Random Access to Abstract Data Types

Martin Erwig

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

Abstract. We show how to define recursion operators for random access data types, that is, ADTs that offer random access to their elements, and how algorithms on arrays and on graphs can be expressed by these operators. The approach is essentially based on a representation of ADTs as bialgebras that allows catamorphisms between ADTs to be defined by composing one ADT's algebra with the other ADT's coalgebra. The extension to indexed data types enables the development of specific recursion schemes, which are, in particular, suited to express a large class of graph algorithms.

Keywords: Category Theory, ADT, Catamorphism, Graph Algorithm

1 Introduction

In [6] we have proposed to model abstract data types as bialgebras, that is, as (algebra, coalgebra)-pairs with a common carrier. In this approach a program on an ADT D can be defined by a mapping to another ADT D', and such a mapping, called *metamorphism*, is essentially given 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 laws developed for algebraic data types can still be used for program transformation and optimization 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 update-in-place implementations for them [7].

ADTs (as well as algebraic data types) are restricted in the sense that the decomposition (order) cannot be controlled from the outside. In other words, the decomposition of ADT values is completely determined by themselves in advance. This makes the treatment of some data types, such as arrays or graphs, difficult. In these data types the decomposition is often controlled by explicitly given indices (respectively, nodes) telling which parts of the ADT value are to be processed next. We call data types that offer such an index access *random access data types* or simply *indexed data types*. Index access behavior can, in

principle, be realized in the ADT approach by appropriately defining new, compound ADTs that contain the ADT values (array, graph) to be indexed as well as the index values. However, this results in rather complex definitions that are difficult to comprehend.

A different solution is proposed in this paper: first, we generalize the definition of ADT to IDT (*indexed data type*). Essentially, this means to extend the argument type of the destructor so that it has explicit access to index values. This leads to a definition of an IDT as a *trialgebra*. Second, the definition of metamorphism is generalized to take into account the use and dynamic generation of index values. This generalization comes in two flavors: first, for data types like arrays having to deal with only one index value at a time, a simple construction, called *exomorphism*, suffices. However, in the more general case, for example, when dealing with graphs, collections of index values must be handled, and this requires a much more involved definition in which two data types, the primary one and an auxiliary one for storing index values, are processed simultaneously. This general mapping is called *synchromorphism*.

The paper is structured as follows: after describing related work in the next section, we briefly review the general categorical approach to data types in Section 3 followed by an introduction to our bialgebra approach to abstract data types in Section 4. The generalization to indexed data types is then described in Section 5. A simple way to map between IDTs with is shown in Section 6, and the development of a more powerful kind of morphisms is presented in Section 7. Conclusions in Section 8 complete the paper.

2 Related Work

The so-called Bird/Meertens formalism [1, 16] is concerned with the derivation of programs from specifications. Essential in that approach is the use of a few powerful operators, like *catamorphisms* (also called *fold* or *reduce*), instead of general recursion. Their work is originally based on lists only, but it has been extended to arbitrary inductively defined data types [15, 17–19, 9]: 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.

Work on program optimization has profited a lot from the categorical approach: when programs are expressed as catamorphisms (or even better as hylomorphisms), powerful fusion laws can be used to eliminate intermediate data structures [13, 19, 20].

The categorical framework has been almost always applied to algebraic data types, that is, data types that are just given by free term structures. The only general approach for expressing catamorphisms over non-free data types we know of is the work of Fokkinga [11, 10]. 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 transform-

ers 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. Although this works for sub-algebras that satisfy certain laws, one cannot map into algebras with less structure [10, 11, 14]. This innocent looking restriction means a rather severe limitation of expressiveness: for instance, a program for counting the elements of a set *cannot* be expressed as a catamorphism. This restriction was lifted by the proposal we made in [6].

Some work has been done for specific abstract data types. Interestingly, these are always indexed data types in our sense: Chuang presents in [3] three different views of arrays and defines for each view corresponding fold operations. The first exploits the indexing facilities of arrays, whereas the second views arrays as sequences. In the third view arrays are treated as mappings. We can recover the first two of these views by an IDT based on appropriate coalgebras, and programs on arrays can be conveniently expressed by exomorphisms.

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, 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 [5]. 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. We can express graph algorithms by synchromorphisms, including depth-first and breadth-first search and even Dijkstra's shortest path algorithm. The view on graph algorithms that is provided by synchromorphisms is similar in spirit to the fixed set of graph exploration operators that were identified in [4].

3 Categorical Data Types

In this section we give a very brief review of the categorical framework for modeling data types. More detailed introductions can be found, for example, in [2, 10, 17, 20]. Examples follow in later sections.

Our 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 recursive equations, such as for hylomorphisms and those of Sections 6 and 7.1. We consider polynomial endofunctors on $\mathcal{C}$ which are built by the four basic functors *identity* ($I\ A = A$ and $I\ f = f$), *constant* ($\underline{A}\ B = A$ and $\underline{A}\ f = \mathrm{id}_A$), *product* ($A \times B = \{(x, y) \mid x \in A, y \in B\}$), and *separated sum* ($A + B = \{1\} \times A \cup \{2\} \times B \cup \{\perp\}$). The definition of $\times$ and $+$ on functions is given below with several additional

operations:

$$\begin{array}{rcl@{\qquad}rcl}
(f+g)\ (1,x) & = & (1, f\ x) & (f \times g)\ (x,y) & = & (f\ x, g\ y) \\
(f+g)\ (2,y) & = & (2, g\ y) & \langle f,g\rangle\ x & = & (f\ x, g\ x) \\
(f+g)\ \bot & = & \bot & \pi_1\ (x,y) & = & x \\
[f,g]\ (1,x) & = & f\ x & \pi_2\ (x,y) & = & y \\
[f,g]\ (2,y) & = & g\ y & \iota_1\ x & = & (1,x) \\
[f,g]\ \bot & = & \bot & \iota_2\ y & = & (2,y)
\end{array}$$

For an object x we denote its constant function by $\underline{x}$, that is, $\underline{x}\ y = x$. (Note also that function application binds strongest, and $\times$ binds stronger than $+$, which in turn binds stronger than composition "$\circ$".)

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 $A \times B$.

We will later need the following functors:

$$\begin{array}{rcl@{\qquad}rcl}
O_A & = & \underline{\mathbf{1}} + \underline{A} & L_A & = & \underline{\mathbf{1}} + \underline{A} \times I \\
P_A & = & \underline{A} \times I & Q & = & \underline{\mathbf{1}} + I \times I
\end{array}$$

We usually denote $(1,x)$ and $(2,x)$ by $\mathrm{L}\,x$ and $\mathrm{R}\,x$, and we use abbreviations, such as $\mathrm{LR}\,x$ for $\mathrm{L}\,(\mathrm{R}\,x)$.

Let $F : \mathcal{C} \to \mathcal{C}$. 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 $\varphi : A \to F(A)$. 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 $h \circ \alpha = \beta \circ F(h)$.

The category of F-algebras $\mathbf{Alg}(F)$ has as objects F-algebras and as arrows F-homomorphisms. 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 $T = U(in_F)$. For example, the algebraic data type of cons-lists with constructors $[Nil, Cons] : L_{list\ A} \to list\ A$ is nothing but the initial algebra in_{L_A}. 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; in_F and out_F are each other's inverses, and they define an isomorphism $T \cong F(T)$ in **CPO**.

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 $\mathbf{Alg}(F)$ there is exactly one F-homomorphism $h : in_F \to \alpha$. Since h is uniquely determined by α, it is conveniently denoted by $(\!|\alpha|\!)_F$; h is called a *catamorphism* [17]. Dually, for each F-coalgebra φ in $\mathbf{CoAlg}(F)$ there is exactly one homomorphism $h : \varphi \to out_F$, which is denoted by $[\!(\varphi)\!]_F$ and which is called an *anamorphism*. A *hylomorphism* is essentially the composition of a catamorphism with an anamorphism.

Formally, a hylomorphism $[\![\alpha, \varphi]\!]_F$ is defined as the least morphism h satisfying:

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

Several laws for {cata, ana, hylo}-morphisms can be found in [17, 20]. The most important result is the fusion rule for hylomorphisms:

$$[\![\alpha, \varphi]\!]_F \circ [\![\beta, \psi]\!]_F \;=\; [\![\alpha, \psi]\!]_F \;\Leftarrow\; \varphi \circ \beta = \mathrm{id} \qquad \text{(HyloFusion)}$$

4 Abstract Data Types and Metamorphisms

We define an ADT to be a pair $(\alpha, \overline{\alpha})$ where α is an F-algebra, $\overline{\alpha}$ is an H-coalgebra, and $U(\alpha) = U(\overline{\alpha})$. Such an algebra/coalgebra-pair with a common carrier is called an F, H-*bialgebra* [11] (where an F, G-bialgebra is a special case of an F, G-*dialgebra*, that is, $\mathbf{BiAlg}(F, G) = \mathbf{DiAlg}([F, I], [I, G])$ [10]. Working with bialgebras is sufficient for our purposes and makes the separation of constructors and destructors more explicit.) Given an ADT $D = (\alpha, \overline{\alpha})$, we call α the *constructor* of D and $\overline{\alpha}$ the *destructor* of D.

Let us consider two 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.

As an example for a non-algebraic type consider an ADT for sets. To define sets based on the "cons"-view given by L_A we take in_{L_A} as the constructor, and the destructor must be defined so that a value is retrieved from a set at most once. This can be realized by splitting off an arbitrary element (for example, the first one) and removing all occurrences of this element in the returned set. With a function *filter* that takes a predicate p and selects a sublist of elements for which p yields true we can first define a further function *remove*:

$$\mathit{remove}(x, l) \;=\; \mathit{filter}\ (\neq x)\ l$$

Here, the partial application $(\neq x)$ denotes the function $\lambda y.y \neq x$, that is, the predicate that yields true for all values that are not equal to x.

Thus, we can define the set destructor and the set ADT by:

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

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

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

Given a natural transformation $f : H \dot{\to} M$, the f-*metamorphism* from D to D' is defined as the least solution of the equation

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

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:

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

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

$$D \rightsquigarrow D' \;=\; (\!|\varphi|\!)_H \quad \Leftarrow \quad D = (in_H, out_H) \qquad \text{(MetaAlg)}$$

Let us consider a few examples. Metamorphisms for algebraic data types translate directly from the corresponding catamorphisms. For instance, if we represent the natural numbers by $Nat = [Zero, Succ] = in_{\underline{1}+I}$, the length of a list can be computed by the metamorphism

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

Since metamorphisms are based on explicitly defined destructors, we can also count the number of elements in a set:

$$card \;=\; Set \overset{I+\pi_2}{\rightsquigarrow} Nat$$

The composition of two metamorphisms $C \rightsquigarrow D'$ and $D \rightsquigarrow C$ filters the values of D "through" C before putting them into D'. We thus define the *C-filter* from D to D' as:

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

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.

ADT filters provide a convenient way for expressing certain algorithms, for example,

$List \rightsquigarrow Set \rightsquigarrow List$	Remove duplicates
$List \rightsquigarrow Heap \rightsquigarrow List$	Heapsort

As for algebraic data types there are several laws for ADTs, see [6, 7]. One important result is a generalization of the fusion law for algebraic data types (recall that $C = (\beta, \overline{\beta})$):

Theorem 1 (ADT Fusion). $\overline{\beta} \circ \beta = \mathrm{id} \implies D \rightsquigarrow C \rightsquigarrow D' = D \rightsquigarrow D'$ □

Another very general relationship can be obtained by deriving the "free theorem" [21] for the type of metamorphisms.

Theorem 2 (FreeMeta). *If l is strict, then for any two F, H-bialgebras $D = (\alpha, \overline{\alpha})$ and $D' = (\alpha, \overline{\gamma})$ and two M, N-bialgebras $C = (\varphi, \overline{\varphi})$ and $C' = (\delta, \overline{\varphi})$:*

$$l \circ \varphi = \delta \circ H(l) \;\wedge\; \overline{\gamma} \circ r = H(r) \circ \overline{\alpha} \implies l \circ (D \rightsquigarrow C) = (D' \rightsquigarrow C') \circ r \qquad \square$$

This general law can be instantiated to many different useful program transformation rules (see [7]).

5 Indexed Data Types

When an ADT is processed by a metamorphism, the decomposition is completely controlled by the ADT itself, that is, the definition of the coalgebra completely determines the decomposition order. (The same is, of course, true for algebraic data types where catamorphisms just follow the term construction.) For some applications, however, it is very useful to have external control over the decomposition of the involved ADT. Consider the simple task of deleting a specific number x from an integer set s. Of course, we can express this by a set-catamorphism that selects from s all elements that are not equal to x, but an even simpler solution is, instead of blindly decomposing all numbers from s, to directly ask for the specific decomposition (x, s'). Then the result is simply given by s'.

This example raises several issues: first, the destructor of such an ADT is not any more simply of type $A \to H(A)$, but rather of type $G(A) \to H(A)$ to account for additional arguments ("indices") for the decomposition. We will therefore extend the definition of ADT into an "IDT". Second, the requested decomposition might not be possible at all, for example, in the above example x might not be contained in s. This affects the definition of mappings from such IDTs, which has to handle such cases. Finally, we need a way to specify how index values are generated during (or fed into) the decomposition process. In the simplest case the next index can be computed by a function parameter; we will consider this case in Section 6. The more general case is treated in Section 7.

We start by generalizing the definition of ADT. An *indexed data type* (*IDT*) is a pair $D = (\alpha, \overline{\alpha})$ where α is an F-algebra, $\overline{\alpha}$ is a G, H-dialgebra, and $U(\alpha) = U(\overline{\alpha})$. We call D an *F, G, H-trialgebra*. Again, α is the *constructor* and $\overline{\alpha}$ is the *destructor* of D. As an example consider the above set ADT with random access to its elements: we have $\alpha = in_{L_A}$ and $\overline{\alpha} = \mathit{extract}$ with

$$\begin{aligned} \mathit{split}(x, s) &= \langle \mathit{filter}\ (= x), \mathit{filter}\ (\neq x) \rangle\ s \\ \mathit{extract}(x, s) &= \begin{cases} \mathrm{L}\,() & \text{if } \pi_1 \circ \mathit{split}(x, s) = \mathit{Nil} \\ \mathrm{R}\,((\mathit{hd} \times I \circ \mathit{split})\ (x, s)) & \text{otherwise} \end{cases} \end{aligned}$$

In this example we have $F = H = L_A$ and $G = P_A$. The example also illustrates that we use the term "index" in a rather broad sense: an index can be any value that controls the decomposition of an IDT. Hence, "index" is just a name for a particular role of a type.

Arrays are probably the most prominent representatives of IDTs; they are typically used whenever indexed access is needed. A simple array constructor is given by $in_{L(X \times A)}$ where X and A denote the index type and the type of stored elements. The treatment of duplicate index entries can happen within the array destructor. A simple approach is to define a function *dearr* that takes an index i and an array a (which is represented by a list of pairs) and returns the first pair (i, x) and the array without all those pairs (j, x') for which $j = i$. (This realizes the behavior that newer entries in the array "overwrite" older ones.) If the index

is not contained in the array, the unit value $() : \mathbf{1}$ is returned. Thus, *dearr* is a $P_X, L_{(X\times A)}$-dialgebra, and the IDT defined by $Array = (in_{L_{(X\times A)}}, dearr)$ is an $L_{(X\times A)}, P_X, L_{(X\times A)}$-trialgebra.

Graphs are another example for IDTs. In the inductive view of directed graphs we have proposed in [5] graphs can be constructed by two constructors: *empty*, which denotes the empty graph without any nodes, and *embed*, which extends a graph by a *node context*, that is, a labeled node together with its incoming and outgoing edges. To stay with polynomial functors we need a functor $\_^{(k)}$ for denoting lists of length not greater than k: $X^{(0)} = \mathbf{1}$ and $X^{(k+1)} = \mathbf{1} + X^{(k)} \times X$.

Now the type of node contexts for node type X and label type Y is given by the following bifunctor:

$$Ctx(X,Y) \;=\; X^{(k)} \times X \times Y \times X^{(k)}$$

that is, a four-tuple consisting of a list of nodes (predecessors), a node, a label, and another list of nodes (successors). The type of graphs of bounded in- and out-degree of k is then defined by the following ternary functor:

$$Gr(X,Y,G) \;=\; \mathbf{1} + (Ctx(X,Y) \times G)$$

Then the graph constructor is given by a $Gr_{X,Y}$-algebra [*empty*, *embed*]. (For a precise semantics of *empty* and *embed*, see [5].)

The graph destructor *degraph* essentially retrieves and removes a specific node context from the graph. This means, given a node x and a graph g, *degraph*(x, g) returns a pair (c, g') where $c = (p, x, l, s)$ is the context of x and where g' is g without x and its incident edges. If x is not contained in g, *degraph* yields $()$. Thus, $degraph : X \times G \to Gr_{X,Y}(G)$, and we obtain a graph IDT by the $Gr_{X,Y}, P_X, Gr_{X,Y}$-trialgebra

$$Graph = ([empty, embed], degraph)$$

6 Exomorphisms

The recursion in a metamorphism h is realized by applying $H(h)$ to the result of $\overline{\alpha}$ which works fine because $\overline{\alpha}$ has type $A \to H(A)$. Since the destructor of an IDT is a G, H-dialgebra, that is, $\overline{\alpha} : G(A) \to H(A)$, we *cannot* simply express the recursion by $H(h) \circ \overline{\alpha}$ since $\overline{\alpha}$, and thus h, too, applies to $G(A)$-values and not simply A-values. Therefore, we have to prepare the recursion by first applying a function $g : H(A) \to H(G(A))$ which, in fact, supplies index values for all the recursive occurrences of A-values.

Let $D = (\alpha, \overline{\alpha})$ be an F, G, H-trialgebra with $A = U(\alpha) = U(\overline{\alpha})$, and let $D' = (\varphi, \overline{\varphi})$ be an M, N-bialgebra. Given two functions

$$f \;:\; H(C) \to M(C) \qquad\qquad g \;:\; H(A) \to H(G(A))$$

we define the *exomorphism* from D to D' as the least morphism satisfying:

$$h \;=\; \varphi \circ f \circ H(h) \circ g \circ \overline{\alpha}$$

We denote h by $D \overset{f}{\underset{g}{\rightsquigarrow}} D'$. Since g is a parameter of the exomorphism, it provides control over the IDT decomposition from the "outside".

As a simple example, consider a store of linked lists implemented with arrays: each array cell consists of a pair (x, p) representing a cell where p is an integer pointing to the next cell. With the function $g = I + \langle \pi_1, \langle \pi_2 \circ \pi_1, \pi_2 \rangle \rangle$ that pairs the pointer of the decomposed array cell ($\pi_2 \circ \pi_1$) with the remaining array (π_2) and pairs this with the found list entry (π_1), we can retrieve the list stored in A beginning at position i by

$$(\mathit{Array} \overset{}{\underset{g}{\rightsquigarrow}} \mathit{List})\ (i, A)$$

7 Synchromorphisms

We have already seen that the next index value depends, in general, on preceding decompositions. This means that index generation happens dynamically; it must be performed "on the fly" during the decomposition of the IDT, hence, the sequence of indices is generally not known in advance.

The limitations of exomorphisms are mainly due to their inability to handle more than one index at a time, that is, we are missing an option to intermediately store collections of indices. Now ADTs themselves are suited very well for this index buffering, and when we are going to define a recursion scheme for IDTs in Section 7.1, this will in fact turn out to be a scheme for processing the IDT with the buffer ADT hand in hand. In Section 7.2 we present some examples.

7.1 Buffered Decomposition of IDTs

A synchromorphism takes three arguments: a source IDT, a target ADT, and a buffer ADT for storing and delivering index values. A synchromorphism informally works as follows: the IDT is decomposed, and (i) from the result some fresh index values are computed that are inserted into the buffer ADT, and (ii) a part of the result is inserted into the target ADT. Immediately after that the buffer is requested to yield a new index which is then used in the next iteration to decompose the remaining IDT-value.

Let $D = (\alpha, \overline{\alpha})$ be an F, G, H-trialgebra with $A = U(\alpha) = U(\overline{\alpha})$, let $D'' = (\beta, \overline{\beta})$ be a K, J-bialgebra with $B = U(\beta) = U(\overline{\beta})$, and let $D' = (\varphi, \overline{\varphi})$ be an M, N-bialgebra with $C = U(\varphi) = U(\overline{\varphi})$. X is the type of index values. It is shared between the types of D and D'', and we assume that all functors F, G, H, K and J are left sections of bifunctors having X fixed as their first argument.

Recall the roles of the functors: F and G define the argument type of the source ADT constructor and destructor, respectively, and H defines the result type of the source ADT destructor. K defines the argument type of the buffer ADT constructor, and J defines the result type of the buffer ADT destructor. This means that D and D'' carry index values, whereas the target ADT D' does, in general, not.

In the following we use variable names that indicate their type: for example, x_G is an element of $G(A)$, and x_{HG} an element of $H(G(A))$. We develop the definition of synchromorphisms step by step, collecting requirements and incrementally fixing design decisions. The construction is summarized in Figure 1.

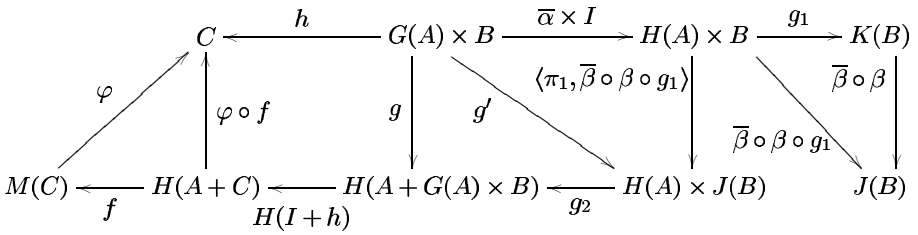

Fig. 1. Categorical Definition of Synchromorphisms.

First, a synchromorphism (h) takes an IDT-argument and a buffer and produces a value of the target ADT. Therefore, h has the following type:

$$h : G(A) \times B \to C$$

Since the IDT-destructor yields an element x_H, we have to apply a function g to x_H to enable the recursive application of the synchromorphism. After the recursive application by H we apply a function f extracting relevant information to be aggregated by the constructor φ of the target ADT D'.

Next we explain how to obtain a suitable definition for g. The synchromorphism has to perform the following steps:

1. Initially, decompose the IDT with the supplied index, that is, $x_H = \overline{\alpha}(x_G)$.

2. Extract fresh index values j from x_H to be inserted into the buffer. How this should be done is application-specific, and it is specified by a parameter function g_1.

3. Insert the fresh index values into the buffer, and retrieve the next index value(s) i from the buffer for further decomposition of the IDT . We thus obtain something like

$$i = \ldots \circ \overline{\beta} \circ \beta \circ g_1(x_H)$$

The dots "..." indicate that $\overline{\beta}$ actually yields a value x_J of which i is, in general, only a part.

Note that g_1 not only has to extract fresh index values, but also has to arrange them properly around the buffer so that β can be applied. Thus, the result must be of type $K(B)$, and we get:

$$g_1 \ : \ H(A) \times B \to K(B)$$

We can now compose all three steps. With:

$$\begin{aligned} \overline{\alpha} \times I \ &: \ G(A) \times B \to H(A) \times B \\ \overline{\beta} \circ \beta \circ g_1 \ &: \ H(A) \times B \to J(B) \end{aligned}$$

we obtain $g' = \langle \pi_1, \overline{\beta} \circ \beta \circ g_1 \rangle \circ \overline{\alpha} \times I$ of type

$$g' \ : \ G(A) \times B \to H(A) \times J(B)$$

Now two things remain to be done:

4. Combine the remaining IDT(s) d (as part of x_H) and the next index values i into a value x_{HG} that allows d to be decomposed indexed by i (in a recursion structure specified by H). Again this is application-specific and should therefore be specified by a parameter function, say, $g_2' : H(A) \times J(B) \to H(G(A))$.

5. Select the resulting buffer b (from x_J) for distribution into x_{HG}, the structure containing the remaining IDT/next index combinations. The selection is directed by the application and requires a further function, say, $g_2'' : J(B) \to B$ (distribution into x_{HG} could be achieved by H).

Now a problem occurs if x_J does not contain a buffer value at all. This usually will occur if the buffer is exhausted (then x_J will be, for example, ()). Now in order to not complicate the typings further it seems best to combine g_2' and g_2'' into one function: $g_2 : H(A) \times J(B) \to H(G(A) \times B)$ that is supplied by the programmer and that handles all the above cases internally.

But as x_J is not guaranteed to contain buffer values, it might as well fail to produce next index values (again, for example, in the case the buffer is exhausted). In that case we cannot distribute a buffer, and we cannot even build a value x_{GH}. Then we simply pass x_H so that the value can be used by f and φ. Thus we have $g_2 : H(A) \times J(B) \to H(A) + H(G(A) \times B)$ which can be also written by moving the sum into H:

$$g_2 : H(A) \times J(B) \to H(A + G(A) \times B)$$

We have not yet discussed the case when $\overline{\alpha}$ fails to produce a new IDT value. This could well happen if the required index decomposition is not possible. In that case the old, undecomposed IDT value (from x_G) should be taken and combined with the buffer (if available, otherwise the recursion stops). But since x_G is not available any more, the easiest solution is to rely on appropriately adapted definitions of H and $\overline{\alpha}$, that is, instead of simply returning (), $\overline{\alpha}$ could well be defined to return its argument unchanged whenever decomposition is not possible.

Then (the type of) g_2 need not be changed, and the necessary transformation has to specified in the definition of g_2. To summarize, there are four cases to be considered by g_2 depending on the success of $\overline{\alpha}$ and $\overline{\beta}$ in producing new IDT, respectively, index/buffer values:

$\overline{\alpha}$	$\overline{\beta}$	description of case
fail	fail	immediately stop recursion
fail	ok	preserve (old) x_G and continue recursion
ok	fail	just pass x_H and stop decomposition
ok	ok	normal recursion

Whenever $\overline{\beta}$ is successful, the result type of g_2 is $H(G(A) \times B)$, otherwise it is $H(A)$.

Now we can formally define a synchromorphism. Given the functions

$$\begin{aligned} f &: H(A+C) \to M(C) \\ g_1 &: H(A) \times B \to K(B) \\ g_2 &: H(A) \times J(B) \to H(A + G(A) \times B) \end{aligned}$$

the *D, D''-synchromorphism to D'* is defined the least solution of the following equation

$$h \;=\; \varphi \circ f \circ H(I+h) \circ g$$

where

$$g \;=\; g_2 \circ \langle \pi_1, \overline{\beta} \circ \beta \circ g_1 \rangle \circ \overline{\alpha} \times I$$

We denote the synchromorphism h by $D' \xleftarrow{f} D \overset{g_1}{\underset{g_2}{\rightleftarrows}} D''$.

7.2 Examples

Let us begin with expressing depth-first search (dfs) as a synchromorphism. Roughly spoken, dfs decomposes a graph by extracting a particular node context c, pushing the successors from c onto a stack, and extracting the top of the stack to continue graph decomposition. In addition, part of c is aggregated in a target ADT, for example, the visited nodes are put into a list.

Thus, we need a stack buffer with a constructor that can insert lists of nodes. We can use the following ADT defined as a Q, L_X-bialgebra ($+\!\!+$ is the function for concatenating two lists):

$$\mathit{Stack} = ([\mathit{Nil}, +\!\!+], \mathit{out}_{L_X})$$

We can use the ADT *List* for collecting visited nodes, but we have to account for the case that a visited node is not available for insertion into the result list whenever the graph decomposition fails. Therefore, we use the "option" or "maybe"

type O_A to wrap nodes. Finally, we cannot directly use the $Gr_{X,Y}, P_X, Gr_{X,Y}$-trialgebra *Graph* from Section 5 as IDT since we require the destructor to pass the argument graph if destruction is not possible. We therefore redefine

$$\begin{aligned} \mathit{degraph}'(x,g) &= (\underline{g} + I)\ (\mathit{degraph}(x,g)) \\ \mathit{Graph} &= ([\mathit{empty}, \mathit{embed}], \mathit{degraph}') \end{aligned}$$

To summarize we have the functors $G = P_X$, $H = I + Ctx_{X,Y} \times I$, $K = Q$, $J = L_X$, and $M = L_{\mathbf{1}+X}$ $(= \underline{\mathbf{1}} + (\underline{\mathbf{1}} + \underline{X}) \times I)$. This gives the following types (A, B, and C are the carriers of graphs, stacks, and lists, and X and Y are the types of nodes and node labels):

$$\begin{aligned} G(A) &= X \times A & \qquad H(A) &= A + Ctx_{X,Y} \times A \\ K(B) &= \mathbf{1} + B \times B & \qquad J(B) &= \mathbf{1} + X \times B \\ & & \qquad M(C) &= \mathbf{1} + (\mathbf{1} + X) \times C \end{aligned}$$

Next we define the functions g_1, g_2 and f: g_1 pairs the successors (π_4) of the extracted context (π_1) or an empty list with the buffer and therefore always returns the second alternative (ι_2) of $K(B)$.

$$g_1 = \iota_2 \circ ([\underline{\mathit{Nil}}, \pi_4 \circ \pi_1]) \times I$$

g_2 combines the remaining graph and the next index and distributes fresh indices into the remaining stack. It also has to preserve decomposed values for insertion into D'; g_2 actually controls the different cases of the recursion:

(i) If graph decomposition has failed and the stack is exhausted, the recursion is stopped; the graph, which is passed only for typing reasons, will be ignored by f.

(ii) If graph decomposition failed and a new index i is available, proceed with decomposing the old graph (delivered by the modified $\mathit{degraph}'$) at i.

(iii) If graph decomposition yields context c and remaining graph g but the stack is exhausted, terminate recursion, and pass c so that the last visited node can be put into the target list.

(iv) If graph decomposition yields context c and remaining graph g and a new index i is available – this is the "normal" recursion case –, pass c to let f extract the visited node, and continue recursion with decomposing g at i.

For readability we provide a pointwise definition of g_2.

$$\begin{aligned} g_2\ (\mathrm{L}\, g, \mathrm{L}\, ()) &= \mathrm{LL}\, g & \qquad g_2\ (\mathrm{L}\, g, \mathrm{R}\, (i,s)) &= \mathrm{LR}\, ((i,g),s) \\ g_2\ (\mathrm{R}\, (c,g), \mathrm{L}\, ()) &= \mathrm{R}\, (c, \mathrm{L}\, g) & \qquad g_2\ (\mathrm{R}\, (c,g), \mathrm{R}\, (i,s)) &= \mathrm{R}\, (c, \mathrm{R}\, ((i,g),s)) \end{aligned}$$

Finally, the definition for f follows the structure of results yielded by g_2: (i) on termination, a unit value is returned which is mapped by the target ADT into *Nil*. (ii) If no context is available, build a pair of $\mathrm{L}\, ()$ and the recursively computed list of nodes. The "none" value $()$ will be inserted into this list and

can be eventually removed by applying a post-processing function keeping only the nodes. (iii,iv) Extract the visited node from the context and pair it with the empty list (on termination (iii)) or the recursively computed list (in case (iv)).

$$\begin{aligned} f\ (\text{LL}\, g) &= \text{L}\,() & f\ (\text{LR}\, l) &= \text{R}\,(\text{L}\,(), l) \\ f\ (\text{R}\,(c, \text{L}\, g)) &= \text{R}\,(\text{R}\,(\pi_2(c)), \mathit{Nil}) & f\ (\text{R}\,(c, \text{R}\, l)) &= \text{R}\,(\text{R}\,(\pi_2(c)), l) \end{aligned}$$

Now we can define dfs as follows:

$$\mathit{dfs} = \mathit{List} \underset{f}{\leftarrow} \mathit{Graph} \overset{g_1}{\underset{g_2}{\rightleftarrows}} \mathit{Stack}$$

It is also obvious how to express breadth-first search: we can just substitute a queue buffer for the stack buffer, and we obtain:

$$\mathit{bfs} = \mathit{List} \underset{f}{\leftarrow} \mathit{Graph} \overset{g_1}{\underset{g_2}{\rightleftarrows}} \mathit{Queue}$$

We can also express more complex algorithms, for example, Dijkstra's algorithm for finding shortest paths. This is shown in the long version of this paper [8].

We have only shown graph algorithms as examples for synchromorphisms (further examples are: Prim's minimum spanning tree algorithm and Kruskal's minimum spanning tree algorithm), and in fact, expressing graph algorithms as instances of a fixed recursion scheme was the main motivation for developing synchromorphisms. However, we believe that there are different application areas. For example, the plane-sweep paradigm for algorithms of computational geometry seems to fit the presented scheme: the buffer ADT can be used to implement the sweep-line status structure, and the IDT is a collection of geometric objects (which, however, is scanned in fixed order most of the time so that indexed access is not always needed).

8 Conclusions

We have demonstrated how to extend categorical abstract data types to indexed data types, and we have shown definitions of recursion operators operating on these data types. With these combinators we can now express algorithms that use data types in a random access manner.

The next step is to investigate the transformation of such algorithms into efficient programs. This can go along the same line as in [7] by introducing libraries of efficient ADT implementations and defining simple optimizing transformations that automatically select these implementations.

References

1. 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.

2. R. S. Bird and O. de Moor. *The Algebra of Programming.* Prentice-Hall International, 1997.
3. T.-R. Chuang. A Functional Perspective of Array Primitives. In *2nd Fuji Int. Workshop on Functional and Logic Programming*, pages 71–90, 1996.
4. M. Erwig. Graph Algorithms = Iteration + Data Structures? The Structure of Graph Algorithms and a Corresponding Style of Programming. In *18th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, LNCS 657, pages 277–292, 1992.
5. M. Erwig. Functional Programming with Graphs. In *2nd ACM Int. Conf. on Functional Programming*, pages 52–65, 1997.
6. M. Erwig. Categorical Programming with Abstract Data Types. In *7th Int. Conf. on Algebraic Methodology and Software Technology*, LNCS 1548, pages 406–421, 1998.
7. M. Erwig. The Categorical Imperative – Or: How to Hide Your State Monads. In *10th Int. Workshop on Implementation of Functional Languages*, pages 1–25, 1998.
8. M. Erwig. Random Access to Abstract Data Types. Technical Report 266, Fern-Universität Hagen, 2000.
9. L. Fegaras and T. Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions. In *23rd ACM Symp. on Principles of Programming Languages*, pages 284–294, 1996.
10. M. M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, University of Twente, 1992.
11. M. M. Fokkinga. Datatype Laws without Signatures. *Mathematical Structures in Computer Science*, 6:1–32, 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. J. T. Jeuring. *Theories for Algorithm Calculation.* PhD thesis, University of Utrecht, 1993.
15. G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, LNCS 375, pages 335–347, 1989.
16. L. Meertens. Algorithmics – Towards Programming as a Mathematical Activity. In *CWI Symp. on Mathematics and Computer Science*, pages 289–334, 1986.
17. 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.
18. E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Conf. on Functional Programming and Computer Architecture*, pages 324–333, 1995.
19. T. Sheard and L. Fegaras. A Fold for all Seasons. In *Conf. on Functional Programming and Computer Architecture*, pages 233–242, 1993.
20. A. Takano and E. Meijer. Shortcut Deforestation in Calculational Form. In *Conf. on Functional Programming and Computer Architecture*, pages 306–313, 1995.
21. P. Wadler. Theorems for Free! In *Conf. on Functional Programming and Computer Architecture*, pages 347–359, 1989.