Non Homomorphic Reductions of Data Structures

Luis Antonio Galán, Manuel Núñez, Cristóbal Pareja, Ricardo Peña
e-mail: {lagalan,manuelnu,cpareja,ricardo}@dia.ucm.es

Departamento de Informática y Automática
Universidad Complutense de Madrid
fax: (34-1) 394 4607 ph: (34-1) 394 4429
E-28040 Madrid. Spain

Abstract

In this paper we study different kinds of *reductions* of data types. By reduction we mean applying the higher order function *fold* to a data structure. An appropriate *fold* function can be defined for any recursive data type. These reductions have been presented as homomorphisms by several authors [2, 6]. Although many useful functions on data structures can be programmed as instances of *fold*, there are some that cannot. This is due to the fact that they are not mathematical homomorphisms. We show some examples of these functions. Then, we introduce two generalizations of *fold* (one for lists and the other for binary trees) in terms of which many non homomorphic mappings can be defined. Some examples are presented.

A second problem addressed in the paper is the relationship between the definitions of some particular reductions in different data types. We show that the definition of a particular reduction, e.g. to insert an element in a data structure, in terms of *fold* (either the generalized version or the usual one), looks the same for different data structures. In many cases, one can be obtained by transforming the other. We define a hierarchy of data types according to the amount of "structural" information they possess. It is shown how some reductions of a data structure A with more structural information than another one B can be obtained by composing the homomorphism from A to B that "forgets" structural information, with the homomorphism reducing B.

1 Introduction

There are two ideas that have been frequently stressed in transformational functional programming: the use of object-free functions [1] and the definition of a small set of higher order functions (*map, map2, foldr, filter*, etc.) that can cope with a large number of situations (e.g. the Bird-Merteens formalism [2]). Modern functional languages such as SML, Miranda, Haskell and Gofer [8, 7, 4, 5] are examples of this. The advantage of using these functions is twofold: in one hand, they encapsulate well

known recursion patterns, avoiding the use of explicit recursion in situations matching these patterns. In the other hand, they satisfy a number of algebraic laws allowing the transformation of programs in a secure manner, from inefficient clear formulations to more-efficient, less-clear ones [2, 3].

Among these patterns, the function *fold* deserves special mention. The intuition behind it is to *reduce* a polymorphic data structure of type $T\ \alpha$ into a value of type β. In particular, β may coincide with $T\ \alpha$. Many functions getting information from a data structure or modifying it (in the sense of producing a modified copy of it), can be programmed as instances of *fold*. In fact, other higher order functions such as *map* and *filter* can be defined in terms of *fold*. In [6] the *fold* function is characterized as a homomorphism. However, not every function $f : T\ \alpha \longrightarrow \beta$ is a homomorphism. We present some examples of non homomorphic funtions on lists and trees.

Looking for a common pattern for these non homomorphic functions, we propose a modified version of *fold* for lists, named *nhl* in the paper, and another for binary trees, named *nht*, allowing to express some common non homomorphic accesses to these structures. Also some homomorphic functions very complex to express as such, can be defined in a simple way using these new reductions.

A second problem we address in the paper is the relationship between reductions in different data types. We follow an approach that considers a data structure as formed by two components: the data elements and the "structure" itself. We define a hierarchy of data types according to the quantity of "structural" information they possess. We show that some reductions of data structures with more structural information can be expressed as the composition of two homomorphisms: the one that forgets all or part of the structural information, and the one that reduces the target structure.

Prosecuting this idea, we compare the definition of some interesting reductions (inserting and deleting an element in a data structure) for different data structures. We find out the definitions so similar, that all of them could be obtained by transforming the reduction of the structure with no structural information (sets and multisets).

The organization of the paper is as follows: after this introduction, in section 2 we review some of the work done in the last few years on homomorphisms. We present some examples of homomorphisms over lists, sets, multisets and binary trees. In section 3 we present some non homomorphic functions on lists and binary search trees, and introduce our generalization of the *fold* function for these two structures. In particular, it is shown how several variants of insertion and deletion can be easily expressed in terms of the new higher order functions. Section 4 is devoted to presenting the hierarchy of data structures and the transformations between reductions of different data structures. Finally, section 5 provides a short conclusion and summarizes the lines along which this work can be continued in the near future.

2 Homomorphic reductions

Homomorphisms [2, 3, 6] constitute a uniform way of expressing functions that distribute (*promote* in words of R. Bird) their activity both to the data elements and to

the constructors of a data structure. Since most of the interesting data structures have recursive constructors, homomorphisms encapsulate a recursion scheme consisting of applying to all the substructures the same function applied to the main structure. This scheme is used by many functions working on the structure.

Definition 2.1 A function h defined on a data structure of elements of type α, (denoted $T\ \alpha$) is a *homomorphism* if there exist m reductor functions $\oplus_{K_i}$ $(1 \leq i \leq m)$, one for every constructor K_i, such that h is defined as:

$$\begin{aligned} h\ &:: T\ \alpha \to \beta \\ h\ (K_i\ e_1 \ldots e_{r_i}) &= \oplus_{K_i} e'_1 \ldots e'_{r_i} \end{aligned}$$

where $e'_j = \begin{cases} e_j & \text{, if } e_j :: \alpha \\ h\ e_j & \text{, if } e_j :: T\ \alpha \end{cases}$
We will use the notation $h = H(\oplus_{K_1} \ldots \oplus_{K_m})$. □

2.1 Homomorphisms on lists

In most functional languages, lists are recursively defined in terms of two constructors: the empty list, denoted $[\,]$, and the *Cons* constructor denoted $\_ : \_$.

$$list_r\ \alpha = [\,] \mid (:)\ \alpha\ (list_r\ \alpha)$$

We call these lists *forward lists* after [6] and, since they are reduced from right to left, we denote them by $list_r$. The homomorphisms over $list_r$ are then denotated $H(\oplus_:, \oplus_{[\,]})$. When referring to any kind of lists we will use HL instead of H. The symmetric definition (a constructor adding an element to the right of the list) leads us to the corresponding *backwards lists*, denoted $list_l$ as they are reduced from left to right. In FP [1] *sequences* encapsulate both types of lists and mechanisms for accessing and modifying both ends of a list are provided by the language.

The most important higher order function on lists is the reduction. It is called *foldr* (in Miranda and Haskell) for $list_r$, and *foldl* for $list_l$[1]. In general, the operator $(\oplus)$ is not associative . When it is so, and using Bird notation, $fold$ is denotated $\oplus/$. It is defined as follows:

Definition 2.2 Let $(\oplus) : \alpha \to \alpha \to \alpha$ be associative and with neutral element $id_\oplus$. We define:

$$\begin{aligned} /\ &:: (\alpha \to \alpha \to \alpha) \to [\alpha] \to \alpha \\ \oplus/\ [x_1 \ldots x_n] &= x_1 \oplus \ldots \oplus x_n \\ \oplus/\ [\,] &= id_\oplus \end{aligned}$$

□

Bird considers lists as monoids (*join-lists* in [6]). The base cases are the empty list and the unit list. The recursive case is the list constructed by appending two

[1] Usually, *foldl* is defined in terms of the forward constructor (:). However, the reduction conceptually corresponds to a definition in terms of the backwards constructor.

lists with the $+\!\!+$ operator which is associative and has the empty list as its neutral element. We will denote them $list_m$.

$$list_m\ \alpha = [\,] \mid [.]\alpha \mid (+\!\!+)\ (list_m\alpha)\ (list_m\alpha)$$

The main advantage of $list_m$ is its generality: $list_r$ and $list_l$ are particular cases of it. In contrast, they have the drawback of not having free constructors, so the following laws must be settled amongst them:

$$\begin{array}{ll} \forall x, y, z \in list_m & (x +\!\!+ y) +\!\!+ z = x +\!\!+ (y +\!\!+ z) \\ & x +\!\!+ [\,] = [\,] +\!\!+ x = x \end{array}$$

Homomorphisms associated to $list_m$ must preserve the associativity of $+\!\!+$. A mapping h over $list_m$ is a homomorphism if there exists an associative function $\oplus_{+\!\!+} : \beta \to \beta \to \beta$ and functions $\oplus_{[.]} : \alpha \to \beta$ and $\oplus_{[\,]} : \beta$ such that

$$\begin{array}{l} h(l_1 +\!\!+ l_2) = (h\ l_1) \oplus_{+\!\!+} (h\ l_2) \\ h\ [x] = \oplus_{[.]}\ x \\ h\ [\,] = \oplus_{[\,]} \end{array}$$

The main law satisfied by these homomorphisms is the following:

$$h \text{ homomorphism over } list_m \Leftrightarrow h = \oplus/.\ f*$$

where $f*$ denotes the expression $map\ f$.

We will abbreviate after Bird $HL(\oplus_{+\!\!+}, \oplus_{[.]}, \oplus_{[\,]})$ by $B(\oplus, f)$, and we will call these homomorphisms B-homomorphisms, being $\oplus = \oplus_{+\!\!+}$, $f = \oplus_{[.]}$ and $id_{\oplus_{+\!\!+}} = \oplus_{[\,]}$[2]. It is inmmediate to express any B-homomorphisms as an $HL(\oplus_:, \oplus_{[\,]})$.

A large number of functions over finite lists can be expressed as B-homomorphisms:

$$\begin{array}{rcl} f* & = & B(+\!\!+, [.].f) \\ \oplus/ & = & B(\oplus, id_\alpha) \\ p \triangleleft & = & B(+\!\!+, p \to [.]; K[\,]) \\ all\ p & = & B(\wedge, p) \\ some\ p & = & B(\vee, p) \\ length & = & B(+, K1) \\ reverse & = & B(\tilde{+\!\!+}, [.]) \\ mergesort & = & B(merge, [.]) \\ id_{[\alpha]} & = & B(+\!\!+, [.]) \end{array}$$

where $K\ x\ y = x$, $merge$ merges two ordered lists, $a\ \tilde{+\!\!+}\ b = b +\!\!+ a$ and

$$(p \to f; g)\ x = \begin{cases} f\ x, & \text{if } p\ x \text{ holds} \\ g\ x, & \text{otherwise} \end{cases}$$

It must be noted that B-homomorphisms are strictly less general than forward list homomorphisms $HL(\oplus_:, \oplus_{[\,]})$ as the following example shows.

[2] When $id_{\oplus_{+\!\!+}}$ doesn't exist, the domain excludes the empty list.

Example 2.1 We wish to compute the integer represented by a list of digits, each one in the range 0..9, being the most significant digit the rightmost one.

$$\begin{aligned} integer &= HL(\oplus_{:}, 0) \\ &\quad \textbf{where } x \oplus_{:} s = s * 10 + x \end{aligned}$$ □

This function cannot be expressed as a B-homomorphism, because there is no way (with a homomorphism) of recording the number of zeroes in the middle of the computation.

2.2 Homomorphisms over other usual monoids

Homomorphisms can be easily generalized to other data structures. The main requirement for them is that the reductor operators must preserve the constructor laws.

Multisets are commutative monoids. Their algebraic definition is as follows:

$$mset\ \alpha = \{\!\}\ |\ \{\!.\!\}\alpha\ |\ (\biguplus)\ (mset\ \alpha)\ (mset\ \alpha)$$

where the binary constructor $\biguplus$ satisfies the associative and commutative laws. As a consequence, homomorphisms h over multisets must provide an associative, commutative reductor $\oplus_{\biguplus}$ and satisfy:

$$h(m_1 \biguplus m_2) = (h\ m_1) \oplus_{\biguplus} (h\ m_2)$$

For instance, the cardinal of a multiset, or the sum of all their elements are homomorphisms. In both cases, $\oplus_{\biguplus}$ is the $+$ operator over natural numbers.

Sets are commutative, idempotent monoids. Their algebraic definition is:

$$set\ \alpha = \{\ \}\ |\ \{.\}\ \alpha\ |\ (\cup)\ (set\ \alpha)\ (set\ \alpha)$$

where the binary constructor $\cup$ satisfies the associative, commutative and idempotent laws. As a consequence, homomorphisms h over sets must provide an associative, commutative, idempotent reductor $\oplus_{\cup}$ and satisfy:

$$h(s_1 \cup s_2) = (h\ s_1) \oplus_{\cup} (h\ s_2)$$

Functions computing the maximum element of a set, or the greatest common divisor of all of them can be expressed as homomorphisms over sets. Also, the predicates using *all* p, such as the section $(\subset S)$ for a given S, defined as $(\subset S) = all\ (\in S)$, or those using *some* p, such as the section $(x \in)$, defined as $(x \in) = some\ (x =)$, are homomorphisms.

2.3 Homomorphisms over binary trees

In this section we introduce the algebraic definition of binary trees and their associated homomorphisms:

$$Tree\ \alpha = \triangle\ |\ \bullet\ (Tree\ \alpha)\ \alpha\ (Tree\ \alpha)$$

Then, homomorphisms over binary trees need the existence of two reductors.

Definition 2.3 We say that a function $h : \text{Tree}\ \alpha \to \beta$ is a *homomorphism* if there exist reductor functions $\oplus_\bullet : \beta \to \alpha \to \beta \to \beta$ and $\oplus_\triangle : \beta$ such that:

$$\begin{array}{rcl} h\ \triangle & = & \oplus_\triangle \\ h\ (\bullet\ l\ x\ r) & = & \oplus_\bullet\ (h\ l)\ x\ (h\ r) \end{array}$$

We will use the notation $h = HT(\oplus_\bullet, \oplus_\triangle)$ to denotate these homomorphisms. □

A number of homomorphic functions over lists have their homolog ones over trees:

$$\begin{array}{rcl} f* & = & HT(\oplus_\bullet, \triangle) \\ & & \textbf{where}\ \oplus_\bullet\ l\ x\ r = \bullet\ l\ (f\ x)\ r \\ all\ p & = & HT(\bigwedge, True) \\ & & \textbf{where}\ \bigwedge\ b_l\ x\ b_r = (p\ x) \wedge b_l \wedge b_r \\ some\ p & = & HT(\bigvee, False) \\ & & \textbf{where}\ \bigvee\ b_l\ x\ b_r = (p\ x) \vee b_l \vee b_r \\ size & = & HT(\oplus_\bullet, 0) \\ & & \textbf{where}\ \oplus_\bullet\ i_l\ x\ i_r = i_l + 1 + i_r \\ height & = & HT(\oplus_\bullet, (-1)) \\ & & \textbf{where}\ \oplus_\bullet\ i_l\ x\ i_r = 1 + max\ i_l\ i_r \\ flatten & = & HT(+\!\!+_3, [\,]) \\ & & \textbf{where}\ +\!\!+_3\ l\ x\ r = l +\!\!+ [x] +\!\!+ r \\ id_{Tree\ \alpha} & = & HT(\bullet, \triangle) \end{array}$$

3 Non homomorphic reductions

In this section, two higher order functions are defined —one for forward lists and another one for binary trees— that generalize common non homomorphic functions on these structures.

The underlying idea on these functions comes from realizing that many functions on a structure, first search along the structure looking for some property to hold; then some special treatment is done, and then the rest of the structure is ignored. In terms of reduction, this idea can be interpreted as follows: the structure is divided into two parts, the active one and the passive one with the aid of one or more predicates. Reducing the active part amounts to say that the reductor operator does the search, and if the search succeeds applies the special treatment. Reducing the passive part consists of applying a different reductor operator, which in many cases simply copies the structure.

We are specially interested in those operations that return a structure of the same type as the original one, i.e. we pay attention to insertion and deletion operations.

3.1 Lists

We study ordered and unordered lists, with and without multiple copies of elements. For *ordered lists without repetitions* both operations are homomorphisms. Their definitions follow.

Example 3.1

$$
\begin{aligned}
&\text{insert x} = \text{B}(\oplus,\text{f}) \\
&\qquad \textbf{where f y} = [\text{x,y}] \ , \textbf{ if x}<\text{y} \\
&\qquad\qquad\quad = [\text{y,x}] \ , \textbf{ if x}>\text{y} \\
&\qquad\qquad\quad = [\text{y}] \quad , \textbf{ if y}=\text{x} \\
&\qquad\qquad (\text{l1}+\!\!+[\text{x1}]) \oplus ([\text{x2}]+\!\!+\text{l2}) = \text{l1}+\!\!+[\text{x2}]+\!\!+\text{l2} \ , \textbf{ if x1}=\text{x} \\
&\qquad\qquad\qquad\qquad\qquad\quad = \text{l1}+\!\!+[\text{x1}]+\!\!+\text{l2} \ , \textbf{ if x2}=\text{x}
\end{aligned}
$$

$$\text{delete x} = (\text{x}\neq) \triangleleft$$ □

We see that expressing *insert* as a homomorphism is little intuitive: we first insert a copy of x either to the right or to the left of *each* element. Then we remove all the copies except the one that must remain.

For *unordered lists without repetitions* both operations are also homomorphisms. In fact, *delete* is the same function of example 3.1. The *insert* function is the following homomorphism:

Example 3.2

$$
\begin{aligned}
&\text{insert x} = \text{B}(\oplus, \text{f}) \\
&\qquad \textbf{where f y} = [\text{x,y}] \ , \textbf{ if x}\neq\text{y} \\
&\qquad\qquad\quad = [\text{y}] \quad , \textbf{otherwise} \\
&\qquad\quad \text{l1} \oplus ([\text{x2}]+\!\!+\text{l2}) = \text{l1}+\!\!+\text{l2}
\end{aligned}
$$ □

Again, the technique consists of inserting a copy of x for every element in the list not equal to x, and then removing all the copies except the leftmost one.

However, for *lists with repetitions* some operations are not homomorphisms. Insertion of a new copy indeed it is, and can be defined by slightly modifying the **otherwise** clause of example 3.2. But to *delete* the leftmost copy of a value x is not a homomorphism as the following lemma shows.

Lemma 3.1 The function that deletes the leftmost copy of a value x in an ordered list with repetitions *is not* a homomorphism.

Proof. Let us assume that h implements that function an h is a homomorphism. Then, there exists a reductor $\oplus$ such that $h\ (y\!:\!ys) = y \oplus h\ ys$.

Let xs be any list without occurrences of the value x. Then, $h\ xs = xs$, and $h\ (x\!:\!xs) = xs$. But $h\ (x\!:\!xs) = x \oplus h\ xs$, then $x \oplus xs = xs$. As a consequence, we have $h\ (x\!:\!x\!:\!xs) = x \oplus h\ (x\!:\!xs) = x \oplus xs = xs$, in contradiction with what the hypothesis says: that h only deletes one copy of x. □

For the case of unordered lists the proof is very similar, but it must be noted that $h\ xs$ needs not be equal to xs but to a permutation of the elements of xs.

Despite the fact that some of these functions are not homomorphisms or, if they are, it is difficult to express them as such, it is possible to define a homomorphism, named *split*, which does part of the work of *insert* and *delete*. What remains to be done is simple enough to make the programming of these two functions an easy task. The homomorphism *split*, defined below, applies a predicate q to the list and splits it into two lists in such a way that the first element of the second list (in the case the list is not empty) is the leftmost element of the original list satisfying q. For example, *split* (>5) [1,3,5,7,4]=[[1,3,5],[7,4]].

Definition 3.1

$$\begin{aligned}\text{split q} = \text{B}(\oplus,\text{f})\\ \textbf{where f x} = [[\,],[\text{x}]]\ ,\ \textbf{if q x}\\ = [[\text{x}],[\,]]\ ,\ \textbf{otherwise}\\ [\text{l1},[\,]] \oplus [\text{l2},\text{m2}] = [\text{l1}+\!\!+\text{l2},\ \text{m2}]\\ [\text{l1},\text{m1}] \oplus [\text{l2},\text{m2}] = [\text{l1},\text{m1}+\!\!+\text{l2}+\!\!+\text{m2}]\end{aligned}$$

□

Let us remark that the neutral element of operation $\oplus$ in this case is $[[\,],[\,]]$. With the aid of *split*, deletion of the leftmost copy of x can be programmed as:

$$\begin{aligned}\text{delete x} = \text{del.(split (x=))}\\ \textbf{where del } [\text{l1 , x:l2}] = \text{l1} +\!\!+ \text{l2}\\ \text{del } [\text{l1 , }[\,]] = \text{l1}\end{aligned}$$

Unfortunately, *split* is not a solution for all the problems concerning insertion and deletion. For instance, the homomorphism that deletes all the copies of an element is not easily constructed with *split*. Besides, it is not clear how to generalize *split* to other data structures. So, we define a more general non homomorphic reduction for lists based on the idea presented at the beginning of this section.

Definition 3.2 Non Homomorphic reduction function for Lists (*nhl*):

$$\begin{aligned}\text{nhl q } \oplus_a \oplus_d \text{ b } [\,] &= \text{b}\\ \text{nhl q } \oplus_a \oplus_d \text{ b (x:xs)} &= \text{x } \oplus_a \text{ (nhl q } \oplus \oplus_d \text{ b xs)}\\ &\quad \textbf{where } \oplus = \oplus_a\ ,\ \textbf{if } \neg\ (\text{q x})\\ &\qquad\quad = \oplus_d\ ,\ \textbf{otherwise}\end{aligned}$$

□

Reductor operators $\oplus_a$ (active) and $\oplus_d$ (default or passive) correspond to the (:) constructor, and b corresponds to the $[\,]$ constructor. We will use the following notation which mimics that used for homomorphisms:

$$\text{nhl } q\ \oplus_a \oplus_d\ b = \text{NHL}(q,\oplus_:^a,\oplus_:^d,\oplus_{[\,]})$$

The function *nhl* "inserts" an active reductor after each element of the list including the first, if any, satisfying the q predicate. From that point to the end of the list, *nhl* inserts the passive reductor. The following expression pictures this idea:

$$\begin{aligned}&\text{nhl q } \oplus_a \oplus_d \text{ b } [x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_{n-1}, x_n] =\\ &x_1 \oplus_a (x_2 \oplus_a (\ldots x_i \oplus_a (x_{i+1} \oplus_d (\ldots x_{n-1} \oplus_d (x_n \oplus_d b)\ldots)\end{aligned}$$

being $i = min\{1 \ldots n\}$ such that $q\ x_i$ holds.

Trivially, any homomorphism $HL(\oplus_:, \oplus_{[\,]})$ on forward lists can be expressed as the non homomorphic reduction $NHL(q, \oplus_:^a, \oplus_:^d, \oplus_{[\,]})$ making $\oplus_:^a = \oplus_:^d = \oplus_:$, and being q any boolean function. Some awkward homomorphic functions can be elegantly expressed as a non homomorphic reduction. For instance, inserting a value x in an ordered list without repetitions is now:

Example 3.3

$$\begin{aligned}\text{insert x} = \text{nhl } (\geq \text{x}) \oplus (:)\ [\,]\\ \textbf{where y} \oplus \text{ys} = \text{x:y:ys}\ ,\ \textbf{if x} < \text{y}\\ = \text{y:x:[]}\ ,\ \textbf{if ys=[]} \wedge \text{x} > \text{y}\\ = \text{y:ys}\quad ,\ \textbf{otherwise}\end{aligned}$$

□

We assume that the argument list is non empty. If it is empty, then *insert* x $[]= [x]$. The other insertion functions can be similarly programmed. For deletion, the following function is enough for all the cases presented in this section.

Example 3.4

$$\begin{aligned}\text{delete x} = \text{nhl } (=\text{x}) \oplus (:)\ [\,] \\ \textbf{where } \text{y} \oplus \text{ys} = \text{ys} \quad &, \textbf{if } \text{x} = \text{y} \\ = \text{y:ys} \ &, \textbf{otherwise}\end{aligned}$$

□

Let us note in these examples that the passive reductor is (:) and the base case is [], so they reconstruct (or copy) the original list.

3.2 Binary trees

For these structure, we only consider binary search trees without multiple occurrences of elements. The treatment of multiple occurrences does not present special problems in this structure.

The non homomorphic reduction we are proposing here encapsulates as a particular case the usual recursion scheme on binary search trees: the search propagates either to the left or to the right subtree, never to both of them at the same time.

Definition 3.3 Non Homomorphic reduction function for Trees (*nht*):

$$\begin{aligned}&\text{nht ql qr } \oplus_a \oplus_d \text{ b } \triangle = \text{b} \\ &\text{nht ql qr } \oplus_a \oplus_d \text{ b } (\bullet\ l\ x\ r) = \oplus_a \text{ (nht ql qr } \oplus_l \oplus_d \text{ b l) x (nht ql qr } \oplus_r \oplus_d \text{ b r)} \\ &\qquad \textbf{where } \oplus_l = \oplus_a \ , \textbf{if } \neg \text{ (ql x)} \\ &\qquad\qquad = \oplus_d \ , \textbf{otherwise} \\ &\qquad\quad \oplus_r = \oplus_a \ , \textbf{if } \neg\text{(qr x)} \\ &\qquad\qquad = \oplus_d \ , \textbf{otherwise}\end{aligned}$$

□

Predicates ql and qr trigger the change from active to passive mode in the corresponding subtrees, the first time they hold. The active part of the tree extends from the root to the nearest elements satisfying ql or qr. The figure 1 shows an example of execution.

Reductor operators $\oplus_a$ and $\oplus_d$ correspond to the $\bullet$ constructor, and the base case b corresponds to the $\triangle$ constructor. We will use then the following notation:

$$\begin{aligned}&\text{nht } ql\ qr\ \oplus_a\ \oplus_d\ b = \\ &\text{NHT}(ql, qr, \oplus_\bullet^a, \oplus_\bullet^d, \oplus_\triangle)\end{aligned}$$

Like in the case of lists, we could express the insertion and deletion as homomorphisms. Unfortunately, the resulting functions are not so easy to understand. By using the non homomorphic reduction, we see below that the result is very simple.

Example 3.5

$$\begin{aligned}\text{insert x} = \text{nht } (\leq\text{x})\ (\geq\text{x}) \oplus \bullet\ \triangle \\ \textbf{where } \oplus\ \text{l y r} = \bullet\ \text{l y } (\bullet\ \triangle\ \text{x}\ \triangle) \ &, \textbf{if } \text{r}{=}\triangle \wedge \text{x}{>}\text{y} \\ = \bullet\ (\bullet\ \triangle\ \text{x}\ \triangle)\ \text{y r} \ &, \textbf{if } \text{l}{=}\triangle \wedge \text{x}{<}\text{y} \\ = \bullet\ \text{l y r} \qquad\quad &, \textbf{otherwise}\end{aligned}$$

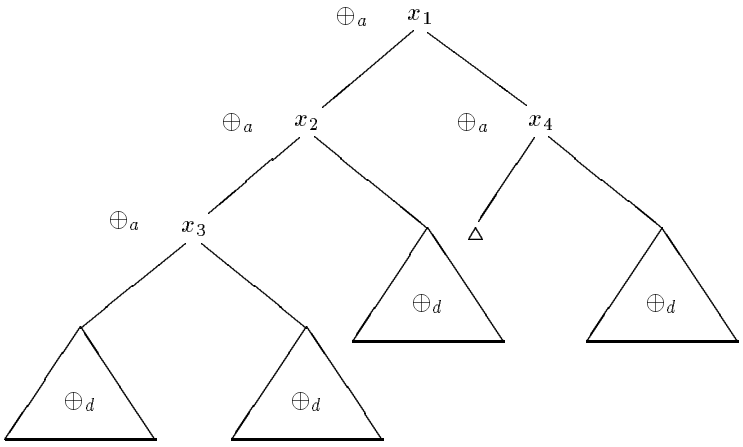

Figure 1: An example of *nht* execution.

□

Example 3.6

$$\begin{aligned} \text{delete x} = \text{nht } (\leq \text{x}) \ (\geq \text{x}) \ \oplus \bullet \triangle \\ \textbf{where} \ \oplus \text{ l y r} \quad &= \bullet \text{ l y r, } \textbf{if } \text{x} \neq \text{y} \\ \oplus \triangle \text{ y r} \quad &= \text{r} \\ \oplus \text{ l y r} \quad &= \bullet \text{ (delmax l) (selmax l) r} \end{aligned}$$

□

The functions *delmax* and *selmax*, respectively deletes and selects the maximum element of a tree and could also be expressed as NHT reductions.

4 Transformations between reductions

Any data structure, in particular those studied in the previous sections, may be considered as composed of two disjoint pieces of information:

- Its *contents*, i.e. the data elements stored in it. We will refer to them as its set *set* α or multiset *mset* α of elements of type α.
- The *structure* itself, i.e. the spatial disposition of the elements inside the structure. We will call *structural information* to this aspect of the data structure.

According to this view, we classify data structures based on the *amount* of structural information they contain. We establish a hierarchy in which structures on top of it contain the least structural information (i.e. sets and multisets), and structures at the bottom contain the most. The aspect of this hierarchy is shown in figure 2.

Let us note that, in this hierarchy, ordered lists are located above unordered lists. The idea is to be able to define unique homomorphisms from structures located at low levels of the hierarchy to structures located at higher levels. The meaning of these homomorphisms is "lost of structural information". They are indicated in the figure as filled arrows. The homomorphism going from unordered lists to ordered

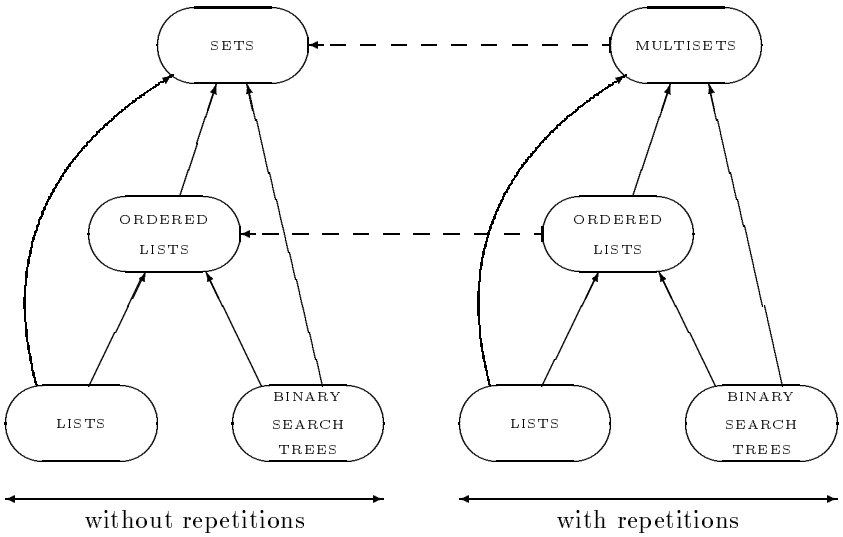

Figure 2: data structure hierarchy

ones means "sort the list",and it is an n to 1 mapping. The dotted arrows of figure 2 correspond to the homomorphisms "eliminate multiple copies of all the elements". In principle, as they represent lost of information about the data elements, we are not very interested on them.

To speak properly about homomorphisms it is mandatory to say which are the correspondent operations in both domains. For instance, in the homomorphisms from unordered lists to ordered ones, $+\!\!+$ in the first domain is the homolog of *merge* in the second domain. Sometimes we need to add appropriate operations in one of the domains to be able to define the homomorphism. For instance, the one from binary search trees without repetitions to sets can be defined:

$$\psi_{ts} = \mathrm{HT}(\oplus_\bullet, \{\})$$
$$\textbf{where } \oplus_\bullet \text{ sl x sr} = \text{sl} \cup \{\text{x}\} \cup \text{sr}$$

So, we define an operation on sets $\cup^{\{\}}\cup : set \to elem \to set \to set$ that joins two sets and one element producing a new set. Then, this new operation corresponds in the homomorphism to the binary tree constructor $\bullet$.

The homomorphism going from unordered lists with possible repetitions to multisets is defined as follows:

$$\psi_{lm} = \mathrm{HL}(\cup, \{\!|.|\!\}, \{\!|\,|\!\})$$

In this case we use the normal constructors of multisets as homolog operations of the constructors of $list_m$.

Many of the data structures reductions studied in previous sections depend on the contents but not on the structural information. This is the case when computing the maximum element of a binary tree or the sum of all the elements of a list. In these cases, the reduction to apply can be obtained as the composition of two homomorphisms: the one abstracting the structural information and the one reducing

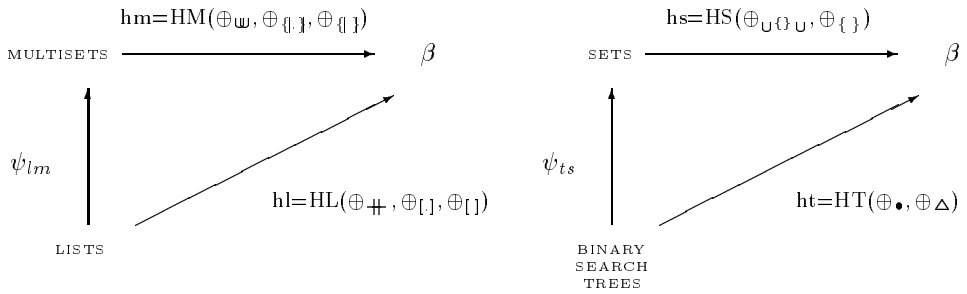

Figure 3: Composition of homomorphic reductions

the resulting set or multiset. We denote by ψ_{xy} the homomorphism abstracting (a part of) the structural information from a structure x to a structure y. For instance, ψ_{lm} reduces unordered lists with repetitions to multisets. The diagrams of figure 3 show how homomorphisms compose. There, $HM(\oplus_{\uplus}, \oplus_{\{\!\{.\}\!\}}, \oplus_{\{\!\{\}\!\}})$ reduces multisets with constructors homolog to those of lists, and $HS(\oplus_{\cup\{\}\cup}, \oplus_{\{\,\}})$ reduces sets with constructors homolog to those of binary trees.

Let us imagine a homomorphism $hl = HL(\oplus_{+\!\!+}, \oplus_{[.]}, \oplus_{[\,]})$ from lists to a value of type β which is independent of the structural information of lists. If we express hl as a composition of homomorphisms, the reductor operators relate each other in the following way:

$$\begin{array}{rcccl}
\oplus_{[\,]} & = & hl\,[\,] & = & (hm.\psi_{lm})\,[\,] = hm\,\{\!\{\}\!\} = \oplus_{\{\!\{\}\!\}} \\
\oplus_{[.]}x & = & hl\,[x] & = & (hm.\psi_{lm})\,[x] = hm\,\{\!\{x\}\!\} = \oplus_{\{\!\{.\}\!\}}x \\
hl\;xs \oplus_{+\!\!+} hl\;ys & = & hl\,(xs +\!\!+ ys) & = & (hm.\psi_{lm})\,(xs +\!\!+ ys) \\
 & & & = & hm\,(\psi_{lm}\;xs \uplus \psi_{lm}\;ys) \\
 & & & = & (hm.\psi_{lm})\;xs \oplus_{\uplus} (hm.\psi_{lm})\;ys \\
 & & & = & hl\;xs \oplus_{\uplus} hl\;ys
\end{array}$$

Then, it holds that $\oplus_{[\,]} = \oplus_{\{\!\{\}\!\}}, \oplus_{[.]} = \oplus_{\{\!\{.\}\!\}}$ and $\oplus_{+\!\!+} = \oplus_{\uplus}$. Let us note that $\oplus_{\uplus}$ is associative and commutative as it is a reductor operator of multisets, and so homolog to $\cup$ constructor.

Using the constructors for sets that are homolog to those of binary trees, we can define the reduction on sets as the following homomorphism $hs = HS(\oplus_{\cup\{\}\cup}, \oplus_{\{\,\}})$

$$\begin{array}{rcl}
hs\;\{\,\} & = & \oplus_{\{\,\}} \\
hs\;(\cup^{\{\}} \cup\, s1\;x\;s2) & = & \oplus_{\cup\{\}\cup}\;(hs\;s1)\;x\;(hs\;s2)
\end{array}$$

If we wish to reduce a tree into a value of type β by using a structure-independent homomorphism $ht = HT(\oplus_{\bullet}, \oplus_{\triangle})$, we can do it by composing the two homomorphisms ψ_{ts} and hs. The reductor operations relate each other as follows:

$$\begin{array}{rcccl}
\oplus_{\triangle} & = & ht\;\triangle & = & (hs.\psi_{ts})\;\triangle = hs\{\,\} = \oplus_{\{\,\}} \\
\oplus_{\bullet}\;(ht\;l)\;x\;(ht\;r) & = & ht(\bullet l\;x\;r) & = & (hs.\psi_{ts})\;(\bullet l\;x\;r) \\
 & & & = & hs(\cup^{\{\}}(\psi_{ts}\;l)\;x\;(\psi_{ts}\;r)) \\
 & & & = & \oplus_{\cup\{\}\cup}\;((hs.\psi_{ts})\;l)\;x\;((hs.\psi_{ts})\;r) \\
 & & & = & \oplus_{\cup\{\}\cup}(ht\;l)\;x\;(ht\;r)
\end{array}$$

That is, $\oplus_\bullet = \oplus_{\cup\{\}\cup}$ and $\oplus_\triangle = \oplus_{\{\}}$.

In other cases, the reduction of the data structure depends only on the structural information, or on both the contents and the structural information. An example of the first type is to compute the height of a tree. An example of the second type is the example 2.1 for lists.

In particular, very interesting are those reductions giving as a result a structure of the same type as the one being reduced, as it is the case with the insertion and deletion operations studied in section 3. In these cases, the reductor operator cannot add essential properties (such as commutativity or idempotency) over the properties already satisfied by the constructor. Normally, the reductor will be a slight variation of the constructor.

We wish to detect common patterns in these types of reductions by comparing how they behave for different data structures.

4.1 Lists

We use the free constructors on forward lists $[\,]$ and $\_ : \_$ and consider unordered multiple-copies ones. Abstracting the structural information we get multisets. To establish a homomorphism we add the $\_\hookrightarrow\_$ operation to multisets. It adds an element to a multiset and satisfies the following *permutative* law:

$$x\hookrightarrow(y\hookrightarrow m) = y\hookrightarrow(x\hookrightarrow m)$$

The reduction homomorphism for multisets is defined then:

$$\begin{aligned} hm\ \{\!\{\,\}\!\} &= \oplus_{\{\!\{\,\}\!\}} \\ hm\ (x\hookrightarrow m) &= x \oplus_\hookrightarrow hm\ m \end{aligned}$$

where $\oplus_\hookrightarrow$ must satisfy the permutative law.

Let us consider inserting in an unordered multiple-copies list. We choose to insert at the end of it:

$$\begin{aligned} \text{insert}_l\ \text{x} = \text{HL}(\oplus_:, [\,]) \\ \textbf{where}\ \text{y} \oplus_:\ [\,] = \text{y:x:}[\,] \\ \text{y} \oplus_:\ \text{ys} = \text{y:ys} \end{aligned}$$

If we interpret to insert "at the end" of a multiset as inserting before inserting any other element, we obtain:

$$\begin{aligned} \text{insert}_m\ \text{x} = \text{HM}(\oplus_\hookrightarrow, \{\!\{\,\}\!\}) \\ \textbf{where}\ \text{y} \oplus_\hookrightarrow \{\!\{\,\}\!\} = \text{y}\hookrightarrow\text{x}\hookrightarrow\{\!\{\,\}\!\} \\ \text{y} \oplus_\hookrightarrow\ \text{m} = \text{y}\hookrightarrow\text{m} \end{aligned}$$

The structure of boths reductions is identical. This similarity remains when we insert in ordered lists by using the non homomorphic reductor *nhl* (see example 3.3).

Deletion of all the copies of an element in ordered lists with repetitions is the following homomorphism:

$$\begin{aligned} \text{delete}_l\ \text{x} = \text{HL}(\oplus_:, [\,]) \\ \textbf{where}\ \text{y} \oplus_:\ \text{m} = \text{m} \quad , \textbf{if}\ \text{x=y} \\ = \text{y:m}\ , \textbf{otherwise} \end{aligned}$$

The version for multisets is:

$$\begin{aligned}\text{delete}_m\ \text{x} = \text{HM}(\oplus_{\hookrightarrow}, \{\!|\,|\!\}) \\ \textbf{where}\ \text{y}\ \oplus_{\hookrightarrow}\ \text{m} &= \text{m} && , \textbf{if}\ \text{x=y} \\ &= \text{y} \hookrightarrow \text{m} && , \textbf{otherwise}\end{aligned}$$

The structures of both reductions are again identical. The main difference is that the multiset reduction can be done in any order, taking into account the permutative law that $\hookrightarrow$ satisfies. In forward lists, reduction will always be done from right to left. We obtain the same similarity if we compare *delete* for lists without repetitions with *delete* in a set. However, *delete* the first copy in lists with repetitions is not a homomorphism as it has been shown in lemma 3.1. We can use here the non homomorphic reduction:

$$\begin{aligned}\text{delete x} = \text{nhl}\ (=\text{x})\ \oplus_a\ (:)\ [\,] \\ \textbf{where}\ \text{y}\ \oplus_a\ \text{ys} &= \text{ys} && , \textbf{if}\ \text{x=y} \\ &= \text{y:ys} && , \textbf{otherwise}\end{aligned}$$

We observe great similarity with deletion of all the copies. The role of the $(= x)$ predicate consists only of stopping the deletion of copies once the first one has been deleted.

4.2 Binary search trees

We consider trees without repetitions and sets with the special constructors that are homomorphic to the tree constructors.

In example 3.5 it is shown how to insert in a binary search tree. The corresponding version for sets is:

$$\begin{aligned}\text{insert}_s\ \text{x} = \text{HS}(\oplus_{\cup\{\}\cup}, \{\,\}) \\ \textbf{where}\ \oplus_{\cup\{\}\cup}\ \text{s1 y s2} &= \cup^{\{\}}\cup\ \text{s1 y}\ (\cup^{\{\}}\cup\ \{\,\}\ \text{x}\ \{\,\}) && , \textbf{if}\ \text{s2}=\{\,\} \\ &= \cup^{\{\}}\cup\ (\cup^{\{\}}\cup\ \{\,\}\ \text{x}\ \{\,\})\ \text{y s2} && , \textbf{if}\ \text{s1}=\{\,\} \\ &= \cup^{\{\}}\cup\ \text{s1 y s2} && , \textbf{otherwise}\end{aligned}$$

It is obvious to see that this is only one of the possibilities. We can freely permute elements in a set. We have chosen the possibility that takes into account the order present in search trees. Then, we can say that the reduction of a search tree can be obtained by adding to the reduction of the corresponding set some hypothesis about the order of the elements (i.e. by taking into account the structural information of search trees).

In example 3.6 it is shown how to delete an element from a binary search tree. The corresponding version for sets is:

$$\begin{aligned}\text{delete}_s\ \text{x} = HS(\oplus_{\cup\{\}\cup}, \{\,\}) \\ \textbf{where}\ \oplus_{\cup\{\}\cup}\ \text{s1 y s2} &= \cup^{\{\}}\cup\ \text{s1 y s2}, \textbf{if}\ \text{x}\neq\text{y} \\ \oplus_{\cup\{\}\cup}\ \{\,\}\ \text{y s2} &= \text{s2} \\ \oplus_{\cup\{\}\cup}\ \text{s1 y s2} &= \cup^{\{\}}\cup\ (\text{delone s1})\ (\text{selone s1})\ \text{s2}\end{aligned}$$

where *delone s1* deletes the element that *selone s1* selects. The similarity is very clear again. The order hypothesis in search trees forces us to select always the maximum of the left subtree. In sets we are free to choose any element.

æ

5 Conclusion

We have studied in detail two kinds of higher order functions which reduce two common data structures, lists and binary trees, either in a homomorphical or in a non homomorphical way. They encapsulate a powerful recursion scheme so that many usual functions on these structures can be easily expressed in terms of them. Present work is devoted to generalize this idea to other data structures.

Also, we have presented a way of comparing reductions in a hierarchy of data structures differing in the degree of structural information they have. This idea constitutes the basis for transforming algorithms expressed in terms of the structures with less structural information to algorithms for the structures with more structural information.

Acknowledgements

We would like to express our gratitude to Margarita Bradley and Pedro Palao for their help while preparing this paper and to an anonymous referee who pointed out some errors in the draft version. æ

References

[1] J.W. Backus. Can functional programming be liberated from the Von Neumann style? *Communications of the ACM*, 21:613–641, 1978.

[2] R. S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer Verlag. NATO ASI Series, vol. F36, 1987.

[3] R. S. Bird. Lectures on constructive functional programming. In *Constructive Methods in Computer Science*, pages 151–216. Springer Verlag. NATO ASI Series, vol. F55, 1989.

[4] A. J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press, 1992.

[5] M. P. Jones. *GOFER*. Oxford University Computing Laboratory, 1992.

[6] G. Malcolm. Homomorphisms and promotability. In *Mathematics of Program Construction. LNCS 375*, pages 335–347. Springer-Verlag, 1989.

[7] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *LNCS 201*, pages 1–16, Berlin, 1985. Springer-Verlag.

[8] A. Wikström. *Functional Programming Using Standard ML*. Prentice-Hall, 1987.