Categorical Programming with Abstract Data Types

Martin Erwig

FernUniversität Hagen, Praktische Informatik IV
58084 Hagen, Germany
erwig@fernuni-hagen.de

Abstract. We show how to define fold operators for abstract data types. The main idea is to represent an ADT by a bialgebra, that is, an algebra/coalgebra pair with a common carrier. A program operating on an ADT is given by a mapping to another ADT. Such a mapping, called *metamorphism*, is basically a composition of the algebra of the second with the coalgebra of the first ADT. We investigate some properties of metamorphisms, and we show that the metamorphic programming style offers far-reaching opportunities for program optimization that cover and even extend those known for algebraic data types.

1 Introduction

Expressing recursion over data types in terms of catamorphisms, or fold operations, has been successfully employed by Bird and Meertens to calculate programs from specifications [3, 21]. They formulated laws expressing algebraic identities of programs and used them to derive algorithms in a sequence of simple transformation steps. Their work was primarily focused on lists, but it has been extended to regular algebraic data types [20, 22, 25]: a data type is given by a morphism which is a fixed point of a functor defining the signature of the data type. Since fixed points are initial objects, homomorphisms to other data types are uniquely defined, and this makes it possible to specify a program on a data type by simply selecting an appropriate target data type. Along with these generalizations a lot of work on program optimization has emerged that essentially relies on programs being expressed as catamorphisms, for example, [13, 25, 26, 16, 17]. The strong interest in program fusion is certainly due to the fact that catamorphisms encapsulate a class of recursion over data types that enjoys some nice mathematical properties.

Besides the original idea of having a framework for calculating programs from their specifications, avoiding general recursion is also important from a programming methodology point of view. For instance, Meijer et al. [22] stress in their often cited paper the aspect that using folds in functional languages is truly in the spirit of the structured programming methodology. Similar beliefs, that is, avoiding general recursion and using a fixed set of higher order functions, had been already emphasized before by Backus [1]. The programming languages CPL

[14] and Charity [11] are designed thoroughly on the basis of this programming methodology.

It is striking that the categorical framework has been applied only sporadically to data types that are not just given by free term structures, such as abstract data types. Although the catamorphism approach principally works also for sub-algebras that satisfy certain laws, one cannot map into algebras with less structure [9, 10]. This innocent looking restriction means a rather severe limitation of expressiveness: for instance, such a simple task as counting the elements of a set *cannot* be expressed by a catamorphism.

Therefore we propose to base mappings between ADTs not just on constructors, but on explicitly defined destructors. Formally, this means to represent an ADT by a bialgebra, that is, a pair (algebra, coalgebra) with a common carrier, and to define a mapping between two ADTs D and D' by composing the algebra of D' with the coalgebra of D. This offers much freedom in specifying ADTs and mappings between them. It also provides a new programming style encouraging the compositional use of ADTs. The proposed approach essentially uses existing concepts, such as algebra and coalgebra, on a higher level of abstraction, and this is the reason that all the optimization rules developed for algebraic data types are still valid in this extended framework. But in addition to this, the "programming by ADT composition" style offers some new optimization opportunities: for example, since intermediate ADTs are intrinsically used in a single-threaded way, a compiler can automatically insert efficient imperative update-in-place implementations for them.

The rest of the paper is structured as follows: after reviewing the categorical definition of algebraic data types in Sect. 2, we show how to represent abstract data types as bialgebras in Sect. 3. In Sect. 4 we introduce metamorphisms as mappings between ADTs, we provide various example programs, and we show some basic properties of ADTs and metamorphisms. In Sect. 5 we investigate several aspects of program transformation. Related work is described in Sect. 6, and conclusions follow in Sect. 7.

2 Data Types and Homomorphisms

In this section we briefly review the categorical framework for modeling data types. More detailed introductions are given, for example, in [4, 9]. Specific information about coalgebras can be found in [15, 24], and hylomorphisms are explained and used in [22, 26, 16]. We assume some basic knowledge about category theory (an understanding of *category*, *functor*, and *natural transformation* should be sufficient), for an introduction see, for example, [2] or [4].

In Sect. 2.1 we briefly recall the notion of algebraic data types of functional languages. In Sect. 2.2 we show how to express signatures by functors. This is the basis for the definition of algebras and coalgebras in Sect. 2.3. We demonstrate how algebra homomorphisms can express programs on data types in Sect. 2.4.

2.1 Algebraic Data Types in Functional Languages

In modern functional languages, like ML or Haskell, data structures are represented by terms that are built by typed constructors. All constructors of one type are introduced in one definition, and the defined type is called an *algebraic data type.* For example, a term representation for natural numbers and a polymorphic list data type are given by:

$$\begin{aligned} \mathit{nat} &= \mathit{Zero} \mid \mathit{Succ}\ \mathit{nat} \\ \mathit{list}\ A &= \mathit{Nil} \mid \mathit{Cons}(A, \mathit{list}\ A) \end{aligned}$$

This introduces the four constructors $\mathit{Zero} : \mathbf{1} \to \mathit{nat}$, $\mathit{Succ} : \mathit{nat} \to \mathit{nat}$, $\mathit{Nil} : \mathbf{1} \to \mathit{list}\ A$, and $\mathit{Cons} : A \times \mathit{list}\ A \to \mathit{list}\ A$, where $\mathbf{1}$ denotes the *unit type* that contains just the one element $()$, that is, constants of type A are identified with functions of type $\mathbf{1} \to A$. Thus, a data type can be viewed as an algebra whose operations are given by the data type constructors. Note that we consider non-strict constructors.

2.2 Polynomial Functors

In this paper the default category $\mathcal{C}$ is **CPO**, whose objects are complete partially ordered sets with a least element $\perp$ and whose morphisms are continuous functions. Working in **CPO** guarantees the existence of least fixed points for certain recursive equations needed in Sect. 2.4 and in Sect. 4. The terminal object in $\mathcal{C}$ is the one-element type $\mathbf{1}$. In the sequel we consider only endofunctors on $\mathcal{C}$ (that is, functors from $\mathcal{C}$ to $\mathcal{C}$) which are built by the four basic functors I (*identity*), $\underline{A}$ (*constants*), $\times$ (*product*), and $+$ (*separated sum*).

The effect of the identity functor I on types and functions is $I\ A = A$ and $I\ f = f$, and the constant functor for type A, denoted by $\underline{A}$, operates on types and functions by $\underline{A}\ B = A$ and $\underline{A}\ f = \mathrm{id}_A$ where id_A denotes the identity morphism on A. For an object x we denote its constant function by $\underline{x}$, that is, $\underline{x}\ y = x$. The *product* of two types A and B and its operation on functions is defined as:

$$\begin{aligned} A \times B &= \{(x, y) \mid x \in A, y \in B\} \\ (f \times g)\ (x, y) &= (f\ x, g\ y) \end{aligned}$$

Related operations are left and right projection and tupling (also called split):

$$\begin{aligned} \pi_1\ (x, y) &= x \\ \pi_2\ (x, y) &= y \\ \langle f, g \rangle\ x &= (f\ x, g\ x) \end{aligned}$$

Finally, the *separated sum* of two types A and B and its operation on functions is defined as:

$$\begin{aligned} A + B &= \{1\} \times A \cup \{2\} \times B \cup \{\bot\} \\ (f+g)\ (1,x) &= (1, f\ x) \\ (f+g)\ (2,y) &= (2, g\ y) \\ (f+g)\ \bot &= \bot \end{aligned}$$

Related operations are left and right injection and case analysis (also called junc):

$$\begin{aligned} \iota_1\ x &= (1,x) \\ \iota_2\ y &= (2,y) \\ [f,g]\ (1,x) &= f\ x \\ [f,g]\ (2,y) &= g\ y \\ [f,g]\ \bot &= \bot \end{aligned}$$

The use of seperated sum is convenient for the treatment of, for example, infinite lists. However, algebras for, say, natural numbers, are usually better modeled with coalesced sum (which identifies bottom elements). These issues are discussed in some detail in [9, 19].

Separated sum and product are bifunctors that map from the product category $\mathcal{C} \times \mathcal{C}$ to $\mathcal{C}$. Fixing one parameter of a bifunctor yields a monofunctor: the (left) *section* of a bifunctor F and an object A is defined as $F_A(B) = F(A, B)$. Thus, for example, $\times_A$ is a monofunctor which takes an object B and maps it to the product $A \times B$.

Now *polynomial functors* are inductively defined as follows: (i) I and $\underline{A}$ are polynomial functors, and (ii) if F and G are polynomial functors, then so are their composition FG, their sum $F+G$, and their product $F\times G$ where: $(F+G)(X) = F(X) + G(X)$ and $(F \times G)(X) = F(X) \times G(X)$ (for both types and functions X). Two examples of polynomial functors are:

$$\begin{aligned} N &= \underline{\mathbf{1}} + I \\ L_A &= \underline{\mathbf{1}} + \underline{A} \times I \end{aligned}$$

Here L_A is actually a left section of a bifunctor.

(A note on operator precedence: function application binds strongest, and $\times$ binds stronger than $+$, which in turn binds stronger than composition "$\circ$".)

2.3 Algebras and Coalgebras

Let endofunctor $F : \mathcal{C} \to \mathcal{C}$ represent a signature. Then an *F-algebra* is a morphism $\alpha : F(A) \to A$. Object A is called the *carrier* of the algebra. We can extract the carrier of an algebra with the forgetful functor U, that is, $U(\alpha) = A$. Dually, an *F-coalgebra* is a morphism $\overline{\alpha} : A \to F(A)$. An *F-homomorphism* from algebra $\alpha : F(A) \to A$ to algebra $\beta : F(B) \to B$ is a morphism $h : A \to B$ in $\mathcal{C}$ that satisfies $h \circ \alpha = \beta \circ F(h)$. As a shorthand for this condition we write: $h : \alpha \to_F \beta$. The category of F-algebras $\mathbf{Alg}(F)$ has as objects F-algebras and as arrows F-homomorphisms. (Composition and identities in $\mathbf{Alg}(F)$ are taken from $\mathcal{C}$.) Dually, $\mathbf{CoAlg}(F)$ is the category of F-coalgebras with F-cohomomorphisms (where a morphism $h : A \to B$ from coalgebra $\overline{\alpha} : A \to F(A)$ to coalgebra $\overline{\beta} : B \to F(B)$ is a *cohomomorphism* if it satisfies $F(h) \circ \overline{\alpha} = \overline{\beta} \circ h$).

If F is a polynomial functor on **CPO**, $\mathbf{Alg}(F)$ has an initial object, which is denoted by in_F. This means that $in_F : F(T) \to T$ is an F-algebra with carrier $T = U(in_F)$ (the "T" reminds of "term algebra"). Dually, $\mathbf{CoAlg}(F)$ has a terminal object, denoted by out_F, and $out_F : T \to F(T)$ is an F-coalgebra with the same carrier T as in_F. Moreover, in_F and out_F are each other's inverses, and they thus define an isomorphism $T \cong F(T)$ in **CPO**.

Now the definitions of Sect. 2.1, written categorically as:

$$\begin{aligned} [\mathit{Zero}, \mathit{Succ}] &: N(\mathit{nat}) \to \mathit{nat} \\ [\mathit{Nil}, \mathit{Cons}] &: L_A(\mathit{list}\ A) \to \mathit{list}\ A \end{aligned}$$

with $N(\mathit{nat}) = \mathbf{1} + \mathit{nat}$ and $L_A(\mathit{list}\ A) = \mathbf{1} + A \times \mathit{list}\ A$, define the data types as initial objects in the category of N-algebras, respectively, L_A-algebras, that is, $\mathit{nat} = [\mathit{Zero}, \mathit{Succ}] := in_N$ and $\mathit{list}\ A = [\mathit{Nil}, \mathit{Cons}] := in_{L_A}$.

With sums we can also define conditionals. First, we define a type for booleans by:

$$\mathit{bool} = \mathit{True} \mid \mathit{False}$$

(which is just syntax for $\mathit{bool} = [\mathit{True}, \mathit{False}] := in_B$ with $B = \underline{\mathbf{1}} + \underline{\mathbf{1}}$.) Now for each predicate $p : A \to \mathit{bool}$ a morphism $p? : A \to A + A$ is defined by [22]:

$$p?(a) = \begin{cases} \bot & \text{if } p(a) = \bot \\ \iota_1\ a & \text{if } p(a) = \mathit{True} \\ \iota_2\ a & \text{if } p(a) = \mathit{False} \end{cases}$$

The conditional is then simply defined by **if** p **then** f **else** $g = [f, g] \circ p?$. (A more detailed categorical exposition of this can be found in [4].)

Those coalgebras that are the inverses of initial algebras defined by algebraic data type definitions just undo the term construction. So we can think, for example, of out_N and out_{L_A} as being defined by:

$$\begin{aligned} out_N &= \lambda n.\mathbf{case}\ n\ \mathbf{of} \\ &\qquad\quad \mathit{Zero} \to (1, ()) \\ &\qquad \mid\ \mathit{Succ}(m) \to (2, m) \\ out_{L_A} &= \lambda l.\mathbf{case}\ l\ \mathbf{of} \\ &\qquad\quad \mathit{Nil} \to (1, ()) \\ &\qquad \mid\ \mathit{Cons}(x, l') \to (2, (x, l')) \end{aligned}$$

2.4 Catamorphisms, Anamorphisms, and Hylomorphisms

Initial and terminal objects are unique up to isomorphism, and they are characterized by having exactly one morphism to, respectively, from, all other objects. This means that for each F-algebra α in the category $\mathbf{Alg}(F)$ there is exactly one F-homomorphism $h : in_F \to_F \alpha$. Since h is uniquely determined by α, it is conveniently denoted by $(\![\alpha]\!)_F$, or just $(\![\alpha]\!)$ when F is clear from the context, and h is called a *catamorphism* [22]. Accordingly, for each F-coalgebra $\overline{\alpha}$ in the category $\mathbf{CoAlg}(F)$ there is exactly one F-cohomomorphism $h : \overline{\alpha} \to_F out_F$, which is denoted by $[\!(\overline{\alpha})\!]_F$ (or just $[\!(\overline{\alpha})\!]$) and which is called an *anamorphism*.

Programs mapping from an initial algebra to another data type can be succinctly expressed as catamorphisms. An F-catamorphism $(\![\alpha]\!)$ can be thought of as a function replacing the constructors of in_F by the functions/constructors of α; catamorphisms offer a canonical way of consuming a data structure. Similarly, mappings to terminal algebras can be expressed by anamorphisms, which provide a canonical way of constructing data structures. It is clear that the identity is the unique morphism from the initial F-algebra to itself (respectively, to the terminal F-coalgebra from itself):

$$(\![in_F]\!)_F \;=\; [\!(out_F)\!]_F \;=\; \mathrm{id} \qquad \text{(CataId, AnaId)}$$

Finally, a *hylomorphism* is essentially the composition of a catamorphism with an anamorphism. Formally, a hylomorphism $[\![\alpha, \overline{\alpha}]\!]_F$ is defined as the least morphism h satisfying:

$$h \;=\; \alpha \circ F(h) \circ \overline{\alpha} \qquad \text{(HyloDef)}$$

Hylomorphisms are related to cata- and anamorphisms in an obvious way:

$$\begin{aligned} [\![\alpha, \overline{\alpha}]\!]_F &= (\![\alpha]\!)_F \circ [\!(\overline{\alpha})\!]_F & \text{(HyloSplit)} \\ [\![\alpha, out_F]\!]_F &= (\![\alpha]\!)_F & \text{(HyloCata)} \\ [\![in_F, \overline{\alpha}]\!]_F &= [\!(\overline{\alpha})\!]_F & \text{(HyloAna)} \end{aligned}$$

Hylomorphisms enjoy a number of useful laws [22, 26], for example,

$$\begin{aligned} [\![\alpha \circ \eta, \overline{\alpha}]\!]_F &= [\![\alpha, \eta \circ \overline{\alpha}]\!]_G \quad \Leftarrow \quad \eta : F \dot{\to} G & \text{(HyloShift)} \\ [\![\alpha, \overline{\alpha}]\!]_F \circ [\![\beta, \overline{\beta}]\!]_F &= [\![\alpha, \overline{\beta}]\!]_F \quad \Leftarrow \quad \overline{\alpha} \circ \beta = \mathrm{id} & \text{(HyloFusion)} \end{aligned}$$

A hylomorphism $[\![\alpha, \overline{\alpha}]\!]_F$ defines a recursive function whose recursion follows that of the functor F.

3 Abstract Data Types as Bialgebras

The main hindrance for expressing certain catamorphisms on ADTs is that homomorphisms are not able to map to less constrained structures.

One solution is to decouple the decomposition of ADT values from their construction to gain more flexibility. This can be achieved by modeling an ADT by a pair $(\alpha, \overline{\alpha})$ where α is an F-algebra, $\overline{\alpha}$ is a G-coalgebra, and $U(\alpha) = U(\overline{\alpha})$. Such an algebra/coalgebra-pair with a common carrier is called an *F, G-bialgebra* [10]. An *F, G-bialgebra homomorphism* from $(\alpha, \overline{\alpha})$ to $(\beta, \overline{\beta})$ is a morphism satisfying $h \circ \alpha = \beta \circ F(h)$ and $G(h) \circ \overline{\alpha} = \overline{\beta} \circ h$. The F, G-algebras and homomorphisms form a category $\mathbf{BiAlg}(F, G)$, which is built upon $\mathcal{C}$.[1] In the sequel we use the terms "ADT" and "bialgebra" as synonyms. Given an ADT $D = (\alpha, \overline{\alpha})$, we call α the *constructor* of D and $\overline{\alpha}$ the *destructor* of D.

Note that, in general, we have to provide the carrier of an ADT explicitly, since it is not determined by a universal property (like for initial algebras or terminal coalgebras). If not stated otherwise, we will always implicitly take the carrier of the initial algebra (or terminal coalgebra) whenever it is used as a constructor (respectively, as a destructor). For example, the carrier of the bialgebras *Nat*, *Range*, and *Prod* is always $U(in_N)$, and the carrier of the bialgebras *List*, *Queue*, and *Set* is $U(in_{L_A})$.

Let us consider some examples. First of all, algebraic data types can be regarded as ADTs by taking the initial algebra as constructor and its inverse as destructor. For example, ADT $List = (in_{L_A}, out_{L_A})$ is an L_A, L_A-bialgebra; *List* can also be used as a stack ADT. Similarly, we can define: $Nat = (in_N, out_N)$. But we can define many more different ADTs for natural numbers. We can consider, for instance, binary destructors, that is, $L_{U(in_N)}$-coalgebras. One example (that will be used later) is the ADT *Range* which decomposes a number by returning the number itself in addition to the predecessor:

$$Range \;=\; (in_N, [I, \langle succ, I \rangle] \circ out_N)$$

Note that using *succ* is indeed correct here, since out_N gives the predecessor (which is preserved by I) and *succ* re-builds the original number value. This also shows that the constructor and the destructor of an ADT need not have the same signature. Another example for this is the $L_{U(in_N)}, N$-bialgebra *Prod* that constructs numbers by multiplication:

$$Prod \;=\; ([1, *], out_N)$$

Our next example is an ADT for queues. The constructors of a queue are the same as for *List*. The destructor is also an L_A-coalgebra, but it is different from out_{L_A}, since elements are taken from the end. There are different ways to define the queue destructor. First of all, we can give a recursive function definition (which is possible, since we are working in **CPO**).

$$\begin{array}{rclll} dequeue & = & \mu[\; f = \lambda l.\mathbf{case}\; l\; \mathbf{of} & & \\ & & \quad Nil & \rightarrow (1, ()) & \\ & & \mid\; Cons(x, Nil) & \rightarrow (2, (x, Nil)) & \\ & & \mid\; Cons(x, l') & \rightarrow (I + I \times (Cons \circ \langle \underline{x}, I \rangle))\; (f\; l') \;] & \end{array}$$

[1] An F, G-bialgebra is just a special case of an *F, G-dialgebra*, that is, $\mathbf{BiAlg}(F, G) = \mathbf{DiAlg}([F, I], [I, G])$ [10, 9, 7]. Working with bialgebras is sufficient for our purposes and makes the separation of constructors and destructors more explicit.

(This definition could be written more conveniently if **let**-expressions were available.) A more categorical style is to use only combinators and catamorphisms. With the aid of a function *snoc* for appending a single element at the end of a list and a function *rev* for reversing a list – both defined by L_A-catamorphisms:[2]

$$\begin{aligned} \mathit{snoc}(x,l) &= (\![\mathit{Cons}(x,\mathit{Nil}),\mathit{Cons}]\!)_{L_A}\ l \\ \mathit{rev} &= (\![\mathit{Nil},\mathit{snoc}]\!)_{L_A} \end{aligned}$$

we can define the queue destructor as follows.

$$\mathit{dequeue} = I + I \times \mathit{rev} \circ \mathit{out}_{L_A} \circ \mathit{rev}$$

This means, a queue is represented by a list where elements are enqueued at the front and dequeued from the rear. In particular, dequeueing from a list l works as follows: reverse l, take first element x and tail l' from $\mathit{rev}\, l$, and finally reverse l' to get the standard queue representation. Now we can define:

$$\mathit{Queue} = (\mathit{in}_{L_A}, \mathit{dequeue}).$$

As our final example we define a set ADT, again based on the "cons"-view given by L_A. This can be done in two principally different ways. One possibility that quickly comes to mind is to define equations E expressing idempotence and commutativity and work with the quotient algebra "in_{L_A}/E".[3] The problem with this approach is that homomorphisms are forced to stay within L_A/E-algebras, and it is not obvious how to define destructors into different algebras. Thus, it is not clear how to define a function for counting the elements of a set.

To define sets as bialgebras, we can use a list carrier and normalize lists in constructors or destructors. The second option means to take in_{L_A} as constructor. The destructor must then be defined so that a value is retrieved from a set at most once. This can be realized by splitting an arbitrary element off (for example, the one that was inserted last) and removing all occurrences of this element in the returned set. We need the following functions:

$$\begin{aligned} \mathit{append}(l,l') &= (\![l',\mathit{Cons}]\!)_{L_A}\ l \\ \mathit{flatten} &= (\![\mathit{Nil},\mathit{append}]\!)_{L_{list\ A}} \\ \mathit{map}\ f &= (\![\mathit{Nil},\mathit{Cons} \circ f \times I]\!)_{L_A} \\ \mathit{filter}(p,l) &= (\mathit{flatten} \circ \mathit{map}\ (\lambda y.\mathbf{if}\ p(y)\ \mathbf{then}\ \mathit{Cons}(y,\mathit{Nil})\ \mathbf{else}\ \mathit{Nil}))\ l \\ \mathit{remove}(x,l) &= \mathit{filter}\ ((\neq x), l) \end{aligned}$$

Now we can define the set destructor and the set ADT by:

$$\begin{aligned} \mathit{deset} &= I + \langle \pi_1, \mathit{remove} \rangle \circ \mathit{out}_{L_A} \\ \mathit{Set} &= (\mathit{in}_{L_A}, \mathit{deset}) \end{aligned}$$

[2] For readability we omit the junc-brackets inside catamorphisms.

[3] Equations can be expressed categorically by *transformers* [10], which are mappings between algebras. Examples can be found in [10, 9, 18].

Note that the definition works only for types A on which equality is defined.

We have actually given one concrete implementation of sets based on lists, and strictly we have still to prove that this implementation is correct. So the presented bialgebra approach to programming with ADTs is definitely not as high-level as equational specifications. However, we believe it is more flexible. In particular, the bialgebra approach encourages to combine different F-algebras and G-coalgebras, which makes it easy to adapt ADTs to changing requirements. For example, instead of splitting off single elements with *deset* we can also use a P-coalgebra *split* (where $P = \underline{\mathbf{1}} + I \times I$) to partition a set into two equally sized sets. This can be useful, for example, for divide-and-conquer algorithms.

4 Programming by Metamorphisms

From now on let $D = (\alpha, \overline{\alpha})$ be an F, G-bialgebra, let $D' = (\beta, \overline{\beta})$ be an H, J-bialgebra, and let $C = (\varphi, \overline{\varphi})$ be a K, M-bialgebra.

Metamorphisms and Data Type Filters. If $f : G \dot{\to} H$ is a natural transformation, the *f-metamorphism* from D to D' is defined as the least solution of the equation

$$h \;=\; \beta \circ f \circ G(h) \circ \overline{\alpha} \qquad \text{(MetaDef)}$$

and is denoted by $D \overset{f}{\rightsquigarrow} D'$ (we write $D \rightsquigarrow D'$ if $f = \mathrm{id}$). We call D/D' the *source*/*target* and f the *map* of the metamorphism. This definition says that a metamorphism from D to D' is essentially a hylomorphism:

$$D \overset{f}{\rightsquigarrow} D' \;=\; [\![\beta \circ f, \overline{\alpha}]\!]_G \qquad \text{(MetaHylo)}$$

As an important special case, metamorphisms from algebraic data types reduce to catamorphisms, that is,

$$D \rightsquigarrow D' \;=\; (\![\beta]\!)_G \quad \Leftarrow \quad D = (in_G, out_G) \qquad \text{(MetaAlg)}$$

This can be seen as follows. First we know $H = G$, since $f = \mathrm{id}$. Then

$$\begin{aligned} D \rightsquigarrow D' &= [\![\beta, out_G]\!]_G && \{ \text{ MetaHylo } \} \\ &= (\![\beta]\!)_G \circ [\!(out_G)\!]_G && \{ \text{ HyloSplit } \} \\ &= (\![\beta]\!)_G \circ \mathrm{id} && \{ \text{ AnaId } \} \\ &= (\![\beta]\!)_G \end{aligned}$$

As an abbreviation for the composition of two metamorphisms we introduce the notion of an *ADT-filter*. The *C-filter* from D to D' is defined as:

$$D \overset{f}{\rightsquigarrow} C \overset{g}{\rightsquigarrow} D' \;=\; C \overset{g}{\rightsquigarrow} D' \circ D \overset{f}{\rightsquigarrow} C \qquad \text{(FilterDef)}$$

Here D and D' are called the *source* and *target* of the filter, and C is called the *filter data type*. Again, we omit f and g if they are just identities.

A New Programming Style for ADTs. Let us now consider some metamorphic programs. First of all, examples for algebraic data types translate directly from the corresponding catamorphisms. For instance, the length of a list can be computed by the metamorphism

$$\mathit{length} \;=\; \mathit{List} \overset{I+\pi_2}{\leadsto} \mathit{Nat}$$

We can always save the metamorphism map, here $I + \pi_2$, by selecting a target ADT whose constructor functor agrees with the functor of the source ADT destructor. We can actually calculate the desired ADT as follows.

$$\begin{aligned} \mathit{List} \overset{I+\pi_2}{\leadsto} \mathit{Nat} &= [\![[\mathit{Zero}, \mathit{Succ}] \circ (I + \pi_2), \mathit{out}_{L_A}]\!]_{L_A} && \{ \text{MetaHylo} \} \\ &= [\![[\mathit{Zero} \circ I, \mathit{Succ} \circ \pi_2], \mathit{out}_{L_A}]\!]_{L_A} && \{ \text{sum} \} \end{aligned}$$

Thus, we can define the L_A, N-bialgebra $\mathit{Count} = ([\mathit{Zero}, \mathit{Succ} \circ \pi_2], \mathit{out}_N)$, and we obtain for *length* the modified form:

$$\mathit{length} \;=\; \mathit{List} \leadsto \mathit{Count}$$

In an actual programming environment there will be lots of different ADTs representing many algebra/coalgebra-combinations. We envision a system that supports the writing of metamorphisms by automatically offering sets of functor-matching target ADTs and/or sets of natural transformations that can be used as maps in metamorphisms.

Now let us consider the more interesting case for non-algebraic ADTs. We are eventually able to count the number of elements in a set by:

$$\mathit{count} \;=\; \mathit{Set} \leadsto \mathit{Count}$$

Mapping a function f to all elements of a set can be expressed by:

$$\mathit{mapset} \;=\; \mathit{Set} \overset{I+f\times I}{\leadsto} \mathit{Set} \qquad (= \mathit{Set} \overset{L_A(f)}{\leadsto} \mathit{Set})$$

And we must not forget the factorial function, which can be computed by:

$$\mathit{fac} \;=\; \mathit{Range} \leadsto \mathit{Prod}$$

Filters are very handy in expressing certain algorithms, for example,

$\mathit{List} \leadsto \mathit{Set} \leadsto \mathit{List}$	Remove duplicates
$\mathit{List} \leadsto \mathit{Set} \overset{I+\pi_2}{\leadsto} \mathit{Nat}$	Number of different list elements
$\mathit{List} \leadsto \mathit{Queue} \leadsto \mathit{List}$	List reverse
$\mathit{List} \leadsto \mathit{PQueue} \leadsto \mathit{List}$	Heapsort

We have not defined the ADT *PQueue* for priority queues yet. This can be done similar to *Queue*, except that the destructor selects the smallest instead of the last list element. The reader might wonder whether *Stack* (= *List*) instead of *Queue* should be used in the list reverse example. *Queue* is indeed the proper filter here, since metamorphisms proceed in a "bottom-up" manner (which means for L_A "right-to-left"), that is, the last element of the first list will be inserted first into the queue and will thus be consed last to the target list. One might also object that the given program for reverse is unacceptably inefficient because during the decomposition phase each intermediate queue value is reversed twice thus resulting in a quadratic running time. We will address two ways for optimization in Sect. 5.

Until now we have only worked with types of linear functors. It is clear that all presented concepts also apply to, say, tree-like structures. For these the ADT approach gives a very nice view on divide-and-conquer algorithms.

With a final example we give an impression of the power the metamorphic programming style can offer when the right abstractions are chosen. Suppose we have defined a representation for graphs (based on a suitable functor Gr) and a function *roots* for computing and removing the list of a graph's roots.[4] We can then define an ADT $RootGraph = (in_{Gr}, roots)$, which can be used for realizing topological sorting as follows:[5]

$$topsort \;=\; flatten \circ RootGraph \rightsquigarrow List$$

5 Program Transformation

One interesting property of ADTs is *invertability*:

$$D \text{ is } invertible \quad \Longleftrightarrow \quad \overline{\alpha} \circ \alpha = \mathrm{id}$$

Invertible data types are important, since they can be fused away (see Theorem 1 below). In particular, all algebraic data types are invertible. It is clear that a data type can be invertible only if destructor and constructor have compatible signatures:

Lemma 1. *An F,G-algebra D can be invertible only if $F = G$.* □

Next we consider how program transformation and optimization present themselves in the framework of bialgebras and metamorphisms. First of all, we stress that we can use all of the existing results developed for algebraic data types: we show that fusion of algebraic data types is still possible, and we demonstrate the use of well-known laws in the optimization of metamorphic programs. In addition, we show that the fixed recursion pattern enables specific optimizations for ADTs, and finally we show that the filter programming style offers optimization opportunities that go beyond fusion and even promise asymptotic speed-ups.

[4] We will show a simple relational representation for graphs in Sect. 5. Based on that a function for computing roots is given by $- \circ \langle dom, rng \rangle$.

[5] Note that this does *not* work with the representation of Sect. 5, since it is not possible to represent isolated nodes, which might occur during the graph decomposition. The extension is not difficult, but it is not needed here to understand the point.

A Fusion Law for ADTs. An important property of invertible data types is that they do not have an effect as filter data types, that is, they can be safely omitted from filters.

Theorem 1 (Filter Fusion). *C is invertible* $\implies$ $D \rightsquigarrow C \rightsquigarrow D' = D \rightsquigarrow D'$.

Proof.
$$\begin{aligned}
D \rightsquigarrow C \rightsquigarrow D' &= C \rightsquigarrow D' \circ D \rightsquigarrow C && \{ \text{FilterDef} \} \\
&= [\![\beta, \overline{\varphi}]\!]_H \circ [\![\varphi, \overline{\alpha}]\!]_G && \{ \text{MetaHylo} \} \\
&= [\![\beta, \overline{\varphi}]\!]_G \circ [\![\varphi, \overline{\alpha}]\!]_G && \{ \text{Lemma 1} \} \\
&= [\![\beta, \overline{\alpha}]\!]_G && \{ \text{Assumption, HyloFusion} \} \\
&= D \rightsquigarrow D' && \{ \text{MetaHylo} \}
\end{aligned}$$
□

This is a reformulation of the well-known fusion law for algebraic data types. Its importance lies in the fact that the extension to ADTs and metamorphisms is conservative in the sense that the fusion optimization for algebraic data types is not affected and can still be applied in the extended framework.

Applying Classical Transformations. Assume we represent a graph by a binary relation on integers. We can use the already defined ADT *Set* for this; we call it *Rel* here just for clarity. A simple method for computing the set of all nodes in a graph is then to take the union of the domain and the codomain of the relation, which are defined by two simple metamorphisms:

$$\begin{aligned}
dom &= Rel \overset{I+\pi_1 \times I}{\rightsquigarrow} Set \\
rng &= Rel \overset{I+\pi_2 \times I}{\rightsquigarrow} Set \\
nodes &= \cup \circ \langle dom, rng \rangle
\end{aligned}$$

With this implementation the relation must be traversed twice, once for computing the left components of all pairs and once for computing the right components. With the aid of the so-called *banana-split* law [4]

$$\langle (\![\alpha]\!)_F, (\![\beta]\!)_F \rangle \;=\; (\![\langle \alpha \circ F(\pi_1), \beta \circ F(\pi_2) \rangle]\!)_F$$

we can obtain an improved version of *nodes* in which *dom* and *rng* are computed by a single scan over the relation. First, we expand *dom* and *rng* by (MetaHylo) and (HyloSplit) to:

$$\begin{aligned}
dom &= (\![in_{L_A} \circ I + \pi_1 \times I]\!) \circ [\![deset]\!] \\
rng &= (\![in_{L_A} \circ I + \pi_2 \times I]\!) \circ [\![deset]\!]
\end{aligned}$$

By factorizing the anamorphism from the split we get:

$$nodes \;=\; \cup \circ \langle (\![in_{L_A} \circ I + \pi_1 \times I]\!), (\![in_{L_A} \circ I + \pi_2 \times I]\!) \rangle \circ [\![deset]\!]$$

Now we can apply the banana-split law (with $F = L_A$) and obtain the following optimized version for *nodes*:

$$\begin{aligned} nodes \;=\; & \cup \circ (\!|\langle (in_{L_A} \circ I + \pi_1 \times I) \circ L_A(\pi_1), \\ & \qquad (in_{L_A} \circ I + \pi_2 \times I) \circ L_A(\pi_2)\rangle|\!) \circ [\![deset]\!] \end{aligned}$$

This can be simplified by evaluating L_A and applying laws for product and sum yielding:

$$nodes \;=\; \cup \circ (\!|\langle in_{L_A} \circ I + \pi_1 \times \pi_1, in_{L_A} \circ I + \pi_2 \times \pi_2\rangle|\!) \circ [\![deset]\!]$$

which can be finally written as a metamorphism:

$$nodes \;=\; \cup \circ Rel \overset{\langle I+\pi_1\times\pi_1, I+\pi_2\times\pi_2\rangle}{\rightsquigarrow} (Set \times Set)$$

Depending on the definition of the function $\cup$, we can possibly optimize further. For example, if $\cup$ is itself defined by a catamorphism, we can fuse that definition with the metamorphism just obtained. We do not elaborate on this here, the goal of this part was just to show that optimizations and transformations can be well performed using already existing laws.

Exploiting Fixed Recursion Scheme. We have already noted that the filter for implementing list reverse is unacceptably inefficient, since actually each tail of the list is reversed twice. This gives a quadratic running time, and, no doubt, a direct use of the function *rev* would be much better.[6] But if we look at how the queue is used in a metamorphism, we observe that in each step one element is taken from the queue and the (intermediate) queue values themselves are never needed, except for decomposing/dequeueing. In order to exploit this knowledge we formulate equations for different versions of the queue (q_i) and the dequeued elements (x_i). We abbreviate $\pi_1 \circ out_{L_A}$ by *hd* and $\pi_2 \circ out_{L_A}$ by *tl*. Recall the definition of $dequeue = I + I \times rev \circ out_{L_A} \circ rev$. Now given an (non-empty) intermediate queue q_{i-1}, we have:

$$\begin{aligned} q_i &= (rev \circ \pi_2 \circ out_{L_A} \circ rev)\; q_{i-1} \;=\; (rev \circ tl \circ rev)\; q_{i-1} \\ x_i &= (\pi_1 \circ out_{L_A} \circ rev)\; q_{i-1} \;=\; (hd \circ rev)\; q_{i-1} \end{aligned}$$

Since $q_{i-1} = (rev \circ tl \circ rev)\; q_{i-2}$ we have $q_i = (rev \circ tl \circ rev \circ rev \circ tl \circ rev)\; q_{i-2} = (rev \circ tl^2 \circ rev)\; q_{i-2}$. By induction it follows (given an initial queue q_0) that

$$q_i \;=\; (rev \circ tl^i \circ rev)\; q_0$$

[6] In general, however, we do not know about the implementation of an ADT, and thus we might not have access to a function like *rev*.

Now we observe that the last queue value is *Nil* and that all other queue values are only used for dequeueing. This means that we can work inside the decomposition with reversed queues, that is, using $r_i = rev\ q_i$ we get:

$$\begin{aligned} r_1 &= rev\ q_0 \\ r_i &= rev\ ((rev \circ tl \circ rev)\ q_{i-1}) \ = \ (tl \circ rev)\ q_{i-1} \ = \ tl\ r_{i-1} \\ x_i &= hd\ r_{i-1} \end{aligned}$$

This gives a much more efficient implementation for the *Queue* ADT. In particular, the representing list has to be reversed only once.

Single-Threaded Analysis for Free! Consider the filters *List*⇝*Queue*⇝*List* and *List*⇝*PQueue*⇝*List*. First, from the definition of filter it is clear that: (i) the filter ADTs *Queue* and *PQueue* are completely built up before they are decomposed. Second, from the definition of metamorphism it can be seen that (ii) an ADT is constructed from one generator (the source ADT) where only one version exists at any time, and (iii) an ADT is destructed just from one consumer (the target ADT) thus also maintaining only one version at any time.

Hence at any time only one version of the filter ADT is referenced, and this means that the update operations to be performed on the filter can be safely implemented in an imperative way. This can increase the efficiency of programs much more than fusion is ever able to achieve. We are faced with a twisted situation here: it is not the elimination of data structures that improves the running time of programs, but rather the introduction of filter structures.

The nice thing is that a compiler does not need a sophisticated analysis technique to determine single-threadedness. Selecting update-in-place implementations is particularly important for data types like arrays or graphs, since persistent (= functional) implementations for these can become quite complex [23, 6], and as demonstrated in [8], predefined imperative implementations of fold operations can speed up computations considerably.

6 Related Work

Much of the work concerning catamorphisms on algebraic data types has already been mentioned in the introduction. There is surprisingly little work addressing structured recursion on non-algebraic data types, that is, data types satisfying equational laws. In particular, most approaches deal with specific data types, and there is almost no general framework available that could be used for a large class of abstract data types.

Chuang presents in [5] essentially three different views of arrays and defines for each view corresponding fold operations. Gibbons [12] defines a data type for directed acyclic multi-graphs. With a careful choice of operations, which obey certain algebraic laws, the definition of graph catamorphisms becomes feasible, and some functions on graphs, such as reversing the edges of a graph (graph reversal) or determining shortest paths (measured in number of edges), can be expressed as graph catamorphisms. However, the whole approach is very limited, since it applies only to acyclic graphs having no edge labels.

We have presented a more general view of graphs in [8]. In that paper an important aspect was the definition of a couple of fold operations that can be used to express operations, such as graph reversal, depth first search, evaluation of expression DAGs, or computing all simple paths in a graphs. Two theorems for program fusion were presented that allow the removal of intermediate search trees as well as intermediate graph structures.

The only general approach for expressing catamorphisms over non-free data types we know of is the work of Fokkinga [10, 9]. The idea is to represent terms by combinators called *transformers* and to represent an equation by a pair of transformers. Several properties of transformers are investigated, and it is shown how transformers can be combined to yield new transformers thus resulting in a variable-free language for expressing equations. The use of transformers is demonstrated in showing the equivalence of two different stack implementations. However, the whole approach suffers from the already mentioned restrictions caused by the constraints that homomorphisms must map to quotients.

7 Conclusions

We have demonstrated how the structured recursion programming discipline can be applied to abstract data types. The main idea was to represent ADTs by bialgebras and to express mappings between ADTs by metamorphisms.

Our approach demands the explicit definition of destructors. However, this additional effort pays off, since it offers much freedom in the design of ADTs, in particular, the separation into algebra and coalgebra provides a high degree of modularity. Moreover, it also provides with metamorphisms a much more general computing device than homomorphisms, since we can map into types with less structure.

Nevertheless, metamorphisms on bialgebras are a conservative extension of homomorphisms: the fusion law for algebraic data types is still valid and can be applied for invertible ADTs. Moreover, a very promising property of filter ADTs is that they can be safely implemented in a destructive way without loosing referential transparency, since metamorphisms (and filters) use them in a single-threaded way.

References

[1] J. Backus. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *Communications of the ACM*, 21:613–641, 1978.

[2] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice-Hall International, 1996.

[3] R. S. Bird. Lectures on Constructive Functional Programming. In M. Broy, editor, *Constructive Methods in Computer Science*, NATO ASI Series, Vol. 55, pages 151–216, 1989.

[4] R. S. Bird and O. de Moor. *The Algebra of Programming.* Prentice-Hall International, 1997.

[5] T.-R. Chuang. A Functional Perspective of Array Primitives. In *2nd Fuji Int. Workshop on Functional and Logic Programming*, pages 71–90, 1996.

[6] P. F. Dietz. Fully Persistent Arrays. In *Workshop on Algorithms and Data Structures*, LNCS 382, pages 67–74, 1989.

[7] H. Dybkjær. *Category Theory, Types, and Programming Laguages.* PhD thesis, University of Copenhagen, 1991.

[8] M. Erwig. Functional Programming with Graphs. In *2nd ACM Int. Conf. on Functional Programming*, pages 52–65, 1997.

[9] M. M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, 1992.

[10] M. M. Fokkinga. Datatype Laws without Signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.

[11] T. Fukushima and C. Tuckey. *Charity User Manual*, January 1996.

[12] J. Gibbons. An Initial Algebra Approach to Directed Acyclic Graphs. In *Mathematics of Program Construction*, LNCS 947, pages 282–303, 1995.

[13] A. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. In *Conf. on Functional Programming and Computer Architecture*, pages 223–232, 1993.

[14] T. Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, 1987.

[15] T. Hagino. Codatatypes in ML. *Journal of Symbolic Computation*, 8:629–650, 1993.

[16] Z. Hu, H. Iwasaki, and M. Takeichi. Deriving Structural Hylomorphisms from Recursive Definitions. In *1st ACM Int. Conf. on Functional Programming*, pages 73–82, 1996.

[17] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling Calculation Eliminates Multiple Data Traversals. In *2nd ACM Int. Conf. on Functional Programming*, pages 164–175, 1997.

[18] J. T. Jeuring. *Theories for Algorithm Calculation.* PhD thesis, University of Utrecht, 1993.

[19] D. J. Lehmann and M. B. Smyth. Algebraic Specification of Data Types: A Synthetic Approach. *Mathematical Systems Theory*, 14:97–139, 1991.

[20] G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, LNCS 375, pages 335–347, 1989.

[21] L. Meertens. Algorithmics – Towards Programming as a Mathematical Activity. In *CWI Symp. on Mathematics and Computer Science*, pages 289–334, 1986.

[22] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Conf. on Functional Programming and Computer Architecture*, pages 124–144, 1991.

[23] M. E. O'Neill and F. W. Burton. A New Method for Functional Arrays. *Journal of Functional Programming*, 7(5):487–513, 1997.

[24] J. J. M. M. Rutten. Universal Coalgebra: a Theory of Systems. Report CS-R9652, CWI, Department of Software Technology, 1996.

[25] T. Sheard and L. Fegaras. A Fold for all Seasons. In *Conf. on Functional Programming and Computer Architecture*, pages 233–242, 1993.

[26] A. Takano and E. Meijer. Shortcut Deforestation in Calculational Form. In *Conf. on Functional Programming and Computer Architecture*, pages 306–313, 1995.