Recursion Schemes for Dynamic Programming

Jevgeni Kabanov[1] and Varmo Vene[2]

[1] Dept. of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia; ekabanov@gmail.com
[2] Dept. of Computer Science, University of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia; varmo@cs.ut.ee

Abstract. Dynamic programming is an algorithm design technique, which allows to improve efficiency by avoiding the re-computation on identical subtasks. We present a new recursion combinator, dynamorphism, which captures the dynamic programming recursion pattern with memoization and identify some simple conditions when functions defined by structured general recursion can be redefined as a dynamorphism. The applicability of the new recursion operator is demonstrated on classical dynamic programming algorithms: Fibonacci numbers, binary partitions, edit distance and longest common subsequence.

1 Introduction

Solutions for many problems admit a simple recursive description where the original problem is split to some subproblems, these are solved recursively and then combined to a final solution. The corresponding program can then straightforwardly expressed as a hylomorphism [9], where the decomposition of a problem is represented by some functor coalgebra and the forming of a final result by an algebra of the same functor. However, if the problem description contains identical subproblems such a naive implementation is very inefficient as these subproblems are solved independently from each other over and over again. This kind of unnecessary re-computation can often be avoided using dynamic programming techniques.

In this paper we study dynamic programming in the setting of categorical approach to recursive datatypes and constructive algorithmics [2, 7, 8, 5]. We introduce a new recursion combinator, which captures the dynamic programming recursion pattern. It is a generalization of the combinator for a course-of-value iteration, histomorphism [10, 12], and uses annotated tree-like intermediate structure to tabulate previously computed values, hence avoiding re-computation on identical subarguments. Like hylomorphism, it is parametrized by a coalgebra and an algebra but for different (albeit related) functors. We show that given a hylomorphism if its coalgebra satisfies a simple equation it can be redefined in terms of the new combinator involving a related coalgebra and reusing the algebra of the hylomorphism. The applicability of the new recursion operator is demonstrated on classical dynamic programming algorithms: Fibonacci numbers, binary partitions, edit distance and longest common subsequence.

The rest of the paper is organized as follows. Section 2 presents the setting and summarizes the basic theory of recursive datatypes and recursion combinators. Section 3 introduces two recursion combinators for dynamic programming, histomorphism and dynamorphism, and studies their properties. Section 4 provides four concrete examples from dynamic programming that can be expressed as dynamorphisms. Finally, Section 5 concludes by pointing out some directions for further work.

2 Categorical Datatypes and Recursion Combinators

In this section we briefly review the basic notions of categorical approach to recursive datatypes and its application to program calculation. For a more comprehensive and excellent introduction of the subject see e.g. [3, 6].

2.1 Preliminaries

Throughout the work we assume that the category we're dealing with is $\mathcal{CPO}$ category of (pointed) complete partial orders with a least element $\bot$ (*cpos*) and continuous partial functions between them. A function $f : A \to B$ is said to be *strict* if $f(\bot) = \bot$, i.e. it preserves the least element. The final object in $\mathcal{CPO}$ is given by the singleton set $\{\bot\}$ and is denoted by $\mathbf{1}$. The subcategory of $\mathcal{CPO}$ where all functions are strict is denoted by $\mathcal{CPO}_\bot$.

A recursive type is defined categorically using a functor fixpoint. That is given an endofunctor $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ we get the recursive type $\mu\mathsf{F}$ as the solution of the equation $\mu\mathsf{F} \simeq \mathsf{F}\mu\mathsf{F}$. This equation is solvable in $\mathcal{CPO}$ for *locally continuous* functors. We assume that the type signature is given by a combination of following basic functors: Id (identity), $\underline{\mathsf{A}}$ (constant), $\times$ (product) and $+$ (separated sum). The product bifunctor $A \times B$ is given by the Cartesian product. We denote projections by $\mathrm{outl} : A \times B \to A$ and $\mathrm{outr} : A \times B \to B$. The pairing of $f : C \to A$ and $g : C \to B$, i.e. the unique morphism $h : C \to A \times B$ such that $\mathrm{outl} \circ h = f$ and $\mathrm{outr} \circ h = g$, is written $\langle f, g \rangle$. The separated sum bifunctor $A + B$ is given by the disjoint union, with injections denoted by $\mathrm{inl} : A \to A + B$ and $\mathrm{inr} : B \to A + B$. The case analysis of $f : A \to C$ and $g : B \to C$, i.e. the unique strict $h : A + B \to C$ such that $h \circ \mathrm{inl} = f$ and $h \circ \mathrm{inr} = g$, is written $[f, g]$. In the examples, we often use also pattern matching for case analysis.

We shall also use type functors which are attained by taking a fixpoint over bifunctors (e.g. functor List that lifts set of values to set of lists of values). The functors attainable by a combination of described functors are called *regular functors* and are locally continuous in $\mathcal{CPO}$.

2.2 Catamorphisms

Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be a functor. An F-*algebra* is a pair consisting of an object A (called the carrier) and an arrow $\varphi : \mathsf{F}A \to A$. A homomorphism between algebras $\varphi : \mathsf{F}A \to A$ and $\psi : \mathsf{F}B \to B$ is an arrow $f : A \to B$ such that

$f \circ \varphi = \psi \circ \mathsf{F} f$. F-algebras and their homomorphisms form a category $\mathcal{A}lg(\mathsf{F})$ where composition and identities are inherited from the base category $\mathcal{C}$. An initial F-algebra $\mathsf{in}_\mathsf{F} : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$ is the one from which a unique homomorphism exists to any other F-algebra $\varphi : \mathsf{F}A \to A$. This unique morphism is called *catamorphism* or *fold* and is denoted by $(\!|\varphi|\!)_\mathsf{F} : \mu\mathsf{F} \to A$.

An initial algebra $\mathsf{in}_\mathsf{F} : \mathsf{F}\mu\mathsf{F} \to \mu\mathsf{F}$ (if it exists) is necessarily isomorphism (we denote its inverse by $\mathsf{out}_\mathsf{F} : \mu\mathsf{F} \to \mathsf{F}\mu\mathsf{F}$), hence its carrier $\mu\mathsf{F}$ is a solution of recursive domain equation $X \simeq \mathsf{F}X$. Unfortunately, in $\mathcal{CPO}$ initial algebras do not exist, but just weakly initial ones, i.e. the uniqueness of the outgoing homomorphism is missing. On the other hand, a locally continuous functor F, which preserves strictness, has an initial algebra in $\mathcal{CPO}_\perp$. Moreover, the catamorphism combinator can be extended to $\mathcal{CPO}$ by defining: for any F-algebra $\varphi : \mathsf{F}A \to A$, the catamorphism $(\!|\varphi|\!)_\mathsf{F}$ is a least arrow $f : \mu\mathsf{F} \to A$ making the following diagram commute:

$$\begin{tikzcd}
\mathsf{F}\mu\mathsf{F} \arrow[r, "\mathsf{in}"] \arrow[d, "\mathsf{F}f"'] & \mu\mathsf{F} \arrow[d, "f"] \\
\mathsf{F}A \arrow[r, "\varphi"] & A
\end{tikzcd}$$

i.e. the least homomorphism from in_F to φ.

Catamorphisms comes equipped with the following properties:

- **Cancellation:** For any F-algebra $\varphi : \mathsf{F}A \to A$

$$(\!|\varphi|\!)_\mathsf{F} = \varphi \circ \mathsf{F}(\!|\varphi|\!)_\mathsf{F} \circ \mathsf{out}_\mathsf{F} \qquad \text{(cata-cancellation)}$$

- **Reflection:**

$$id_{\mu\mathsf{F}} = (\!|\mathsf{in}_\mathsf{F}|\!)_\mathsf{F} \qquad \text{(cata-reflection)}$$

- **Fusion:** For any F-algebras $\varphi : \mathsf{F}A \to A$, $\psi : \mathsf{F}B \to B$ and a morphism $f : A \to B$

$$f \text{ is strict} \wedge f \circ \varphi = \psi \circ \mathsf{F}f \;\Rightarrow\; f \circ (\!|\varphi|\!)_\mathsf{F} = (\!|\psi|\!)_\mathsf{F} \qquad \text{(cata-fusion)}$$

Catamorphisms capture structural recursion over inductive types.

2.3 Anamorphisms

The notion of a coalgebra is dual to the one of an algebra. Let $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ be a functor. An F-*coalgebra* is a pair consisting of an object A and an arrow $\psi : A \to \mathsf{F}A$. A homomorphism between coalgebras $\psi : A \to \mathsf{F}A$ and $\varphi : B \to \mathsf{F}B$ is an arrow $f : A \to B$ such that $\varphi \circ f = \mathsf{F}f \circ \psi$. F-coalgebras and their homomorphisms form a category $\mathcal{C}o\mathcal{A}lg(\mathsf{F})$ where composition and identities are inherited from the base category $\mathcal{C}$. A final F-coalgebra is the one to which a unique homomorphism exists from any other F-coalgebra $\psi : A \to \mathsf{F}A$. This unique morphism is called *anamorphism* or *unfold* and is denoted by $[\!(\psi)\!]_{\mathsf{F}} : A \to \mu\mathsf{F}$.

In general, carriers of an initial F-algebra and a final F-coalgebra are different. However, in $\mathcal{CPO}$ the inverse of an initial F-algebra $\mathsf{out}_{\mathsf{F}} : \mu\mathsf{F} \to \mathsf{F}\mu\mathsf{F}$ is a final F-coalgebra. Hence, for any F-coalgebra $\psi : A \to \mathsf{F}A$, the anamorphism $[\!(\psi)\!]$ is defined as a unique arrow $f : A \to \mu\mathsf{F}$ making the following diagram commute:

$$\begin{tikzcd} A \arrow[r, "\psi"] \arrow[d, "f"'] & \mathsf{F}A \arrow[d, "\mathsf{F}f"] \\ \mu\mathsf{F} \arrow[r, "\mathsf{out}_{\mathsf{F}}"] & \mathsf{F}\mu\mathsf{F} \end{tikzcd}$$

Anamorphisms come equipped with the following properties:

- **Cancellation:** For any F-coalgebra $\psi : A \to \mathsf{F}A$

$$[\!(\psi)\!]_{\mathsf{F}} = \mathsf{in}_{\mathsf{F}} \circ \mathsf{F}[\!(\psi)\!]_{\mathsf{F}} \circ \psi \qquad \text{(ana-cancellation)}$$

- **Reflection:**

$$id_{\mu\mathsf{F}} = [\!(\mathsf{out}_{\mathsf{F}})\!]_{\mathsf{F}} \qquad \text{(ana-reflection)}$$

- **Fusion:** For any F-coalgebras $\psi : A \to \mathsf{F}A$, $\varphi : B \to \mathsf{F}B$ and a morphism $f : A \to B$

$$\varphi \circ f = \mathsf{F}f \circ \psi \;\Rightarrow\; [\!(\varphi)\!]_{\mathsf{F}} \circ f = [\!(\psi)\!]_{\mathsf{F}} \qquad \text{(ana-fusion)}$$

Anamorphisms capture structural corecursion over coinductive types.

2.4 Hylomorphisms

As in $\mathcal{CPO}$ inductive and coinductive types coincide we can also define a concept that captures general recursion. Given an F-coalgebra $\psi : A \to \mathsf{F}A$ and an F-algebra $\varphi : \mathsf{F}B \to B$, a *hylomorphism* denoted by $\mathsf{hylo}(\varphi, \psi)_{\mathsf{F}}$ is the least arrow $f : A \to B$ that makes the following diagram commute:

$$\begin{tikzcd} \mathsf{F}A \arrow[d, "\mathsf{F}f"'] & A \arrow[l, "\psi"'] \arrow[d, "f"] \\ \mathsf{F}B \arrow[r, "\varphi"] & B \end{tikzcd}$$

Equivalently, a hylomorphism is a composition of an anamorphism with a catamorphism:

$$\mathsf{hylo}(\varphi, \psi)_{\mathsf{F}} = (\!|\varphi|\!)_{\mathsf{F}} \circ [\!(\psi)\!]_{\mathsf{F}} \qquad \text{(hylo-definition)}$$

Hylomorphisms capture general recursion by producing a virtual intermediate structure and then collapsing it. The intermediate structure represents the function *call-tree* that would be otherwise produced on the stack.

Hylomorphisms come equipped with the following properties:

– **Cancellation:** For any F-algebra $\varphi : \mathsf{F}B \to B$ and F-coalgebra $\psi : A \to \mathsf{F}A$

$$\mathsf{hylo}(\varphi,\psi)_{\mathsf{F}} = \varphi \circ \mathsf{F}\mathsf{hylo}(\varphi,\psi)_{\mathsf{F}} \circ \psi \qquad \text{(hylo-cancellation)}$$

– **Cata-Fusion:** For any F-algebra $\varphi : \mathsf{F}B \to B$, $\phi : \mathsf{F}C \to C$, F-coalgebra $\psi : A \to \mathsf{F}A$ and a morphism $f : B \to C$

$$f \text{ is strict} \wedge f \circ \varphi = \phi \circ \mathsf{F}f \;\Rightarrow\; f \circ \mathsf{hylo}(\varphi,\psi)_{\mathsf{F}} = \mathsf{hylo}(\phi,\psi)_{\mathsf{F}} \qquad \text{(hylo-cata-fusion)}$$

– **Ana-Fusion:** For any F-coalgebras $\psi : \mathsf{F}A \to A$, $\xi : \mathsf{F}B \to B$, F-algebra $\varphi : \mathsf{F}C \to C$ and a morphism $f : A \to B$

$$\xi \circ f = \mathsf{F}f \circ \psi \;\Rightarrow\; \mathsf{hylo}(\varphi,\xi)_{\mathsf{F}} \circ f = \mathsf{hylo}(\varphi,\psi)_{\mathsf{F}} \qquad \text{(hylo-ana-fusion)}$$

– **Hylo-Shift:** For any F-coalgebra $\psi : A \to \mathsf{F}A$, G-algebra $\phi : B \to \mathsf{G}B$ and a natural transformation $\tau : \mathsf{F} \dot{\to} \mathsf{G}$

$$\mathsf{hylo}(\varphi \circ \tau_B,\psi)_{\mathsf{F}} = \mathsf{hylo}(\varphi,\tau_A \circ \psi)_{\mathsf{G}} \qquad \text{(hylo-shift)}$$

Further on, subscripts will be omitted for brevity when obvious from the context.

3 Combinators for Dynamic Programming

Dynamic programming is a programming technique that can be applied to some algorithms redefining them with better performance (and specially smaller complexity in terms of the input). The application of dynamic programming requires that the problem would have: a) optimal substructure, and b) overlapping subproblems.

Optimal substructure (also known as the *principle of optimality* [1]) in this context means that to solve the problem we can break it into subproblems, solve them recursively and then combine the results to solve the original problem. Overlapping here means that the subproblem results are recomputed several times, since the same sub-subproblems appear in different subproblems.

In such a case the dynamic programming method suggests to use memoization to reuse the overlapping subproblem computation results and thus reduce the complexity of the algorithm.

Note that every function expressible as a hylomorphism already has an optimal substructure, so we can apply dynamic programming technique knowing only that a hylomorphism has overlapping subproblems. Note also that in case of a hylomorphism overlapping subproblems will mean that some parts of the intermediate structure will be equal (though we do not use this fact in the paper, it helps to understand the problem background).

3.1 Histomorphisms

The simplest instance of dynamic programming problem is provided by a course-of-value iteration. As an example consider the Fibonacci function which can be specified as follows:

$$\begin{aligned} \mathit{fibo}(0) &= 0 \\ \mathit{fibo}(1) &= 1 \\ \mathit{fibo}(n+2) &= \mathit{fibo}(n+1) + \mathit{fibo}(n). \end{aligned}$$

These equations give us the definition for the function as a hylomorphism (see the next section). However this definition is of exponential complexity in terms of its input. In [10], Uustalu and Vene defined a recursion combinator called *histomorphism* which solves the problem by internally using an auxiliary tree-like structure to capture the memoization of previously computed values on substructures.

For a given endofunctor F and an object A, define a new endofunctor $\mathsf{F}^{\times}_{A}$ as follows:

$$\begin{aligned} \mathsf{F}^{\times}_{A}(X) &= A \times \mathsf{F}X \\ \mathsf{F}^{\times}_{A}(f) &= id \times \mathsf{F}f \end{aligned}$$

The $\mathsf{F}^{\times}_{A}$-functor represents one level of the structure $\mu\mathsf{F}$ with an extra annotation. Next we define a recursive type based on this functor:[3]

$$\begin{aligned} \tilde{\mathsf{F}}(A) &= \mu\mathsf{F}^{\times}_{A} \\ \tilde{\mathsf{F}}(f) &= [\![\langle f \circ \varepsilon, \theta \rangle]\!]_{\mathsf{F}^{\times}_{A}} \end{aligned}$$

where natural transformations $\varepsilon_A = \mathrm{outl} \circ \mathsf{out} : \tilde{\mathsf{F}}(A) \to A$ and $\theta_A = \mathrm{outr} \circ \mathsf{out} : \tilde{\mathsf{F}}(A) \to \mathsf{F}\tilde{\mathsf{F}}(A)$ are projections of the final coalgebra $\mathsf{out}_{\mathsf{F}^{\times}_{A}} = \langle \varepsilon_A, \theta_A \rangle : \tilde{\mathsf{F}}(A) \to A \times \mathsf{F}\tilde{\mathsf{F}}(A)$.

Intuitively, the recursive type $\tilde{\mathsf{F}}(A)$ is a datatype of F-branching trees, where every node is annotated by values of type A. In a special case, when annotations are empty, it's isomorphic to the original recursive type, i.e. $\tilde{\mathsf{F}}(\mathbf{1}) \simeq \mu\mathsf{F}$. The function $\varepsilon_A : \tilde{\mathsf{F}}(A) \to A$ gives the value of the annotation in the root node and $\theta_A : \tilde{\mathsf{F}}(A) \to \mathsf{F}\tilde{\mathsf{F}}(A)$ returns "subtrees" of the given tree.

Often, these annotated trees are constructed using anamorphisms in the form $[\![\langle f, \psi \rangle]\!]_{\mathsf{F}^{\times}_{A}} : A \to \tilde{\mathsf{F}}B$, where $\psi : A \to \mathsf{F}A$ is an F-coalgebra and $f : A \to B$ is an arrow for computing an annotation of a node. Later we make use following properties of this anamorphism:

$$\begin{aligned} \varepsilon_A \circ [\![\langle f, \psi \rangle]\!] &= f && (\varepsilon\text{-cancellation}) \\ \theta_A \circ [\![\langle f, \psi \rangle]\!] &= \mathsf{F}[\![\langle f, \psi \rangle]\!] \circ \psi && (\theta\text{-cancellation}) \\ [\![\langle f, \psi \rangle]\!] &= \tilde{\mathsf{F}}f \circ [\![\langle id, \psi \rangle]\!] && (\text{map-build-fusion}) \end{aligned}$$

[3] In [10], the functor $\tilde{\mathsf{F}}$ is defined as $\tilde{\mathsf{F}}(A) = \nu\mathsf{F}^{\times}_{A}$. However, in our setting inductive and coinductive types coincide, hence the definition uses the least fixed point.

The first two equations are obvious. The third one can be shown by using *ana-fusion* law, i.e., we need to show that $h = [\![\langle id, \psi \rangle]\!]$ is a coalgebra homomorphism:

$$\begin{tikzcd} B \times \mathsf{F}A \arrow[d, "id \times \mathsf{F}h"'] & A \arrow[l, "\langle f, \psi \rangle"'] \arrow[d, "h"] \\ B \times \mathsf{F}\tilde{\mathsf{F}}A & \tilde{\mathsf{F}}A \arrow[l, "\langle f \circ \varepsilon, \theta \rangle"'] \end{tikzcd}$$

As the following simple calculation shows, this is indeed the case:

$$\begin{aligned} \langle f \circ \varepsilon, \theta \rangle \circ [\![\langle id, \psi \rangle]\!] &= (f \times id) \circ \mathsf{out} \circ [\![\langle id, \psi \rangle]\!] \\ &= (f \times id) \circ (id \times \mathsf{F}[\![\langle id, \psi \rangle]\!]) \circ \langle id, \psi \rangle \\ &= (id \times \mathsf{F}[\![\langle id, \psi \rangle]\!]) \circ \langle f, \psi \rangle \end{aligned}$$

Definition 1. *(histomorphism)*
Let $\varphi : \mathsf{F}\tilde{\mathsf{F}}A \to A$ *be an* $\mathsf{F}\tilde{\mathsf{F}}$*-algebra. A* histomorphism*, denoted by* $\mathsf{histo}(\varphi)$*, is the least function* $f : \mu\mathsf{F} \to A$ *making the following diagram commute:*

$$\begin{tikzcd} \mathsf{F}\mu\mathsf{F} \arrow[r, "\mathsf{in}"] \arrow[d, "\mathsf{F}[\![\langle f, \mathsf{out} \rangle]\!]"'] & \mu\mathsf{F} \arrow[d, "f"] \\ \mathsf{F}\tilde{\mathsf{F}}A \arrow[r, "\varphi"] & A \end{tikzcd}$$

Informally the definition of the histomorphism tells that its value on the given argument is computed by first building an annotated F-branching tree and then using an $\mathsf{F}\tilde{\mathsf{F}}$-algebra to give the final result. The annotated tree is generated using an anamorphism, which gets the immediate subargument as the initial seed. On every step, the anamorphism computes (recursively) the value of the histomorphism on the current seed, and also a new seed by taking a "predecessor" of the current one. Thus, all nodes in the tree contain the recursively computed value of the histomorphism on the corresponding substructure of the argument. The given $\mathsf{F}\tilde{\mathsf{F}}$-algebra can use these values to produce a final result. Note that the annotated tree is recalculated on every recursive call, hence the definition above corresponds to the naive (exponential) algorithm of computing course-of-value iterative functions.

In the case of natural numbers, the base functor is $\mathsf{F}X = \mathbf{1} + X$ and annotated trees are of type $\tilde{\mathsf{F}}(A) = \mu X.A \times (\mathbf{1} + X)$, i.e. non-empty lists. The Fibonacci function can be defined as a histomorphism $\mathit{fibo} = \mathsf{histo}(\varphi)$, where the $\mathsf{F}\tilde{\mathsf{F}}$-coalgebra is defined as follows:

$$\begin{aligned} \varphi\,(\mathrm{inl}\ \bot) &= \quad 0 \\ \varphi\,(\mathrm{inr}\ x) &= \begin{cases} 1 & \text{if } \theta(x) = \mathrm{inl}\ \bot \\ \varepsilon(x) + \varepsilon(y) & \text{if } \theta(x) = \mathrm{inr}\ y \end{cases} \end{aligned}$$

Histomorphisms come equipped the following properties:

– **Cancellation:** For any $\mathsf{F}\tilde{\mathsf{F}}$-algebra $\varphi : \mathsf{F}\tilde{\mathsf{F}}A \to A$

$$\mathsf{histo}(\varphi) \circ \mathsf{in} = \varphi \circ \mathsf{F}[\!(\langle \mathsf{histo}(\varphi), \mathsf{out}\rangle)\!] \qquad \text{(histo-cancellation)}$$

– **Reflection:**

$$id = \mathsf{histo}(\mathsf{in} \circ \mathsf{F}\varepsilon) \qquad \text{(histo-reflection)}$$

– **Fusion:** For any $\mathsf{F}\tilde{\mathsf{F}}$-algebras $\varphi, \psi : \mathsf{F}\tilde{\mathsf{F}}A \to A$ and an arrow $f : A \to B$

$$f \text{ is strict} \wedge f \circ \varphi = \psi \circ \mathsf{F}\tilde{\mathsf{F}}f \;\Rightarrow\; f \circ \mathsf{histo}(\varphi) = \mathsf{histo}(\psi) \qquad \text{(histo-fusion)}$$

As noted above, the recursive definition of a histomorphism corresponds to the exponential algorithm. In [10], Uustalu and Vene show that every histomorphism can be defined in terms of a catamorphism and this alternative definition gives a more efficient algorithm for computing histomorphisms.

Proposition 1. *Let* $\varphi : \mathsf{F}\tilde{\mathsf{F}}A \to A$ *be an* $\mathsf{F}\tilde{\mathsf{F}}$*-algebra, then*

$$\mathsf{histo}(\varphi) = \varepsilon_A \circ (\!|\, \mathsf{in} \circ \langle \varphi, id\rangle \,|\!) \qquad \text{(histo-as-cata)}$$

The equation can be illustrated by the following diagram where the catamorphism $(\!|\, \mathsf{in} \circ \langle \varphi, id\rangle \,|\!)$ is denoted by f:

$$\begin{tikzcd}
\mathsf{F}\mu\mathsf{F} \arrow[rr, "\mathsf{in}"] \arrow[d, "\mathsf{F}f"'] & & \mu\mathsf{F} \arrow[d, "f"] \arrow[dr, "\mathsf{histo}(\varphi)"] & \\
\mathsf{F}\tilde{\mathsf{F}}A \arrow[r, "\langle \varphi, id\rangle"] & A \times \mathsf{F}\tilde{\mathsf{F}}A \arrow[r, "\mathsf{in}"] & \tilde{\mathsf{F}}A \arrow[r, "\varepsilon"] & A
\end{tikzcd}$$

Instead of computing a value of the histomorphism directly, the catamorphism f builds an F-branching tree, which contains values of the histomorphism on corresponding subarguments and, in particular, the final value in its root node. The tree is constructed "bottom-up" by first recursively building subtrees for the immediate subarguments, which are then combined by the $\mathsf{F}\tilde{\mathsf{F}}$-algebra to get the value of the histomorphism on the current argument and the resulting tree. Since the building of the annotated tree is done by one pass over the argument structure, the equation gives a linear algorithm for computing histomorphisms.

3.2 Dynamorphisms

Although histomorphism is enough for the Fibonacci function, it is not so for some other classic dynamic programming algorithms like edit distance. This is due to histomorphism capturing structural (course-of-value) recursion, which requires that the recursion pattern of the algorithm follows that of its input. Next we will define a combinator that will lift this restriction and allow to capture all generally recursive dynamic algorithms.

Definition 2. *(dynamorphism)*
Let $\psi : A \to \mathsf{F}A$ *be an* F*-coalgebra and* $\varphi : \mathsf{F}\tilde{\mathsf{F}}B \to B$ *be an* $\mathsf{F}\tilde{\mathsf{F}}$*-algebra. A* dynamorphism *denoted by* $\mathsf{dyna}(\varphi, \psi)_\mathsf{F}$ *is the least arrow* $f : A \to B$ *that makes the following diagram commute:*

$$\begin{tikzcd}
\mathsf{F}A \arrow[d, "\mathsf{F}\llbracket\langle f,\psi\rangle\rrbracket"'] & A \arrow[l, "\psi"'] \arrow[d, "f"] \\
\mathsf{F}\tilde{\mathsf{F}}B \arrow[r, "\varphi"] & B
\end{tikzcd}$$

Proposition 2. *Let* $\psi : A \to \mathsf{F}A$ *be an* F*-coalgebra and* $\varphi : \mathsf{F}\tilde{\mathsf{F}}B \to B$ *be an* $\mathsf{F}\tilde{\mathsf{F}}$*-algebra, then*

$$\mathsf{dyna}(\varphi, \psi)_\mathsf{F} = \mathsf{histo}(\varphi)_\mathsf{F} \circ \llbracket\psi\rrbracket_\mathsf{F} \qquad \text{(dyna-definition)}$$

Proof. We will use the fusion law of least fixed points [9]:

$$\mathcal{G} \text{ is strict } \wedge\ \mathcal{F} \circ \mathcal{G} = \mathcal{G} \circ \mathcal{H} \implies \mathsf{fix}\,\mathcal{F} = \mathcal{G}(\mathsf{fix}\,\mathcal{H})$$

In our case we have:

$$\begin{aligned}
\mathcal{F}(f) &= \varphi \circ \mathsf{F}\llbracket\langle f, \psi\rangle\rrbracket \circ \psi \\
\mathcal{G}(g) &= g \circ \llbracket\psi\rrbracket \\
\mathcal{H}(h) &= \varphi \circ \mathsf{F}\llbracket\langle h, \mathsf{out}\rangle\rrbracket \circ \mathsf{out}
\end{aligned}$$

i.e. $\mathsf{dyna}(\varphi, \psi) = \mathsf{fix}\,\mathcal{F}$ and $\mathsf{histo}(\varphi) = \mathsf{fix}\,\mathcal{H}$.

Obviously $\mathcal{G}$ is strict. Thus, we need to show that for any $h : \mu\mathsf{F} \to B$

$$(\mathcal{F} \circ \mathcal{G})(h) = (\mathcal{G} \circ \mathcal{H})(h)$$

The left and right hand side of the equation simplify as follows:

$$\begin{aligned}
(\mathcal{F} \circ \mathcal{G})(h) &= \mathcal{F}(h \circ \llbracket\psi\rrbracket) \\
&= \varphi \circ \mathsf{F}\llbracket\langle h \circ \llbracket\psi\rrbracket, \psi\rangle\rrbracket \circ \psi \\
(\mathcal{G} \circ \mathcal{H})(h) &= \mathcal{G}(\varphi \circ \mathsf{F}\llbracket\langle h, \mathsf{out}\rangle\rrbracket \circ \mathsf{out}) \\
&= \varphi \circ \mathsf{F}\llbracket\langle h, \mathsf{out}\rangle\rrbracket \circ \mathsf{out} \circ \llbracket\psi\rrbracket \\
&= \varphi \circ \mathsf{F}(\llbracket\langle h, \mathsf{out}\rangle\rrbracket \circ \llbracket\psi\rrbracket) \circ \psi
\end{aligned}$$

Here, the last equality holds because of *ana-cancellation*.

Hence, it suffices to show that $\llbracket\langle h, \mathsf{out}\rangle\rrbracket \circ \llbracket\psi\rrbracket = \llbracket\langle h \circ \llbracket\psi\rrbracket, \psi\rangle\rrbracket$. For this we can use *ana-fusion* law because $\llbracket\psi\rrbracket$ is an $\mathsf{F}^\times_B$-coalgebra homomorphism:

$$\begin{tikzcd}
B \times \mathsf{F}A \arrow[d, "id \times \mathsf{F}\llbracket\psi\rrbracket"'] & A \arrow[l, "\langle h \circ \llbracket\psi\rrbracket, \psi\rangle"'] \arrow[d, "\llbracket\psi\rrbracket"] \\
B \times \mathsf{F}\mu\mathsf{F} & \mu\mathsf{F} \arrow[l, "\langle h, \mathsf{out}\rangle"']
\end{tikzcd}$$

The equality on the first component of the product is obvious and the equality on the second component holds by *ana-cancellation*. $\square$

Dynamorphisms come equipped with the following properties:

– **Cancellation:** For any coalgebra $\psi : A \to \mathsf{F}A$ and algebra $\varphi : \mathsf{F}\tilde{\mathsf{F}}B \to B$

$$\mathsf{dyna}(\varphi,\psi)_{\mathsf{F}} = \varphi \circ \mathsf{F}[\!(\langle \mathsf{dyna}(\varphi,\psi)_{\mathsf{F}}, \psi\rangle)\!]_{\mathsf{F}^{\times}} \circ \psi \qquad \text{(dyna-cancellation)}$$

– **Histo-Fusion:** For any algebras $\varphi : \mathsf{F}\tilde{\mathsf{F}}A \to A$, $\psi : \mathsf{F}\tilde{\mathsf{F}}B \to B$ and an arrow $f : A \to B$

$$f \text{ is strict} \wedge f \circ \varphi = \psi \circ \tilde{\mathsf{F}}f \;\Rightarrow\; f \circ \mathsf{dyna}(\varphi,\xi)_{\mathsf{F}} = \mathsf{dyna}(\psi,\xi)_{\mathsf{F}} \qquad \text{(dyna-histo-fusion)}$$

– **Ana-Fusion:** For any coalgebras $\psi : A \to \mathsf{F}A$, $\xi : B \to \mathsf{F}B$, algebra $\varphi : \mathsf{F}\tilde{\mathsf{F}}C \to C$ and morphism $f : A \to B$

$$\xi \circ f = \mathsf{F}f \circ \psi \;\Rightarrow\; \mathsf{dyna}(\varphi,\xi)_{\mathsf{F}} \circ f = \mathsf{dyna}(\varphi,\psi)_{\mathsf{F}} \qquad \text{(dyna-ana-fusion)}$$

– **Histo-as-Dyna:** For any algebra $\varphi : \mathsf{F}\tilde{\mathsf{F}}B \to B$

$$\mathsf{histo}(\varphi)_{\mathsf{F}} = \mathsf{dyna}(\varphi, \mathsf{out})_{\mathsf{F}} \qquad \text{(histo-as-dyna)}$$

– **Ana-as-Dyna:** For any coalgebra $\psi : A \to \mathsf{F}A$

$$[\!(\psi)\!]_{\mathsf{F}} = \mathsf{dyna}(\mathsf{in} \circ \mathsf{F}\varepsilon, \psi)_{\mathsf{F}} \qquad \text{(ana-as-dyna)}$$

Although the recursive form of dynamorphism is more comfortable to reason about it adds an exponential complexity penalty to the function, therefore we use *histo-as-cata* to define a dynamorphism in terms of a corresponding hylomorphism.

Proposition 3. *Let* $\psi : \mathsf{F}A \to A$ *be an* F*-coalgebra and* $\varphi : \mathsf{F}\tilde{\mathsf{F}}B \to B$ *be an* $\mathsf{F}\tilde{\mathsf{F}}$*-algebra, then*

$$\mathsf{dyna}(\varphi,\psi) = \varepsilon_B \circ \mathsf{hylo}(\mathsf{in} \circ \langle \varphi, id\rangle, \psi) \qquad \text{(dyna-as-hylo)}$$

Proof. Follows trivially from *histo-as-cata*, *dyna-definition* and *hylo-definition*. □

Thanks to *dyna-as-hylo* we can now define dynamorphisms which will recursively reuse the annotated structure instead of rebuilding it every time some annotation is needed ("bottom-up" approach versus "top-down"), thus getting an efficient algorithm for their computation. Now we could define effectively the Fibonacci function and others as dynamorphisms (and indeed we do so in the next section), but first we bring in one more property that connects together a hylomorphism function definition with a dynamorphism function definition through a natural transformation.

Theorem 1. *Let* F *and* G *be endofunctors,* $\psi : A \to \mathsf{G}A$ *and* $\xi : A \to \mathsf{F}A$ *coalgebras,* $\varphi : \mathsf{G}B \to B$ *an algebra, and* $\sigma : \mathsf{F}\tilde{\mathsf{F}} \dot{\to} \mathsf{G}$ *a natural transformation. Then*

$$\begin{aligned} &\psi = \sigma_A \circ \theta \circ [\!(\langle id, \xi\rangle)\!]_{\mathsf{F}^{\times}} \\ &\qquad\Longrightarrow\quad \mathsf{hylo}(\varphi,\psi)_{\mathsf{G}} = \mathsf{dyna}(\varphi \circ \sigma_B, \xi)_{\mathsf{F}} \end{aligned} \qquad \text{(rec-compression)}$$

Proof. First note that by *θ-cancellation* the premiss of the implication is equivalent to

$$\psi = \sigma_A \circ \mathsf{F}[\![\langle id, \xi \rangle]\!]_{\mathsf{F}^{\times}} \circ \xi$$

Now, consider the following diagram:

$$\begin{tikzcd}
\mathsf{F}A \arrow[dr, "\mathsf{F}[\![\langle id,\xi\rangle]\!]"] \arrow[dd, "\mathsf{F}[\![\langle f,\xi\rangle]\!]"'] & & & A \arrow[lll, "\xi"'] \arrow[dl, "\psi"] \arrow[dd, "f"] \\
& \mathsf{F}\tilde{\mathsf{F}}A \arrow[r, "\sigma_A"] \arrow[dl, "\mathsf{F}\tilde{\mathsf{F}}f"] & \mathsf{G}A \arrow[d, "\mathsf{G}f"] & \\
\mathsf{F}\tilde{\mathsf{F}}B \arrow[rr, "\sigma_B"] & & \mathsf{G}B \arrow[r, "\varphi"] & B
\end{tikzcd}$$

Here, the square on right corresponds to the hylomorphism and the outer square to the dynamorphism. The upper square corresponds to the left hand side of the implication and the lower square is the naturality square of σ, thus both commute. As the triangle on the left also commutes (due to *map-build-fusion*), it is easy to see that the whole square commutes whenever the square on right does (and vice versa), hence the corresponding sets of functions coincide and their minimal elements, the dynamorphism and the hylomorphism respectively, are equal. □

The *rec-compression* law expresses a property stating that when the coalgebras in hylomorphism and dynamorphism are connected by a natural transformation then we can reuse the algebra in both function definitions.

Intuitively this is the case since dynamic programming technique changes only the recursion structure of the algorithm and there is no need to update the actual calculating part of the algorithm. The required relation between the coalgebras expresses an intuitive fact that the order of recursion in dynamic algorithm must proceed in such a way that the needed subtask solutions are always (if indirectly) available. The natural transformation plays the role of projection that restores one level of the original recursive structure by projecting the annotated values from the depths of the new annotated recursive structure.

4 Examples

In this section we review four classical dynamic algorithms: Fibonacci numbers, binary partitioning of numbers, edit distance and longest common subsequence. In each case we first define it as a hylomorphism and then as a dynamorphism reusing the algebra as shown in *rec-compression* law.

4.1 Fibonacci Numbers

The first algorithm we present is that of Fibonacci numbers, which is a very well known algorithm commonly used for illustrating the dynamic programming technique. Though it is a bit too simple (being the only example we consider that can be expressed as a histomorphism) it is nevertheless useful to illustrate the mechanics of the transformation from a hylomorphism to a dynamorphism definition.

Since we already defined the algorithm in the previous section we start with the hylomorphism definition $\mathit{fibo} = \mathsf{hylo}(\varphi, \psi)_{\mathsf{G}}$, where $\mathsf{G}(X) = \mathbf{1} + \mathbf{1} + X^2$ and $\psi : \mathbb{N} \to \mathsf{G}\mathbb{N}$, $\varphi : \mathsf{G}\mathbb{N} \to \mathbb{N}$ are defined below:

$$\begin{array}{ll} \psi\ (0) & = \mathrm{inj}_1 \perp \\ \psi\ (1) & = \mathrm{inj}_2 \perp \\ \psi\ (n) & = \mathrm{inj}_3\,(n-1,\, n-2) \\ & \\ \varphi\ (\mathrm{inj}_1 \perp) & = 0 \\ \varphi\ (\mathrm{inj}_2 \perp) & = 1 \\ \varphi\ (\mathrm{inj}_3\ (n_1, n_2)) & = n_1 + n_2 \end{array}$$

As one can see the intermediate structure hylomorphism is using is that of binary trees which size is exponential in relation to the size of input. It is also clearly visible that the coalgebra ψ defines the exact shape of the intermediate structure producing dependencies for every argument. So $\psi(0)$ and $\psi(1)$ does not have any dependencies (since their value is constant) and any other natural number n dependencies are $n-1$ and $n-2$.

Next we define Fibonacci as a dynamorphism $\mathit{fibo} = \mathsf{dyna}(\varphi \circ \sigma, \xi)_{\mathsf{F}}$ making use of *rec-compression* law. We use the fact that natural numbers have a natural order and it is obvious that both dependencies ($n-1$ and $n-2$) comes one after another in this order. Therefore we can use the base functor for natural numbers $\mathsf{F}(X) = \mathbf{1} + X$ and define the coalgebra $\xi : \mathbb{N} \to \mathsf{F}\mathbb{N}$ as follows:

$$\begin{array}{l} \xi\ (0) = \mathrm{inl} \perp \\ \xi\ (n) = \mathrm{inr}\,(n-1) \end{array}$$

We derive the definition of $\sigma : \mathsf{F}\tilde{\mathsf{F}} \dot{\to} \mathsf{G}$, by making sure, that the premiss of *rec-compression* law is satisfied. For the case $n = 0$ note that

$$(\theta \circ [\![\langle id, \xi\rangle]\!])\,(0) = \mathrm{inl} \perp$$

Hence, for the case $n = 0$, we have to define $\sigma\ (\mathrm{inl} \perp) = \mathrm{inj}_1 \perp$. If $n > 0$, we have that

$$(\theta \circ [\![\langle id, \xi\rangle]\!])\,(n) = (\mathrm{inr} \circ [\![\langle id, \xi\rangle]\!])\,(n-1)$$

Now, in the case of $n = 1$, we get

$$\begin{aligned} (\theta \circ [\![\langle id, \xi\rangle]\!])\,(1) &= \mathrm{inr}\, x, \text{ where } x = [\![\langle id, \xi\rangle]\!])\,(0) \\ &= \mathrm{inr}\, x, \text{ where } \langle \varepsilon, \theta\rangle(x) = (0,\ \mathrm{inl} \perp) \end{aligned}$$

In the case of $n > 1$, we get

$$\begin{aligned} (\theta \circ [\![\langle id, \xi\rangle]\!])\,(n) &= \mathrm{inr}\, x, \text{ where } x = [\![\langle id, \xi\rangle]\!])\,(n-1) \\ &= \mathrm{inr}\, x, \text{ where } \langle \varepsilon, \theta\rangle(x) = (n-1,\ \mathrm{inr}\, y) \\ &\qquad\qquad \text{and} \quad y = [\![\langle id, \xi\rangle]\!])\,(n-2) \end{aligned}$$

Hence, we can define $\sigma : \mathsf{F}\tilde{\mathsf{F}} \dot{\to} \mathsf{G}$ as follows:

$$\sigma\ (\text{inl}\ \bot) = \quad \text{inj}_1\ \bot$$
$$\sigma\ (\text{inr}\ x) = \begin{cases} \text{inj}_2\ \bot & \text{if } \theta(x) = \text{inl}\ \bot \\ \text{inj}_3(\varepsilon(x), \varepsilon(y)) & \text{if } \theta(x) = \text{inr}\ y \end{cases}$$

Note, that the composite algebra $\varphi \circ \sigma : \mathsf{F}\tilde{\mathsf{F}}\mathbb{N} \to \mathbb{N}$ is equivalent to that of used in the previous section to define Fibonacci as a histomorphism. Moreover, as the coalgebra $\xi : \mathbb{N} \to \mathsf{F}\mathbb{N}$ is the final coalgebra of functor $\mathsf{F}(X) = \mathbf{1} + X$, the dynamorphism we derived is in fact the very same histomorphism.

4.2 Binary Partitions

Binary partitioning of a number is representing this number as a sum of powers of 2. The number of binary partitions for a number n is the number of unique ways to partitions this number (ignoring the order) into powers of 2.

The function can be defined by the following equations:

$$\text{bp}(0) = 1$$
$$\text{bp}(n) = \begin{cases} \text{bp}(n-1), \text{ if } n \text{ is odd} \\ \text{bp}(n-1) + \text{bp}(n/2), \text{ if } n \text{ is even} \end{cases}$$

Defining the function as a hylomorphism is quite straightforward. The intermediate structure in this case is a tree with maximal branching factor of two, i.e., $\text{bp} = \mathsf{hylo}(\varphi, \psi)_{\mathsf{G}}$, where $\mathsf{G}(X) = \mathbf{1} + X + X^2$ and $\psi : \mathbb{N} \to \mathsf{G}\mathbb{N}$, $\varphi : \mathsf{G}\mathbb{N} \to \mathbb{N}$ are defined below:

$$\psi\ (0) \quad = \quad \text{inj}_1\ \bot$$
$$\psi\ (n) \quad = \begin{cases} \text{inj}_2(n-1) & \text{if } n \text{ is odd} \\ \text{inj}_3(n-1,\ n/2) & \text{if } n \text{ is even} \end{cases}$$

$$\begin{aligned} \varphi\ (\text{inj}_1\ \bot) &= 1 \\ \varphi\ (\text{inj}_2\ n) &= n \\ \varphi\ (\text{inj}_3\ (n_1,\ n_2)) &= n_1 + n_2 \end{aligned}$$

To transform the function to a dynamorphism we notice that again $\psi(n)$ functionally depends on the previous numbers only. However, as the dependencies have dynamic depth, we need to know the size of the intermediate structure. Hence we choose the intermediate structure to be a list of natural numbers, i.e., the base functor is $\mathsf{F}(X) = \mathbf{1} + \mathbb{N} \times X$ and define coalgebra $\xi : \mathbb{N} \to \mathsf{F}\mathbb{N}$ as follows:

$$\begin{aligned} \xi\ (0) &= \text{inl}\ \bot \\ \xi\ (n) &= \text{inr}\ (n,\ n-1) \end{aligned}$$

Similarly to Fibonacci function, we derive the definition of $\sigma : \mathsf{F}\tilde{\mathsf{F}} \dot{\to} \mathsf{G}$, by making sure, that the premiss of *rec-compression* law is satisfied. For the case

$n = 0$ we obviously have to define σ (inl $\perp$) $= \mathrm{inj}_1 \perp$. If $n > 0$, we notice that

$$(\theta \circ [\![\langle id, \xi \rangle]\!])\,(n) = \mathrm{inr}(n,\ [\![\langle id, \xi \rangle]\!]\,(n-1))$$

Therefore, if n is odd, we can use ε*-cancellation* $\varepsilon([\![\langle id, \xi \rangle]\!])\,(n-1) = n-1$ to conclude that σ $(\mathrm{inr}(n,\, x)) = \mathrm{inj}_2(\varepsilon(x))$.

If n is even, we need not only the annotation in the root node but also in the node which corresponds to $n/2$. For this, we define a partial function $\pi : \tilde{\mathsf{F}}A \to \tilde{\mathsf{F}}A$ by $\pi(x) = y$, if $\theta(x) = \mathrm{inr}(m, y)$, and denote by π^k its k-fold composition. Using induction over natural numbers, we can show that for $k \leqslant n$

$$(\varepsilon \circ \pi^k \circ [\![\langle id, \xi \rangle]\!])\,(n) = n - k$$

Indeed, the base case $k = 0$ is trivial. For the case $k > 0$ we have

$$\begin{aligned}(\varepsilon \circ \pi^k \circ [\![\langle id, \xi \rangle]\!])\,(n) &= (\varepsilon \circ \pi^{k-1} \circ \pi \circ [\![\langle id, \xi \rangle]\!]),(n)\\ &= (\varepsilon \circ \pi^{k-1}[\![\langle id, \xi \rangle]\!])\,(n-1)\\ &= n - 1 - (k-1) = n - k\end{aligned}$$

Now, since the annotated tree we can access is generated by $[\![\langle id, \xi \rangle]\!]\,(n-1)$, we need to take $k = n - n/2 - 1 = n/2 - 1$. Therefore, we can define the natural transformation $\sigma : \mathsf{F}\tilde{\mathsf{F}} \dot{\to} \mathsf{G}$ as follows:

$$\begin{aligned}\sigma\ (\mathrm{inl}\ \perp) \quad &= \quad \mathrm{inj}_1 \perp\\ \sigma\ (\mathrm{inr}\,(n,\, x)) &= \begin{cases}\mathrm{inj}_2(\varepsilon(x)) & \text{if } n \text{ is odd}\\ \mathrm{inj}_3(\varepsilon(x), \varepsilon(\pi^{n/2-1}(x))) & \text{if } n \text{ is even}\end{cases}\end{aligned}$$

Note that even though the definition $\mathrm{bp} = \mathsf{dyna}(\varphi \circ \sigma, \xi)_{\mathsf{F}}$ uses structural recursion it *cannot* be defined by a single histomorphism. This is due to the need to know the size of the intermediate structure built so far (one can define it though as a composition of a histomorphism and a projection).

4.3 Edit Distance

Edit distance is a classical dynamic programming algorithm that measures the measure of "distance" or "difference" between two strings (i.e. lists of characters List_C). The algorithm can be defined as follows:

$$\begin{aligned}\mathit{editDist}([\,], \mathit{bs}) &= \#\mathit{bs}\\ \mathit{editDist}(\mathit{as}, [\,]) &= \#\mathit{as}\\ \mathit{editDist}(a{:}\mathit{as}, b{:}\mathit{bs}) &= \min\ (\\ &\qquad \mathit{editDist}(\mathit{as}, b{:}\mathit{bs}) + 1,\\ &\qquad \mathit{editDist}(a{:}\mathit{as}, \mathit{bs}) + 1,\\ &\qquad \mathit{editDist}(\mathit{as}, \mathit{bs}) + (\text{if } a = b \text{ then } 0 \text{ else } 1))\end{aligned}$$

Here $\#as$ denotes the length of the list as.

The definition above translates fairly straightforwardly to a hylomorphism $\mathit{editDist} = \mathsf{hylo}(\varphi, \psi)_{\mathsf{G}}$, where $\mathsf{G}(X) = \mathrm{List}_C + C^2 \times X^3$ and $\psi : \mathrm{List}_C^2 \to \mathsf{G}\mathrm{List}_C^2$, $\varphi : \mathsf{G}\mathbb{N} \to \mathbb{N}$ are defined below:

$$
\begin{array}{ll}
\psi\ ([\,],\ bs) & = \mathrm{inl}\, bs \\
\psi\ (as,\ [\,]) & = \mathrm{inl}\, as \\
\psi\ (a{:}as,\ b{:}bs) & = \mathrm{inr}\, ((a,b),\ (as, b{:}bs),\ (a{:}as, bs),\ (as, bs)) \\
\\
\varphi\ (\mathrm{inl}\, as) & = \#as \\
\varphi\ (\mathrm{inr}\ ((a,b),\ x_1,\ x_2,\ x_3)) & \\
 & = \min\ (x_1 + 1,\ x_2 + 1,\ x_3 + (\text{if } a = b \text{ then } 0 \text{ else } 1))
\end{array}
$$

However translating this definition to a dynamorphism is not as simple. Typically a matrix is used to accumulate and look up values in the dynamic version of edit distance algorithm. Matrix however is not an inductive structure and thus cannot be used as the intermediate structure in the dynamorphism. So instead of using a matrix we use a *walk-through* of a matrix—a list of values from the matrix ordered predictably, e.g. row by row or column by column or wavefront way. This list is indeed inductive and also corresponds in some way to the recursion used in the edit distance algorithm to walk through and fill in the matrix. In our case we choose to order row by row (or column by column, depending how you imagine the matrix).

However, taking such a list for the intermediate structure brings in another problem—now to project a value from the previous row or column (which we need in this case) and to build the list recursively we need to know at least one input string in our coalgebra and its length in the natural transformation. As these are constant throughout the computation, we give them as an additional parameter to ξ and σ denoted by subscript. So,

$$\mathit{editDist}\,(s_1, s_2) = \mathsf{dyna}(\varphi \circ \sigma_{\#s_1},\ \xi_{s_1})_{\mathsf{F}}\ (s_1, s_2),$$

where $\mathsf{F}(X) = (\mathrm{List}_C)^2 \times (\mathbf{1} + X)$ and $\xi_{s_1} : \mathrm{List}_C^2 \to \mathsf{F}\mathrm{List}_C^2$, $\sigma_n : \mathsf{F}\tilde{\mathsf{F}} \stackrel{\cdot}{\to} \mathsf{G}$ are defined below:

$$
\begin{array}{ll}
\xi_{cs}\ ([\,], [\,]) & = (([\,], [\,]),\ \mathrm{inl}\, \bot) \\
\xi_{cs}\ ([\,], b{:}bs) & = (([\,], b{:}bs),\ \mathrm{inr}\ (cs, bs)) \\
\xi_{cs}\ (a{:}as,\ bs) & = ((a{:}as, bs),\ \mathrm{inr}\ (as, bs)) \\
\\
\sigma_n\ ((as, bs),\ \mathrm{inl}\, \bot) & = \mathrm{inl}\ [\,] \\
\sigma_n\ (([\,], bs),\ \mathrm{inr}\ x) & = \mathrm{inl}\ \ bs \\
\sigma_n\ ((as, [\,]),\ \mathrm{inr}\ x) & = \mathrm{inl}\ \ as \\
\sigma_n\ ((a{:}as, b{:}bs),\ \mathrm{inr}\ x) & = \mathrm{inr}\ ((a,b),\ \varepsilon\,(x),\ \varepsilon\,(\pi^n\, x),\ \varepsilon\,(\pi^{n+1}\, x))
\end{array}
$$

While the definition of σ_n could be derived similarly to the previous examples, we do not present the derivation here, as it gets quite complicated.

Note that since the base functor F of edit distance differs from that of binary partitions the partial function $\pi : \tilde{\mathsf{F}}A \to \tilde{\mathsf{F}}A$ is defined here by $\pi(x) = y$, if $\theta(x) = (s, \mathrm{inr}(m, y))$, which is also the definition used in the next example.

4.4 Longest Common Subsequence

Longest common subsequence is another well-known dynamic programming algorithm. It finds the longest character subsequence that is common to both input strings. It can be defined as follows:

$$
\begin{aligned}
lcs([\,], bs) &= [\,] \\
lcs(as, [\,]) &= [\,] \\
lcs(a{:}as, b{:}bs) &= \text{if } a = b \text{ then } a{:}lcs(as, bs) \\
&\qquad\quad \text{else (if } \#lcs(a{:}as, bs) > \#lcs(as, b{:}bs) \\
&\qquad\qquad\qquad \text{then } lcs(a{:}as, bs) \\
&\qquad\qquad\qquad \text{else } lcs(as, b{:}bs))
\end{aligned}
$$

The hylomorphism definition comes straight from the above.

$$
\begin{aligned}
&\mathsf{G}(X) && = \mathbf{1} + C^2 \times X^3 \\
&lcs && = \mathsf{hylo}(\varphi, \psi)_{\mathsf{G}} \\
\\
&\psi\ ([\,],\ bs) && = \text{inl}\,\bot \\
&\psi\ (as,\ [\,]) && = \text{inl}\,\bot \\
&\psi\ (a{:}as,\ b{:}bs) && = \text{inr}\,((a, b),\ (as, b{:}bs),\ (a{:}as, bs),\ (as, bs)) \\
\\
&\varphi\ (\text{inl}\,\bot) && = [\,] \\
&\varphi\ (\text{inr}\ ((a, b),\ x_1,\ x_2,\ x_3)) && \\
& && = \text{if } a = b \text{ then } a{:}x_3 \text{ else (if } \#x_1 > \#x_2 \text{ then } x_1 \text{ else } x_2)
\end{aligned}
$$

When defining the dynamorphism version we have the same problems as with edit distance and we solve them the same way.

$$
\begin{aligned}
&\mathsf{F}(X) && = (\text{List}_C)^2 \times (\mathbf{1} + X) \\
&lcs(s_1, s_2) && = \mathsf{dyna}(\varphi \circ \sigma_{\#s_1}, \xi_{s_1})_{\mathsf{F}}\ (s_1, s_2) \\
\\
&\xi_{cs}\ ([\,], [\,]) && = (([\,], [\,]),\ \text{inl}\,\bot) \\
&\xi_{cs}\ ([\,],\ b{:}bs) && = (([\,], b{:}bs),\ \text{inr}\ (cs, bs)) \\
&\xi_{cs}\ (a{:}as,\ b{:}bs) && = ((a{:}as, b{:}bs),\ \text{inr}\ (as, b{:}bs)) \\
\\
&\sigma_n\ ((as, bs),\ \text{inl}\,\bot) && = \text{inl}\,\bot \\
&\sigma_n\ (([\,], bs),\ \text{inr}\ x) && = \text{inl}\,\bot \\
&\sigma_n\ ((as, [\,]),\ \text{inr}\ x) && = \text{inl}\,\bot \\
&\sigma_n\ ((a{:}as, b{:}bs),\ \text{inr}\ x) && = \text{inr}\ ((a, b),\ \varepsilon\,(x),\ \varepsilon\,(\pi^n\, x),\ \varepsilon\,(\pi^{n+1}\, x))
\end{aligned}
$$

5 Conclusions and Future Work

We have shown that the dynamic programming recursion pattern can be captured by a generic recursion combinator, dynamorphism, which avoids the recomputing of identical substructures. We have identified a simple condition when a function defined by a hylomorphism can be redefined as a dynamorphism. While the transformation is not automatic, it requires of guessing a coalgebra, it allows to reuse some parts of original definition, namely the algebra. Moreover, after the coalgebra is given, the another required component of the new definition, a natural transformation, can often be derived without further guessing.

In all our examples, the intermediate structure used by dynamorphisms was linear, i.e. some form of lists. Of course, this is to be expected, as one of the requirements of dynamic programming is the existence of partial order among the value dependencies, which we have sorted topologically to get a linear order. While our formulation allows arbitrary tree-shaped intermediate structures, it's not clear whether this generality is really useful. Additional restrictions on the shape of the intermediate structure might provide necessary information for deriving instead of guessing the corresponding coalgebra. This is an important area of further work.

It is known that histomorphism is an instance of comonadic recursion [11] and that the latter can be generalized in a recursive coalgebra setting [4]. What is the exact relationship between dynamorphisms and comonadic recursive coalgebras is another interesting topic to study.

Acknowledgments. We are grateful to our anonymous referees for their constructive criticism and suggestions. We are also grateful to Jeremy Gibbons for first pointing out that dynamic programming might provide relevant examples of histomorphisms, and to Tarmo Uustalu for technical discussions. This work was partially supported by Estonian Science Foundation grants No. 5567 and No. 6713.

References

1. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton, New Jersey (1957)
2. Bird, R.S.: An introduction to the theory of lists. In Broy, M., ed.: Logic of Programming and Calculi of Discrete Design. Volume 36 of NATO ASI Series F. Springer-Verlag, Berlin (1987) 3–42
3. Bird, R., de Moor, O.: Algebra of Programming. Volume 100 of Prentice Hall Int. Series in Computer Science. Prentice Hall, London (1997)
4. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. In Adámek, J., Milius, S., eds.: Proc. of 7th Int. Wksh. on Coalgebraic Methods in Computer Science, CMCS 2004 (Barcelona, Spain, 29–31 March 2004). Volume 106C of Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam (2004) 43–61
5. Fokkinga, M.: Law and Order in Algorithmics. PhD thesis, Dept. of Informatics, Univ. of Twente (1992)
6. Gibbons, J.: Calculating functional programs. In Backhouse, R.C., Crole, R.L., Gibbons, J., eds.: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. Volume 2297 of Lecture Notes in Computer Science., Springer (2000) 149–202

7. Hagino, T.: A Categorical Programming Language. PhD thesis CST-47-87, Laboratory for Foundations of Computer Science, Dept. of Computer Science, Univ. of Edinburgh (1987)
8. Malcolm, G.: Data structures and program transformation. Science of Computer Programming **14**(2–3) (1990) 255–279
9. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In Hughes, J., ed.: Proc. of 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91, Cambridge, MA, USA, 26–30 Aug 1991. Volume 523 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1991) 124–144
10. Uustalu, T., Vene, V.: Primitive (co)recursion and course-of-value (co)iteration, categorically. Informatica **10**(1) (1999) 5–26
11. Uustalu, T., Vene, V., Pardo, A.: Recursion schemes from comonads. Nordic Journal of Computing **8**(3) (2001) 366–390
12. Vene, V.: Categorical Programming with Inductive and Coinductive Types. PhD thesis (Diss. Math. Univ. Tartuensis 23), Dept. of Computer Science, Univ. of Tartu (2000)