints functors arise everywhere.” Our work supports this view: adjunctions provide a unified framework for program transformation.

Given categories \( \mathcal{C}, \mathcal{D} \), we say that functors \( L : \mathcal{C} \to \mathcal{D} \) and \( R : \mathcal{D} \to \mathcal{C} \) form an adjunction, written \( L \dashv R : \mathcal{C} \rightleftarrows \mathcal{D} \) or

\[
\begin{array}{c}
\mathcal{C} \\
\downarrow \quad \downarrow \\
\mathcal{D} \\
\uparrow \quad \uparrow \\
\mathcal{D} \\
\end{array}
\]

iff there is a bijection between the sets of arrows

\[
\{ - \} : \mathcal{C}(L A, B) \cong \mathcal{D}(A, R B) : \{ - \}
\]

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 \( \{ - \} \) is called the left adjunct, and its inverse \( \{ - \} \) the right adjunct.

That the adjoints [−] and [−] are mutually inverse can be captured using an equivalence:

\[
f = [g] \iff [f] = g \quad , \tag{2.3}
\]

for all \( f : L A \to B : \mathcal{C} \) and \( g : A \to R B : \mathcal{D} \). The naturality properties of the adjunctions are alternatively expressed as fusion laws.

\[
\begin{align*}
R k \cdot [f] \cdot h &= [k \cdot f \cdot L h] \quad \text{(2.4a)}
\end{align*}
\]

An alternative definition of adjunction is based on the natural transformations \( \epsilon = [id] \) and \( \eta = [id] \), which are called the counit \( \epsilon : L \circ R \to \text{Id} \) and the unit \( \eta : \text{Id} \to R \circ L \) of the adjunction. All in all, an adjunction consists of six entities: two functors, two adjoints, and two units. Each can be defined in terms of the others:

\[
\begin{align*}
[f] &= R k \cdot [g] \quad \epsilon = [id] \\
\eta &= [id] \quad R k = [k \cdot e A] \quad . \tag{2.5}
\end{align*}
\]

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

\[
\begin{array}{ccc}
\mathcal{C} & \xleftarrow{(+)} & \mathcal{C} \\
\Delta & \downarrow & \mathcal{C} \times \mathcal{C} \\
\downarrow & \Delta & \downarrow (\times) \\
\mathcal{C} & \xrightarrow{(-\times)} & \mathcal{C}
\end{array}
\]

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 adjoint \([f_1, f_2] = f_1 \times f_2 \) is known as the ‘split’ combinator, and the counit \( c = (\text{outl, outr}) \) arises from the projections.

\[
(f_1, f_2) = (\text{outl, outr}) \times A g \iff f_1 \times f_2 = g \quad . \tag{2.6a}
\]

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

\[
(k_1 \times k_2) \cdot (f_1 \times f_2) \cdot h = (k_1 \cdot f_1 \cdot h) \cdot (k_2 \cdot f_2 \cdot h) \quad . \tag{2.6b}
\]

Both laws will be frequently used in calculations.\( \square \)

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.

\[
\begin{array}{c}
\mathcal{C} \\
\downarrow \quad \downarrow \\
\mathcal{C} \times P \\
\uparrow \quad \uparrow \\
\mathcal{C} \\
\end{array}
\]

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

\[
\Lambda : \mathcal{C}(A \times P, B) \cong \mathcal{C}(A, B^P) \quad .
\]

In Set, the object \( B^P \) is the set of total functions from \( P \) to \( B \).\( \square \)

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

Example 2.4. Possibly infinite trees with branching structure determined by \( G \) and labels drawn from \( A \) constitute the cofree \( G \)-coalgebra \( \text{Cofree}_G \), which arises as the right adjoint of the underlying functor \( U_G \).

\[
\begin{array}{c}
\mathcal{C} \\
\downarrow \quad \downarrow \\
G \text{-Coalg}(\mathcal{C}) \\
\uparrow \quad \uparrow \\
\mathcal{C} \\
\end{array}
\]

The action of \( \text{Cofree}_G \) \( A = (G_0, A, tail) \) maps a tree to a \( G \)-structure of subtrees. The count \( e = head \) extracts the label of the root of a tree, and \( \eta (A, a) = [a] \) 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:

\[
f = \text{head} \cdot B \cdot U_G g \iff \text{Cofree}_G f \cdot [a] = g \quad . \tag{2.7}
\]

for all \( f : U_G (A, a) \to B \) and \( g : (A, a) \to \text{Cofree}_G B \).

Since \( \text{Cofree}_G \) is a right adjoint, final coalgebras arise as a special case (RAPL): \( (\forall G, out) \cong \text{Cofree}_G 1 \).

Conversely, cofree coalgebras can be implemented in terms of final coalgebras: \((T, B, tail)\) with head \( B \) : \( T \to B \) is a cofree \( G \)-coalgebra if and only if \((T, head B, tail)\) is a final \( G_B \)-coalgebra where \( G_B X = B \times G X \). It suffices to show that \( h : A \to T \) with 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:

\[
\begin{align*}
\text{head} \cdot B \cdot h &= hd \land \text{tail} \cdot B \cdot h &= G \cdot h \land tl \\
\iff & \{ \triangle \text{ is an isomorphism (2.6a)} \} \\
\text{head} \cdot B \cdot h \land \text{tail} \cdot B \cdot h &= hd \land G h \land tl \\
\iff & \{ \text{product fusion (2.6b)} \} \\
(\text{head} \cdot B \cdot tail) \cdot B \cdot h &= hd \land G h \land tl \\
\iff & \{ \text{definition of } G_B \text{ and product fusion (2.6b)} \} \\
(\text{head} \cdot B \cdot tail) \cdot B \cdot h &= G_B \cdot h \land (hd \land tl)
\end{align*}
\]

Note that every \( G_B \)-coalgebra is of the form \( hd \land tl \), since \( c = \text{outl} \cdot c \land \text{outr} \cdot c \). Consequently,

\[
G_B X \cong v X \cdot B \times G X \quad . \tag{2.8}
\]

This implies that head \( B \cdot tail \) is an isomorphism; we use cons \( B \) to denote its inverse. As an aside, (2.8) motivates the nomenclature: \( G \cdot B \) 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 \( G = \text{Id} \).

\( \square \)

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 \( L \dashv R : \mathcal{C} \rightleftarrows \mathcal{D} \) and \( L' \dashv R' : \mathcal{C}' \rightleftarrows \mathcal{D}' \), and functors \( H : \mathcal{C} \to \mathcal{C}' \) and \( K : \mathcal{D} \to \mathcal{D}' \), the natural transformations \( \sigma : L' \circ H \to L \circ \text{outl} \) and \( \tau : K \circ R' \to R \circ \text{outr} \) are conjugates, written \( \sigma \sim \tau \), if one of

\[
[H f \cdot \sigma A] = \tau B \cdot K [f] \quad , \tag{2.9a}
\]

\[
[H \cdot \sigma A] = \tau B \cdot K [g] \quad , \tag{2.9b}
\]

holds, for all \( f : L A \to B : \mathcal{C} \) and \( g : A \to R B : \mathcal{D} \). 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 \( F : \mathcal{E} \to \mathcal{E} \) be an endofunctor, let \((a, A)\) be an \(F\)-algebra, and let \((C, c)\) be an \(F\)-coalgebra. An arrow \( h : A \rightarrow C \) is a hylomorphism (or algebra-from-coalgebra homomorphism), \( h : (a, A) \rightarrow (C, c) \), if it satisfies

\[
h = a \cdot F \cdot h \cdot c.
\]  
(3.1)

The control functor \( F \) governs the recursive call structure. We use \((a, A) \rightarrow (C, c)\) or \((C, c) \rightarrow (a, A)\) 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 iteration. Tail-recursive programs are captured by

\[
h = (id \land id) \cdot (A + h) \cdot c \quad \iff \quad h = (id \land h) \cdot c,
\]

where \(id \land id : A + A \rightarrow A\) is the so-called co-diagonal.

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.

Recursion Coalgears A coalgebra \((C, c)\) is recursive (or algebraic-initial) if for every algebra \((a, A)\) there is a unique hylo \((a, A) \rightarrow (C, c)\). We denote such hylos by \(a \mapsto c\) 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 \( h : (a, A) \rightarrow (b, B) \) yields again a unique hylo, \( h \cdot (a \mapsto c) = (h \circ b \mapsto c) \). The full subcategory of \( F\)-Coalgebras of recursive coalgebras is denoted \( F\)-Rec(\( \mathcal{E} \))

Example 3.2. Lambek’s Lemma states that the action of the initial algebra \((in, H)\) has an inverse, which implies

\[
h \cdot in = a \cdot F \cdot h \iff h = a \cdot F \cdot h \cdot in^a.
\]

Consequently, \((\mu F, in)\) is recursive and \(\delta_0 = \delta_{a \mapsto in^a}\).

Corecursive Algebras Dually, an algebra \((a, A)\) is corecursive (or coalgebra-final) if for every coalgebra \((C, c)\) there is a unique hylo \((a, A) \rightarrow (C, c)\). We denote such hylos by \((\alpha \mapsto c)\) to emphasize the source of this uniqueness. The full subcategory of \( F\)-Algebras of corecursive algebras is denoted \( F\)-Corec(\( \mathcal{E} \)).

For example, the algebras \((our^a, vG)\) and \(\delta_0, \delta_B G, B\) are corecursive (recall that \(\delta_B B = (\text{head } B \triangle \text{tail } B)^c\)).

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 coalgears.

3.2 Rolling Rule

We now consider algebras and coalgears 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 coalgears and corecursive algebras from old ones. We start with the situation described in the following diagram.

\[
\begin{array}{ccc}
L \circ R \rightarrow \mathcal{E} & \rightarrow & (R \circ L) \rightarrow \mathcal{E} \\
\cup_{L \circ R} & \downarrow & \cup_{R \circ L} \\
\mathcal{E} & \rightarrow & \mathcal{E}
\end{array}
\]

\[ (L \circ R) - \text{Alg}(\mathcal{E}) \longrightarrow (R \circ L) - \text{Alg}(\mathcal{E}) \]

\[
\begin{array}{ccc}
L \circ R & \rightarrow & (R \circ L) \\
\cup_{L \circ R} & \downarrow & \cup_{R \circ L} \\
\mathcal{E} & \rightarrow & \mathcal{E}
\end{array}
\]

\[ (L \circ R) - \text{Coalg}(\mathcal{E}) \leftarrow (R \circ L) - \text{Coalg}(\mathcal{E}) \]

Since the base functors are compositions, the functor \(L\) has a trivial bifunctor to categories of coalgears. Recall that we need a natural transformation of type \(L \circ (R \circ L) \rightarrow (R \circ L) \circ L\); the identity will do nicely: \(L = L_{id}\), and, likewise \(R = R_{id}\).

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

\[
\begin{array}{ccc}
L \circ (C, c) & \rightarrow & (R \circ L) \circ (a, A) \\
\cup_{L \circ R} & \downarrow & \cup_{R \circ L} \\
(C, c) & \rightarrow & (R \circ L) \circ (a, A)
\end{array}
\]

\[
\begin{array}{ccc}
L \circ (C, c) & \rightarrow & (R \circ L) \circ (a, A) \\
\cup_{L \circ R} & \downarrow & \cup_{R \circ L} \\
(C, c) & \rightarrow & (R \circ L) \circ (a, A)
\end{array}
\]

natural in \((C, c) : (R \circ L) - \text{Coalg}(\mathcal{E})\) and \((a, A) : (L \circ R) - \text{Alg}(\mathcal{E})\).

Proof. The correspondence is witnessed by the functions \([-\cdot -]\) and \([-\cdot -]\), defined \([x] = R \circ c\) and \([y] = a \cdot L \circ y\). These trivially form a bijection between fixed-points of \([-\cdot -]\) and fixed-points of \([-\cdot -]\):

\[
x = [x] \iff y = [y],
\]

where \(x = [x]\) and \(y = [y]\). Unrolling the definitions we obtain

\[
x = a \cdot L \circ (R \cdot L) \cdot c \iff y = R \circ a \cdot (L \cdot y) \circ c
\]

which shows that \(x : (L \circ (C, c) \rightarrow (a, A))\) and \(y : (C, c) \rightarrow (R \circ L) \circ (a, A)\). The proof of naturality is straightforward.

The 1-1 correspondence (3.2b) allows us to conclude that \((L \circ (C, c) \rightarrow (a, A))\) is recursive if \((C, c)\) is. Thus, the cofining \(L\) preserves recursiveness, and, dually, the lifting \(R\) preserves corecursiveness.

\[
\begin{array}{ccc}
L : (L \circ R) - \text{Rec}(\mathcal{E}) & \rightarrow & (R \circ L) - \text{Rec}(\mathcal{E}) \\
\cup_{L \circ R} & \downarrow & \cup_{R \circ L} \\
(L \circ R) - \text{Corec}(\mathcal{E}) & \rightarrow & (R \circ L) - \text{Corec}(\mathcal{E})
\end{array}
\]

3.3 Final Recursive Coalgears

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

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

The unique \(F\)-coalgebra homomorphism from the recursive coalgear \((C, c)\) to the final recursive coalgear is written \(\delta_L C,\) which emphasizes that this unfolds into a recursive coalgear.

Proof. \(\Rightarrow\): The inverse of \(c\) is the unique coalgebra homomorphism \(c^\circ = [F c, H]\)—using the rolling rule we know that \(F c\) is recursive.
if $c$ is recursive (3.3a).
\[ C \leftrightarrow c \cdot F \quad \xrightarrow{\text{Rec}} \quad \text{F-Rec}(\mathcal{D}) \]
\[ \xrightarrow{\tau} \quad [x] = \mathbb{R}^T a \cdot G \ [x] \cdot c : \text{RA} \leftarrow C \]

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

The 1-1 correspondence (3.4b) between what we call conjugate hylomorphisms implies that $L_\sigma : (C, c)$ is recursive if $(C, c)$ is. Thus, $L_\sigma$ preserves recursiveness; dually, $R^T$ preserves corecursiveness.

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

Corollary (3.3) is an important result as it allows us to freely mix folds $\mu a$ as consumers with recursive unfolds $\text{Rec}(\mathcal{D})$ 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 $F : C \to C$ and $G : \mathcal{D} \to \mathcal{D}$ when there is an adjunction between $C$ and $\mathcal{D}$. Given $L : R : C \vdash \mathcal{D}$, and conjugates $\sigma : \circ G \Rightarrow F \circ L \vdash \tau : G \circ R \Rightarrow R \circ F$,

we have the following situation.

\[ \text{F-Alg}(\mathcal{D}) \xrightarrow{\text{R}^T} \text{G-Alg}(\mathcal{D}) \]

Using the conjugates we colimit $L$ 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

\[ L_\sigma (C, c) \to (a, A) \cong (C, c) \to \text{R}^T (a, A), \]

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

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

\[ x = a \cdot F x \cdot L_\sigma c \quad \iff \quad [x] = \text{R}^T a \cdot G \ [x] \cdot c, \]

where $[-]$ is the left adjunct of $L \vdash R$.

\[ x = a \cdot F x \cdot L_\sigma c : A \leftarrow L C \]

\[ \iff \{ \text{definition of colifting (2.2)} \} \]

\[ x = a \cdot F x \cdot \sigma C \cdot L C \]

\[ \iff \{ [-] \text{ and } [-] \text{ are isomorphisms (2.3)} \} \]

\[ [x] = [a \cdot F x \cdot \sigma C \cdot L C] \]

\[ \iff \{ [-] \text{ is natural (2.4a)} \} \]

\[ [x] = R a \cdot [F x \cdot \sigma C] \cdot c \]

\[ \iff \{ \sigma \vdash \tau \text{ conjugates (2.9a)} \} \]

\[ [x] = R a \cdot \tau A \cdot G \ [x] \cdot c \]

\[ \iff \{ \text{definition of lifting (2.1)} \} \]

\[ [x] = \mathbb{R}^T a \cdot G \ [x] \cdot c : \text{RA} \leftarrow C \]

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

Then (3.6a) specializes to the so-called \textit{hylo-shift law} [23]:

$$\langle a \mapsto c \cdot \alpha \rangle \circ \langle a \cdot c \mapsto \alpha \rangle = \langle a \cdot \alpha \cdot c \mapsto \alpha \rangle \ ,$$

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

\subsection{Instance: Mutual Recursion: $\Delta \dashv (\times)$}

Mutual recursion describes the situation where a number of functions rely symmetrically on each other's results before proceeding to the next step.

\begin{example}[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:

- $\text{maximize :: Tree} \rightarrow \mathbb{Z}$
- $\text{minimize :: Tree} \rightarrow \mathbb{Z}$

$$\text{maximize} (\text{Empty}) = 0$$ $$\text{minimize} (\text{Empty}) = 0$$

$$\text{maximize} (\text{Node l v r}) = v + (\text{minimize height l} \cdot \text{maximize height r})$$ $$\text{minimize} (\text{Node l v r}) = v + (\text{maximize height l} \cdot \text{minimize height r}) .$$

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

\end{example}

\begin{example}[Perfect trees]
Another example of mutual recursion is where the dependence is asymmetric. Consider the function $\text{perfect :: Tree} \rightarrow \mathbb{B}$, which checks to see whether a tree is perfectly balanced.

$$\text{perfect} (\text{Empty}) = \text{True}$$ $$\text{perfect} (\text{Node l v r}) = \text{perfect l} \land \text{perfect r} \land \text{height l} = \text{height r}$$

$$\text{height} (\text{Empty}) = 0$$ $$\text{height} (\text{Node l v r}) = 1 + (\text{height l}) \cdot \text{max} (\text{height r})$$

The calculation of $\text{perfect}$ depends on $\text{height}$, but the result of $\text{height}$ is independent of $\text{perfect}$.

\end{example}

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

\subsection{Mutu-hylo}

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

$$x = a \cdot \Delta (D [x] \cdot c) \iff \begin{cases} x_1 = a_1 \cdot D (x_1 \triangle x_2) \cdot c : A_1 \leftarrow C \\ x_2 = a_2 \cdot D (x_1 \triangle x_2) \cdot c : A_2 \leftarrow C . \end{cases}$$

The algebras $a_1 : \text{D} (A_1 \times A_2) \rightarrow A_1$ and $a_2 : \text{D} (A_1 \times A_2) \rightarrow A_2$ 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 \textit{zygo-hylo}, where only one of the algebras is dependent on both results: $a_2 := a_2' \cdot D \text{outr}$.

\subsection{Banana-split}

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

$$\text{id} : \Delta \circ D = (D \times D) \circ \Delta \dashv \tau : D \circ (\times) \rightarrow (\times) \circ (D \times D) .$$

Note that $\tau = D \text{out} \triangle D \text{outr}$. Then Equation (3.8) specializes to

$$x = a \cdot (D \times D) x \cdot \Delta_d c \iff \begin{cases} x_1 = a_1 \cdot D x_1 \cdot c : A_1 \leftarrow C \\ x_2 = a_2 \cdot D x_2 \cdot c : A_2 \leftarrow C . \end{cases} .$$

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

$$(a_1 \leftarrow c) \triangle (a_2 \leftarrow c) = (a_1 \cdot D \text{out} \triangle a_2 \cdot D \text{outr} \leftarrow c) ,$$

as $(\times)^\tau (a_1, a_2) = a_1 \cdot D \text{out} \triangle a_2 \cdot D \text{outr}$. We obtain a generalization of the-called \textit{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 $\Delta$ is pronounced ‘split’.)

\subsection{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.

\begin{example}[Append]
The first example of recursion with a parameter is in the definition of $\text{cat}$, which takes two lists and appends them together by structural induction on the first list.

- $\text{cat} : ([a] \times [a]) \rightarrow [a]$  
- $\text{cat} ([], [y]) = yz$  
- $\text{cat} (x : xs, ys) = x : \text{cat} (xs, ys)$

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

\end{example}

\begin{example}[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.

$$\text{flattenCat} : (\text{Tree}, [\mathbb{Z}]) \rightarrow [\mathbb{Z}]$$  
$$\text{flattenCat} (\text{Empty}, x) = x$$  
$$\text{flattenCat} (\text{Node l x r}, xs) = \text{flattenCat} (l, x : \text{flattenCat} (r, xs))$$

The function spaws off two new invocations of itself, but only in the second does the parameter $x$ remain unchanged.

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

\begin{itemize}
  \item \textbf{Varying parameters} Using the canonical control functor, Equation (3.9) specializes to the \textit{accu-hylo}
  
  $$x = a \cdot ((D (A \times x) \cdot c) \times P) : A \leftarrow C \times P .$$
  
  This models the situation where the parameter varies during the recursion; $a : D (A^P) \times P \rightarrow A$ receives a $D$-structure of curried variants of $x$, and can supply each of them with a different argument.

  \item \textbf{Constant parameters} The case where the parameter is passed unchanged can be modelled using the so-called \textit{strength} of a functor:
  
  $$\sigma : (\times^P) \circ D \rightarrow D \circ (\times^P) \rightarrow \tau : D \circ (-)^P \rightarrow (-)^P \circ D .$$
  
  The strength $\sigma : D A \times P \rightarrow D (A \times P)$ broadcasts a value across a structure. (Every endofunctor on Set has a canonical strength.) For this data, Equation (3.8) specializes to
  
  $$x = a \cdot D x \cdot \sigma C : (c \times P) \rightarrow A \leftarrow C \times P .$$
  
  The strength copies the parameter to each of the recursive calls.
\end{itemize}

\subsection{Instance: Course-of-Values Recursion: $U_G \vdash \text{Cofree}_G$}

The remaining adjunction introduced in Section 2.2 that we have not yet discussed is $U_G \vdash \text{Cofree}_G$. It turns out that this instance gives us \textit{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.

\[
\begin{array}{c}
\xymatrix{F \circ \text{G-coalg} \ar[r] & \mathcal{C}' \ar[r]^u \ar[l]_{\text{Cofree}_G} & \text{G-coalg}(\mathcal{C})}
\end{array}
\]

The data functor is now a functor over \(\text{G-coalg}(\mathcal{C})\), for example, the cofree \(F \circ \text{G-coalg}\) where \(\lambda : F \circ \text{G-coalg} \rightarrow G \circ F;\) the corresponding canonical control functor is \(\text{U} \circ \text{F-cofree}_G = F \circ \text{U-cofree}_G = F \circ \text{G-coalg}\). For this data, Equation (3.9) specializes to

\[
x = a \cdot F (\text{F-cofree}_G (x \cdot c)) : A \leftarrow \text{U-cofree}_G (C, d)
\]

where \(a : F (\text{G-coalg} A) \rightarrow A \) and \(c : (C, d) \rightarrow F (\text{F-cofree}_G (x \cdot d)) \). The coalegebra \(c\) lives in \(\text{F}_2\)-\text{G-coalg}(\mathcal{C})\); it is an \(F\text{-cofree}_G\) that is simultaneously a \(G\text{-coalgebra homomorphism}.\) If we wish to apply this instance of the conjugate rule, we have to ensure that \((C, d), x\) is a \(F\text{-cofree}_G\)-coalgebra. The following lemma shows that \(F\text{-cofree}_G\)-recursiveness is actually a weaker notion than \(F\text{-recursiveness.}\)

**Lemma 3.5.** Let \(\mathcal{D}\) be a full subcategory of \(\text{G-coalg}(\mathcal{C})\), let \(F \circ \text{G-coalg} : \mathcal{D} \rightarrow \mathcal{D}\), and let \(((C, d), c) : F \circ \text{G-coalg}(\mathcal{D})\). Then

\[
(C, c) : F \circ \text{Rec}(\mathcal{C}) \implies ((C, d), c) : F \circ \text{Rec}(\mathcal{D})
\]

**Proof.** We have to show that the equation in \(h\)

\[
h = a \cdot F (\text{F-cofree}_G (x \cdot c)) : A \leftarrow \text{U-cofree}_G (C, d)
\]

has a unique solution for each \(F\text{-}G\text{-coalgebra}\) \((a, (A, b), (C, d))\). \((A, b)\) is isomorphic to a \(\lambda\)-bialgebra; \(F\text{-cofree}_G\) coalgebras have been called \(\lambda\)-dicoalgebras [6]. If \((C, c)\) is a recursive \(F\text{-cofree}_G\), we know that (3.11) has a unique solution in \(\mathcal{C}\). Since \(\mathcal{D}\) is a full subcategory of \(\text{G-coalg}(\mathcal{C})\), it suffices to show that \(h\) is a \(G\text{-coalgebra homomorphism}\) of type \((A, b) \leftarrow (C, d)\).

\[
\begin{array}{c}
G \cdot h \\
b \cdot h \\
\{ h \text{ solves (3.11) } \}
\end{array}
\]

\[
\begin{array}{c}
G \cdot a \cdot F (h) \cdot c \cdot d \\
b \cdot a \cdot F \cdot h \cdot c \\
\{ c : (C, d) \rightarrow F (\text{F-cofree}_G (C)) \}
\end{array}
\]

\[
\begin{array}{c}
G \cdot a \cdot \lambda \cdot F \cdot b \cdot F \cdot h \cdot c \\
\{ \lambda \text{ is natural } \}
\end{array}
\]

\[
G \cdot a \cdot \lambda \cdot A \cdot F \cdot b \cdot F \cdot h \cdot c
\]

Since \(x = G \cdot a \cdot \lambda \cdot A \cdot F \cdot x \cdot c\) has a unique solution as \(c\) is recursive, we can conclude that \(b \cdot h = h \cdot d\).

Combining the conjugate rule and Lemma 3.5 we obtain

**Corollary 3.6.** Let \(((C, d), c) : F \circ \text{G-coalg},\) then

\[
(C, c) : F \circ \text{Rec}(\mathcal{C}) \implies (C, \text{F-cofree}_G, c) : F \circ \text{G-coalg}(\mathcal{C})
\]

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

\[
x = a \cdot G (\text{G-coalgebra} x \cdot c \cdot c) : A \leftarrow C
\]

(3.12)

Operationally, \(G [c] \cdot c\) is tail \(C \cdot c\) 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 \(a : G (\text{G-coalgebra} A) \rightarrow A\). 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:

\[
x = \text{head} A \cdot [x] (2.7)\]

where \([x] = [a] \cdot G [x] \cdot c (3.7)\). The conjugate \((C, c) \rightarrow G\text{Id}(A, tail) A\) builds an entire memo-table bottom-up using \([a] = G\text{Id} a \cdot G (\text{tail} A)\). This is a vast improvement, but we can do better still, avoiding the costly \(G\text{Id} a\). Using the fact that \([x] = a\cdot G [x] \cdot c\) is a \(G\text{-coalgebra homomorphism},\) we reason

\[
\{ \text{Leibniz’ law and head } A \cdot [a] = [a] \}
\]

\[
\text{head} A \cdot [x] \Delta \text{tail} A \cdot [x] = a \cdot G [x] \cdot c \Delta G [x] \cdot c
\]

\[
\{ \text{product fusion (2.6b) } \}
\]

\[
\text{head} A \cdot \text{tail} A \cdot [x] = (a \cdot \text{Id}) \cdot G [x] \cdot c
\]

\[
\{ \text{head} A \cdot \text{tail} A \text{ is an isomorphism } \}
\]

\([x] = \text{cons} A \cdot (a \cdot \text{Id}) \cdot G [x] \cdot c\).

Thus, \([\text{cons} A \cdot (a \cdot \text{Id}) \rightarrow c]\) and consequently

\[
x = \text{head} A \cdot [\text{cons} A \cdot (a \cdot \text{Id}) \rightarrow c].
\]

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.

\[
x_1 = a_1 \cdot D (x_1 \Delta A \Delta x_2) \cdot c
\]

\[
x_2 = a_2 \cdot D (x_1 \Delta A \Delta x_2) \cdot (x \cdot P)
\]

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

\[(Id \times (-) \circ P) \circ \Delta = (\times) \circ (Id \times (-))^P : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C},\]

which combines pairing, \(L_1 \times L_2 \rightarrow L_1 \times L_2\), and composition of adjunctions, \(L_2 \circ \text{IdL}_1 \rightarrow \text{IdR}_1 \circ \text{IdR}_2\). (Do not be confused by the four occurrences of \(\times\): the formula involves products both in \(\text{Cat}\), eg \(\text{Id} \times (-)^P\), and in \(\mathcal{C}\), eg \(x \times P\).) 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 \(w < v\) is implicitly provided:

\[
\begin{align*}
\text{knapsack} : \mathbb{N} & \rightarrow \mathbb{R} \\
\text{knapsack} c & = \max \{ 0, v + \text{knapsack}(c - v) \mid (w, v) \leftarrow \text{vals}, 0 < w < c \}
\end{align*}
\]
The value of a knapsack is determined by finding the item in \( w vs \) that maximizes the total value while also decreasing the capacity of the knapsack. The function \( \text{maximum}_0 \) 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 \( 0 < w \land w \leq c \) ensures that the recursive calls are always on smaller arguments, this fits perfectly into the framework of Equation (3.12). The base functor is \( G := \text{Nat} \) defined

\[
\text{data Nat } \text{nat} = \text{Zero} | \text{Suc nat} ,
\]

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

\[
\text{lookup}_{\text{nat}} : (\mathbb{N}, \text{Nat} v) \rightarrow \text{Maybe } v
\]

\[
\text{lookup}_{\text{nat}} (0, \text{Cons } a) = \text{Just } a
\]

\[
\text{lookup}_{\text{nat}} (n + 1, \text{Cons } a (\text{ Zero})) = \text{Nothing}
\]

\[
\text{lookup}_{\text{nat}} (n + 1, \text{Cons } a (\text{Succ } s)) = \text{lookup}_{\text{nat}} (n, a) .
\]

Table look-up amounts to an accu-hylo where the memo-table is the accumulator and the recursive coalgebra is \( \text{in}^n : \mathbb{N} \rightarrow \text{Nat} \). From the specification of knapsack we extract an algebra (named the same to emphasise their relationship, but in a different font) knapsack by making a case distinction on the structure of the naturals, and replacing boundary guards with pattern matching.

\[
\text{knapsack} :: \text{Nat} (\text{Nat} a \ R) \rightarrow R
\]

\[
\text{knapsack} (\text{Zero}) = 0
\]

\[
\text{knapsack} (\text{Succ } \text{table}) = \text{maximum}_0 [v \cdot w | (w, v) \rightarrow \text{just } u \leftarrow [\text{lookup}_{\text{nat}} (w - 1, \text{table})] ,]
\]

The effect of \( \text{lookup}_{\text{nat}} (i, \text{table}) \) 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, \( \text{lookup}_{\text{nat}} (c - 1 - i, \text{table}) \) where \( c \) is the current input. The call \( \text{knapsack} (c - w) \) is turned into \( \text{lookup}_{\text{nat}} (c - 1 - (c - w), \text{table}) = \text{lookup}_{\text{nat}} (w - 1, \text{table}) \), 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

\[
\text{dyn} :: \text{Functor } f \Rightarrow f (f (f a a) a) \rightarrow f (f c f) \rightarrow f (f a)
\]

\[
\text{dyn } a c = \text{head } \cdot \text{h where } h = \text{cons } \cdot (\text{a } \triangle \text{h}) \cdot \text{fmap } h \cdot c ,
\]

then the efficient implementation of knapsack is a breeze.

\[
\text{knapsack } = \text{dyn } \text{knapsack} \text{in}^0
\]

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:

\[
\text{catalan} :: \mathbb{N} \rightarrow \mathbb{N}
\]

\[
\text{catalan } 0 = 1
\]

\[
\text{catalan } (n + 1) = \text{sum } [\text{catalan } i * \text{catalan } (n - i) | i \leftarrow [0 . . n] .]
\]

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 \( (\text{Nat} \mathbb{N}) \infty \), where \( F_C \) is defined in Haskell

\[
type f_c x = (c, f x) .
\]

Instead of the coalgebra \( \text{in}^n \) we use \( \text{id } \triangle \text{in}^n : \mathbb{N} \rightarrow \text{Nat} \). We show in Section 5 that \( \text{id } \triangle \text{in}^n \) is indeed recursive. The algebra catalan can then be written as follows:

\[
\text{catalan} :: \text{Nat} (\text{Nat} \mathbb{N}) \rightarrow \mathbb{N}
\]

\[
\text{catalan } (\text{Zero}) = 1
\]

\[
\text{catalan } (\text{Succ } \text{table}) = \text{sum } (\text{zipWith } * ) (\text{xs } (\text{reverse } \text{xs}))
\]

\[
\text{where } xs = \text{take } \infty (n, \text{table}) .
\]

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 \( \text{take}_\infty :: (\mathbb{N}, (\text{Nat} \mathbb{N}) v) \rightarrow [v] \), 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 \( \eta \triangle c \), as these have the trivial left inverse \( \eta \).

### 5.1 Basic Definitions

**Para-hyloforms (“have your cake and eat it too”)** Recall that \( F_C : X \rightarrow F X \). Let \( (a, A) \) be an \( F_C \)-algebra, and let \( (c, C) \) be an \( F \)-coalgebra. An arrow \( h : A \rightarrow C \) is a para-hyloform, or para-hylo for short, if it satisfies

\[
h = a \cdot (\text{id } \triangle F h \cdot c) \quad (\Leftarrow \Rightarrow \text{ h } : C \rightarrow F C \cdot h \cdot (\text{id } \triangle c)) .
\]

The argument is passed both to the coalgebra \( c \) and the algebra \( a \). (This is closely related to Haskell’s \( \otimes \)-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 \( F_C \)-coalgebra if and only if \( (C, \eta \triangle c) \) is a recursive \( F_C \)-coalgebra. The full subcategory of \( F \cdot \text{Coalg}(\bigvee') \) of para-recursive coalgebras is denoted \( F \cdot \text{PreRec}(\bigvee') \).

A coalgebra that is para-recursive is also recursive—this is a simple application of the hylo-shift law (3.10) with \( \text{outl} : F_C \rightarrow F \). The converse holds if \( c \) has a left inverse. Then the para-hylo is equivalent to a zygo-hylo. Consequently, the recursive coalgebra \( (\mu F, \eta n) \) is also para-recursive.

**Apo-hyloforms (“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 \( F_A X = A + F X \). Let \( (a, A) \) be an \( F \)-algebra, and let \( (C, c) \) be an \( F_A \)-coalgebra. An arrow \( h : A \rightarrow C \) is an apo-hyloform.
or apo-hylo for short, if it satisfies
\[ h = (\text{id} \vee a \cdot F \circ h) \cdot c \quad (\iff) \quad h = (\text{id} \vee a) \cdot F^A \circ h \cdot c \] (5.2)

**Apo-coreursive algebras** The F-algebra \((a, A)\) is apo-coreursive
(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 cofibrations 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 \(x = a \cdot (\text{id} \triangle L (R x) \cdot L c) : A \to L C\), where \(a : L C \times L (R A) \to A \quad \text{and} \quad c : C \to R (L C)\), 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: \(x = [x] \cdot y\), where
\[ [x] = R x \cdot c \quad \quad [y] = a \cdot (\text{id} \triangle L y) \] (5.3)

Clearly, solutions of \(x = [x] \cdot y\) are in 1-1 correspondence to solutions of \(y = [y]\). However, it is not immediate that the latter equation is actually a para-hylo equation. Let’s calculate.
\[ y = [y] \quad \iff \quad \{ \text{definition of } \square \text{ and } \square \} (5.3) \]
\[ y = R (a \cdot (\text{id} \triangle L y)) \cdot c \quad \iff \quad \{ \text{assumption: } R \text{ is zippable, see (5.4) below} \} \]
\[ y = R \cdot \text{zip} \cdot (R (R \cdot L (L y)) \cdot c) \quad \iff \quad \{ \text{product fusion (2.6b)} \} \]
\[ y = R \cdot a \cdot \text{zip} \cdot (R (c \cdot \text{id} \triangle L y)) \cdot c \]

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 \(\text{zip} : R A \times R B \to R (A \times B)\) such that \(\text{zip} \cdot (\text{id} \triangle \text{id}) = R (\text{id} \triangle \text{id})\). Because of naturality this implies
\[ \text{zip} \cdot (R f \triangle R g) = R (f \triangle g) \] (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
\[ L : (L \circ R)\text{-PRec}(\mathcal{E}) \leftarrow (R \circ L)\text{-PRec}(\mathcal{E}) \] (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 \(F\) is para-recursive if \(c\) is (Id is trivially zippable). \(\square\)

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

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 Cofree, \(B = (G_b, B, \text{tail}, B)\). The functor Cofree, arises as the right adjoint of the forgetful functor \(U_c : G\text{-PRec}(\mathcal{E}) \to \mathcal{E}\).
\[ \mathcal{E} \xrightarrow{U_c} G\text{-PRec}(\mathcal{E}) \xrightarrow{\text{Cofree}_c} \]

We have characterized \(G_b\) as the type of possibly infinite trees whose branching structure is determined by \(G\). Roughly speaking, \(G_b\) is the finite counterpart of \(G_a\). So the count \(c = \text{head}\), extracts the root label of a tree, and \(\text{n} (A, a) = [a]\) constructs a finite tree.
\[ G_b \cdot B \cong \mu X \cdot B \times G X \quad G_a \cong \nu X \cdot B \times G X \]

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 \(T \circ B\cdot \text{tail} \cdot B\) with \(B\), \(T \circ B \cdot B\) is a cofree para-recursive \(G\)-coalgebra if and only if \(B\cdot \text{head} \cdot B \cdot \text{tail} \cdot B\) is a final para-recursive \(G\)-coalgebra.

\[ \quad \text{head} \cdot B \cdot B \cdot h = h \cdot d \cdot t \]
\[ \iff \quad \{ \text{see Example 2.4} \} \]
\[ (\text{head} \cdot \text{tail}) \cdot h = G_a \cdot h \cdot (h \cdot d \cdot t) \]

As the arrows now range over a sub-category, there is one additional proof obligation: we have to show that \(B\) is a para-recursive \(G\)-coalgebra if and only if \(h \cdot d \cdot tl\) is a para-recursive \(G_a\)-coalgebra.

**Lemma 5.4.** Let \(f : C \to D\) and let \((C, c)\) be an F-coalgebra. Then
\[ (C, c) : F\text{-PRec}(\mathcal{E}) \iff (C \circ f : D\text{-PRec}(\mathcal{E})) \] (5.6)

**Proof.** “⇒” : First of all, note that \(F \circ C = (C \times - \circ F\) and \((F D)_C = (C \times - \circ (D \times -) \circ F \cong ((C \times D) \times -) \circ F = F \circ C \circ D\).

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

“⇐” : Again, we apply the hylo-shift law (3.10), this time with \(\text{C \times outr} : F \circ C \cong (F D)\).

The characterization of \(G_b\) implies that head, \(\text{B \cdot tail}\), \(\text{B}\) is an isomorphism; we use cons. \(B\) to denote its inverse.

### 5.4 Course-of-Values Recursion Revisited: U_c \circ 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 \(G_a\), we can remedy the problem. As usual, let us first consider the categories involved.

\[ F \circ G_b \circ (\mu F, \text{in}) \leftarrow (U_c \circ \text{Cofree}) \circ G\text{-PRec}(\mathcal{E}) \leftarrow \]

The data functor is now a functor over \(G\text{-PRec}(\mathcal{E})\). Cofibrations \(F \circ \text{Cofree}\) 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
\[ x = a \cdot F (G_b \cdot x \cdot [c]) \cdot c : A \to C \]
where \(a : F (G_b, A) \to A \quad \text{and} \quad c : (C, d) \to F \circ \text{Cofree}(\mathcal{E})\). As an instance of the conjugate rule, Lemma 3.5 is applicable, and we record

**Theorem 5.5.** Let \((C, d)\) be an \(F\circ \text{Cofree}\)-coalgebra, then
\[ (C, d) : F\text{-PRec}(\mathcal{E}) \iff (C \circ [c] : (F \circ \text{Cofree})\text{-PRec}(\mathcal{E})) \]

The special case where we identify \(F\) and \(G\) also works out nicely. The para-rolling rule instantiated to \(L, R := G\). Id implies that \(G_{a\downarrow}\) preserves para-recursiveness. Consequently,
\[ x = a \cdot G (G_b \cdot x \cdot [c]) \cdot c : A \to C \] (5.7)
has a unique solution if \(c : C \rightarrow G C\) is para-recursive. The coalgebra \(c\) has to be para-recursive, because it is used as an argument of \([-\cdot]\). The algebra \(a : G (G, A) \rightarrow A\) 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 \(R (B_1 \times B_2)\) with \(R\) outfl and \(R\) outfl is a product of \(R B_1\) and \(R B_2\). This implies that there is a natural isomorphism \(\iota : R B_1 \times R B_2 \cong R (B_1 \times B_2)\) such that

\[
\iota \cdot ([f_1] \triangle [f_2]) = [f_1] \triangle [f_2],
\]

for all \(f_i : A \rightarrow R B_i\). Because of RAPL the conjugate rule can be adapted without further ado.

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

\[
L_c : F \mathsf{PRec}(\emptyset) \leftarrow G \mathsf{PRec}(\emptyset).
\]  

**Proof.** Adopting the calculation for Theorem 3.4 we reason

\[
x = a \cdot (id \triangle x \cdot L_c), c : A \leftarrow L C
\]

\[
\iff \{\ \text{definition of colifting (2.2)}\ \}
\]

\[
x = a \cdot (id \triangle x \cdot \sigma_\mathcal{C} \cdot \iota \cdot L_c)
\]

\[
\iff \{\ \text{and}\ [-\cdot]\ \text{are isomorphisms (2.3)}\ \}
\]

\[
[x] = a \cdot (id \triangle x \cdot \sigma_\mathcal{C} \cdot \iota)
\]

\[
\iff \{\ \text{is natural (2.4a)}\ \}
\]

\[
x = R a \cdot (id \triangle x \cdot \sigma_\mathcal{C} \cdot \iota)
\]

\[
\iff \{\ \text{products products (5.8)}\ \}
\]

\[
x = R a \cdot \iota \cdot (id \triangle x \cdot \sigma_\mathcal{C} \cdot L_c)
\]

\[
\iff \{\ \text{is natural (2.4a)}\ \}
\]

\[
x = R a \cdot \iota \cdot (id \triangle x \cdot \sigma_\mathcal{C} \cdot L_c)
\]

\[
\iff \{\ \text{product fusion (2.6b)}\ \}
\]

\[
x = R a \cdot \iota \cdot (id \triangle x \cdot \sigma_\mathcal{C} \cdot L_c)
\]

The proof works by pushing \([-\cdot]\) to work on the hylo \(x\). \(\square\)

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-hyllos 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 hyllos in the ambient category to hyllos in the category of coalgebras. However, for para-hyllos 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 \(L \odot D \odot \eta \odot D \odot R\), the para-hylo equation becomes

\[
x = a \cdot (id \triangle L (D [x] \cdot c)) : A \leftarrow L C,
\]

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

### 5.6 Course-of-Values Recursion Re-Revisited

It remains to show that

\[
x = a \cdot (id \triangle G (G_m \circ \cdot [c]) \cdot c) : A \leftarrow C
\]  

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 \((G_m)_{\mathcal{C}}\).

To convert between different types of memo-tables we make use of a natural transformation that changes the base functor. Let \(\alpha : F \rightarrow G\), then \(\alpha_m : F_m \rightarrow G_m\) is given by

\[
\alpha_m = \text{cons} \cdot (id \times \alpha \circ \alpha_m) \cdot (\text{head} \triangle \text{tail}).
\]

Since cons is corecursive, \(\alpha_m\) is uniquely defined. (If all the necessary cofree coalgebras exist, \((-)_m : \mathcal{C}^\mathcal{E} \rightarrow \mathcal{C}^\mathcal{E}\) can be turned into a higher-order functor, whose action on arrows is given by \(\alpha_m\).) The transformation enjoys an attractive property, termed *functor fusion*:

\[
\alpha_m \circ [c] = [\alpha \circ c \cdot c],
\]  

(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).

\[
x = a \cdot G C (G_m x \cdot [c]) \cdot c : A \leftarrow C
\]

\[
\iff \{\ \text{functor fusion (5.11) and outfl}\ (f \triangle g) = g\}
\]

\[
x = a \cdot G C (G_m x \cdot outfl \cdot [c]) \cdot (id \triangle c)
\]

\[
\iff \{\ outfl\ : \ (G_m)_{\mathcal{C}}\ \rightarrow G_m\ \text{is natural}\}
\]

\[
x = a \cdot G C (outfl \cdot (G_m)_{\mathcal{C}} x \cdot [id \triangle c]) \cdot (id \triangle c)
\]

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

As an aside, the algebra \(a \cdot G_C\) outfl discards a lot of useful information. Noting that \((G_m)_{\mathcal{C}} A \cong G_m (C \times A)\), the algebra has type \(C \times G (G_m (C \times A))\) \(\rightarrow A\); 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

\[
x = a \cdot (id \triangle G (G_m x \cdot [c]) \cdot c) : A \leftarrow C
\]

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 \(\alpha_m : F_m \rightarrow G_m\), given by

\[
\alpha_m = \text{cons} \cdot (id \times \alpha \circ \alpha_m) \cdot (\text{head} \triangle \text{tail}).
\]

This equation uniquely defines \(\alpha_m\) as tail, is para-recursive. Using the analogue of functor fusion, \(\alpha_m \circ [c] = [\alpha \circ c \cdot c]\), we have

\[
x = a \cdot G C (G_m x \cdot [c]) \cdot c : A \leftarrow C
\]

\[
\iff \{\ \text{see above}\}
\]

\[
x = a \cdot G C (outfl \cdot G C)(G_m x \cdot [id \triangle c]) \cdot (id \triangle c)
\]

Note that \(id \triangle c\) is used as an argument to \([-\cdot]\). By Lemma (5.4) we know that \(id \triangle c\) 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 \(Nat_m\), the type of non-empty colists, by \(Nat_t\), the type of non-empty lists. Since \(Nat_t\) is an inductive type, we can actually convert memo-tables into standard non-empty lists.

\[
\text{nelist} :: Nat_t, v \rightarrow [v]
\]

\[
\text{nelist} (\text{Cons} _t (\text{Zero})) = [a]
\]

\[
\text{nelist} (\text{Cons}_t (\text{Succ} as)) = a : \text{nelist}\ as
\]

Conjugate Hylomorphisms

10

2014/7/22
The definition of the algebra chain allows us to frame the relative indices: the base case is straightforward, but when \( T \) is actually a subset of \( \mathbb{N} \) and \( \mathbb{N} \) the natural numbers, this gives us that the isomorphism (in the intermediate table).

**Table 1.** Different types of para-hylos building on the canonical control function (ccf); the algebra is \( c : C \to D C \) in each case.

<table>
<thead>
<tr>
<th>recursion scheme</th>
<th>adjunction</th>
<th>conjugates</th>
<th>para-hylo equation</th>
<th>algebra</th>
</tr>
</thead>
<tbody>
<tr>
<td>(hylo-shift law)</td>
<td>( \text{Id} \to \text{Id} )</td>
<td>( x \to x )</td>
<td>( x = a \cdot (id \circ D x \cdot x \cdot C \cdot c) )</td>
<td>( a : C \times D A \to A )</td>
</tr>
<tr>
<td>mutual recursion</td>
<td>( \Delta \to \times )</td>
<td>( c \text{cf} )</td>
<td>( x_1 = a_1 \cdot (id \circ D (x_1 \triangle x_2) \cdot c) )</td>
<td>( a_1 : C \times D (A_1 \times A_2) \to A_1 )</td>
</tr>
<tr>
<td>accumulator</td>
<td>(- \times P + (-)P^\text{c} )</td>
<td>( c \text{cf} )</td>
<td>( x = a \cdot (out \cdot \triangle ((D (A x) \cdot c) \cdot P)) )</td>
<td>( a : C \times D (A^P \times P) \to A )</td>
</tr>
<tr>
<td>course-of-values</td>
<td>( U_\text{D} \to \text{Cofree}_D )</td>
<td>( c \text{cf} )</td>
<td>( x = a \cdot (id \circ D (D_x \cdot \text{const} \cdot c) \cdot c) )</td>
<td>( a : C \cdot D (D_\text{A} A) \to A )</td>
</tr>
<tr>
<td>finite memo-table</td>
<td>( U_\text{A} \to \text{Cofree}_A )</td>
<td>( c \text{cf} )</td>
<td>( x = a \cdot (id \circ D (D_x \cdot \text{const} \cdot c) \cdot c) )</td>
<td>( a : C \cdot D (D_x A) \to A )</td>
</tr>
</tbody>
</table>

\( \text{Chain matrix multiplication} \) 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 \( pqr \) 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:

\[
\begin{align*}
\text{chain} & : (N \times N) \to N \\
\text{chain} (i, j) & | i \leq j = 0 \\
& | i < j = \text{minimum} |a_i \cdot a_{i+1} \cdot a_{i+1} + \text{chain} (i, k) + \text{chain} (k + 1, j) | k \leftarrow |i..j-1| .
\end{align*}
\]

The final answer for this is held in \( \text{chain} (0, n - 1) \), and \( n \) 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 \( \text{chain} \) is actually a subset of \( [N, N] \), since the function is only defined when \( 0 \leq i \leq j \). Thus, the algebra needs access to only a triangle of previous values that can be represented as a set of pairs \( \mathbb{T} = \{(i, j) \mid 0 \leq i \leq j \} \). It is easy to show that there is an isomorphism \( \text{tri} : \mathbb{N} \times \mathbb{N} \to \mathbb{T} \): \( \text{tri} \) between the set of triangle pairs and the natural numbers, and this gives us that \( (\text{ini}, T) \) is initial, where \( \text{ini} = \text{tri} \cdot \text{in} \cdot \text{Nat tri} \). Thus, the coalgebra \( \text{tri} \) is corecursive, and with appropriate choice of \( \text{tri} \), can be given by:

\[
\begin{align*}
\text{tri} & : \mathbb{T} \to \text{Nat} \mathbb{T} \\
\text{tri} (0, 0) & = \text{Zero} \\
\text{tri} (i, j) & | i \leq j = \text{Succ} (0, j - 1) \\
& | \text{otherwise} = \text{Succ} (i + 1, j) .
\end{align*}
\]

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

\[
\begin{align*}
\text{tri} & : \mathbb{T} \to \mathbb{N} \\
\text{tri} (i, j) & = j \cdot (j + 1) \cdot \text{div} 
\end{align*}
\]

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.

\[
\begin{align*}
\text{chain} & : (\mathbb{T}, \text{Nat} (\text{Nat} \times \mathbb{N})) \to \mathbb{N} \\
\text{chain} ((i, j), \text{Zero}) & = 0 \\
\text{chain} ((i, j), \text{Succ} (\text{table})) & = 0 \\
& | i < j = \text{minimum} |a_i \cdot a_{i+1} \cdot a_{i+1} + \text{chain} (i, k) + \text{chain} (k + 1, j) | k \leftarrow |i..j-1| .
\end{align*}
\]

This definition closely mirrors the specification given by \( \text{chain} \), except that rather than being recalculated, results are now extracted from the lookup table using \( \text{lookup}_a : (\mathbb{N}, \text{Nat}, a) \to \text{Maybe} \ a \), 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 Osuiz [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 Milunis [24], where we can glean the dual of some of the technical material in Section 5. Backhouse and Doornbos [2] worked on reducivity 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 algebraic 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 \textit{pro-adjunction}, as hylomorphisms form a profunctor.) He further generalized (a weak form of) the Square adjunction-like correspondence formed a core part of the proof. (He

**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 \( x \cdot \text{L in} = \Psi x \) (or equivalently, \( x = \Psi x \cdot \text{L out} \))...
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-hylors, and the cofree comonad, which gives rise to course-of-values recursion. We saw in Section 3.4.3 that zygo-hylors 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 $U_G \dashv \text{Cofree}_G$, based on $P_Z$-recursiveness, which is more general. They also discuss the so-called co-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 $G_N$, which we covered in detail in Section 5.4. As an aside, zygo-hylors precisely fall out of the adjunction $U_Z \vdash \text{Cofree}_Z$, which we covered in detail in Section 3.4. This shows that the pairing function $P_Z$ 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 $U_N \vdash \text{Cofree}_N$ to integrate the ensuing comonadic recursion scheme into our framework.

For brevity, we have not shown all interesting adjunctions. For example, natural hylomorphisms between parametric datatypes arise from the adjunction $(\rightarrow P) \dashv \text{Ran} P$ 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