

Metamorphic Programming: Structured Recursion for Abstract Data Types

Martin Erwig

FernUniversität Hagen, Praktische Informatik IV

58084 Hagen, Germany

erwig@fernuni-hagen.de

Abstract

We extend the structured recursive programming discipline, which favors the use of fold operations in place of general recursion, to abstract data types. The key idea is to represent an ADT by two parts, a *constructor* and a *destructor*, which are essentially functions to/from a common representation. Then a fold can work on an ADT by applying parameter functions to values that are delivered by the ADT's own destructor. Fold operations that use as a parameter the constructor of another ADT, called *ADT transformers*, play an important role and offer a concise programming style. We present some laws for ADT folds and transformers and show their use in program optimization and verification.

1 Introduction

The structured use of recursion means to employ a fixed set of operators instead of arbitrary recursion in defining functions. The most important such operator is *fold* (also known as *reduce* or *catamorphism*), which is essentially a homomorphism from an algebraic (or free) data type to another (arbitrary) data type. Fold offers a canonical way of consuming a data structure; it can be thought of as a function replacing the constructors of the consumed value by its parameter functions.

The fold operation is well-known for lists, but it can be easily generalized to regular data types [19, 21, 24, 22, 11]. Fold operations encapsulate a fixed pattern of recursion, which makes programming with them attractive for several reasons: (i) in most cases, termination is guaranteed, (ii) powerful program transformations become possible by simple calculations [4], (iii) automatic optimization techniques exist for programs being expressed as folds/unfolds [16, 24, 25, 17], and (iv) the structured use of recursion leads to a better programming style.

One of the first to propagate this style of programming was Backus [2], and it is also the essence of the Bird/Meertens formalism [3, 20]. However, despite the large interest in generalized fold operations, there is no general way to program with folds on abstract data types yet, and the goal of this paper is to fill this gap; it makes the following contributions:

1. Extend folds to ADTs

And as a consequence of this:

2. Broaden the applicability of fold
3. Extend the optimization possibilities for fold

And finally, by choosing a particular presentation:

4. Demystify fold and make it accessible to a wider audience

In the rest of this Introduction we will sketch how the idea of general folds can be generalized from algebraic data types to abstract data types, and we will also discuss related work. In Section 2 we introduce functors and some related definitions which prepares for Section 3 where we introduce ADTs as pairs of algebras. Section 4 then presents the definition of ADT folds and ADT transformers and gives examples that demonstrate the corresponding programming style. (A larger application can be found in Appendix B where we consider how to deal with graph ADTs and graph algorithms.) In Section 5 we present some laws that can be used for program verification and optimization. Conclusions follow in Section 6.

1.1 Generalized Folds for Algebraic Data Types

An algebraic data type T consists of a collection of constructors $c_1 : T_1 \rightarrow T, \dots, c_n : T_n \rightarrow T$. In the formalization of general fold operations, all these constructors are grouped together into one constructor $c : T_1 | \dots | T_n \rightarrow T$ whose argument type is then denoted by an expression $F(T)$, for a suitable type constructor F . Here, F is used to describe the argument type structure of T 's constructors, and $c : F(T) \rightarrow T$ is also called an F -algebra.

For example, a type for polymorphic lists can be defined in Haskell by:

```
data List a = Nil | Cons a (List a)
```

Actually, lists are predefined in Haskell, and since they are used so often, a special notation exists: `Nil` is written as `[]`, `Cons x l` is denoted by `x:l`, and the type `List a` is denoted by `[a]`. Thus, we have two constructors `[] :: [a]` and `(:) :: a -> [a] -> [a]`.¹ To combine them into one constructor we need a type constructor denoting the separated sum of both argument types, a nullary or unit type variant for `[]` and a variant with two types for `(:)`. For this purpose we can use the type constructor:

```
data Binary a b = UnitB | Two a b
```

By fixing the first type parameter of `Binary` we obtain a unary type constructor `Binary a` that, when applied to `[a]`, denotes the required argument type. Then the combined list constructor is defined as follows.

```
cList :: Binary a [a] -> [a]
cList UnitB      = []
cList (Two x l) = x:l
```

Now the generalized fold operation for F -algebras takes n

¹Brackets convert infix constructors like “:” into prefix operators.

parameters f_1, \dots, f_n (corresponding to each possible constructor), and, applied to a T -value v , first performs pattern matching on v , that is, determines the outermost applied constructor c_i and its arguments v_1, \dots, v_k . Then fold recursively folds all v_i of type T and “replaces” c_i by f_i .

1.2 Generalized Folds for Abstract Data Types

It is striking that fold operations have never been defined in a general form for abstract (non-algebraic) data types. The main reason for this is that a fold cannot map to less constrained structures because it would then not be defined uniquely. This means a severe limit of expressiveness, it prevents, for example, a function for counting elements of a set to be expressed as a fold.

A way out is to base the definition of fold on explicitly defined destructors instead of constructors: dually to a constructor, a destructor is a function $d : T \rightarrow G(T)$ where G is a type constructor describing, for example, the different possible result types T'_1, \dots, T'_m ; d is also called a G -coalgebra. An abstract data type (with carrier T) can then be defined simply as a pair (c, d) .

We can easily define canonical destructors for all algebraic data types that just undo the term construction. In that case we have $G = F$. For the list data type we obtain `dList` from `cList` by inverting argument and result type and by exchanging the left and right hand sides of the definitions.

```
dList :: List a -> Binary a (List a)
dList Nil      = UnitB
dList (Cons x l) = Two x l
```

In particular, this means that algebraic data types are completely covered by our ADT approach.

Concerning the non-algebraic case, consider an ADT for sets (with the same type structure as lists). The semantics of sets suggests to return one element from a set at most once, and this behavior can be realized within either the constructor or the destructor. Taking the second option we reuse `cList` as a constructor and define a destructor:

```
dSet :: Eq a => [a] -> Binary a [a]
dSet []      = UnitB
dSet (x:l) = Two x (filter (/=x) l)
```

This means to normalize sets within the destructor by removing the element that is split off the set from the remaining set.

Having destructors available, the fold operation (with parameters f_1, \dots, f_m) can now work as follows: first, d is applied to v yielding a value v_i of type T'_i . Then fold processes recursively all subvalues of type T (that is, fold “replaces” all T -values in v_i by their folded results) and applies f_i to the result. Note that in this case the ADT constructor c is not of any interest for fold; all that matters is the destructor d .

For example, counting the elements of a list can be expressed by a fold mapping `UnitB` to 0 and `Two x n` to $n+1$. Now by virtue of basing fold on destructors, the same works for sets, too: elements of a set are properly counted, since duplicates are eliminated within the set destructor.

As we will see, ADTs are frequently folded with functions that are constructors of other ADTs, so that ADT folding describes, in essence, the transformation of ADTs. This can be also viewed as a metamorphosis of ADTs, and we therefore call this programming style with ADTs and folds/transformers *metamorphic programming*.

1.3 Related Work

There is 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 [14] defines a data type for directed acyclic multi-graphs. With a careful choice of operations, which obey certain algebraic laws, the definition of graph folds becomes feasible. However, the whole approach is very limited, since it applies only to acyclic graphs without edge labels.

We have presented a more general view of graphs in [8]. Several powerful fold operations are defined, and theorems for program fusion allow the removal of intermediate search trees as well as intermediate graph structures. We can recover part of that approach within the current framework. This is demonstrated in Appendix B.

The only general approach for expressing folds over non-free data types we know of is the work of Fokkinga [13, 12]. 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. Although being a nice generalization completely in the categorical style, Fokkinga’s work still suffers from the already mentioned restrictions caused by the constraints that homomorphisms must map to quotients.

Essential in our approach are destructors, which are used in transformers as unfolds. The importance of unfolds has recently been stressed by Gibbons and Jones [15]. ADTs and metamorphic programming underline this and implicitly promote the use of unfolds.

2 Preliminaries: Functors and Bifunctors

We provide some basic machinery that will be needed in the sequel to define ADTs and ADT folds. We will show how to define type constructors as functors and bifunctors, and how these are used in a uniform representation for constructors and destructors.²

Recalling the above descriptions of fold, one point remains to be explained: how does fold find the “recursive occurrences of T -values”? The answer is that a type constructor, such as G , must have a certain structure – it must be a *functor*. In the scope of this paper this just means that G offers a function $map : T \rightarrow U \rightarrow (G(T) \rightarrow G(U))$ which is used to guide fold to the recursive occurrences. Functors describing argument and result types are also called *base functors* [4]. In Haskell there is a predefined class of functor type constructors:

```
class Functor f where
  map :: (t -> u) -> (f t -> f u)
```

We have already encountered the `Binary` type constructor. The justification that, for any type `a`, `Binary a` is indeed a functor is given by providing an implementation for the `map` function:

²We deliberately avoid here any reference to category theory. (A categorical treatment of some aspects of this paper is given in [9].) The reader is invited to download the Haskell source files from: <http://www.fernuni-hagen.de/inf/pi4/erwig/meta/>

```
instance Functor (Binary a) where
  map f UnitB      = UnitB
  map f (Two x y) = Two x (f y)
```

Viewed as a binary type constructor, `Binary` is an example of a *bifunctor*, whose relevant property for the ADT approach is to offer a function mapping “along” both type arguments. We can define a corresponding Haskell class:

```
class BiFunctor f where
  map2  :: (a -> b) -> (t -> u) -> f a t -> f b u
  mapFst :: (a -> b) -> f a t -> f b t
  mapFst f = map2 f id
```

together with the following instance:

```
instance BiFunctor Binary where
  map2 f g UnitB      = UnitB
  map2 f g (Two x y) = Two (f x) (g y)
```

Looking at the examples from the Introduction, we recognize a certain scheme that underlies the definition of constructors and destructors. To support economic programming we define, for each base functor, functions to map from the base functor to the carrier set of the defined data type and vice versa. For example, for the `Binary` type constructor we have:

```
fromB :: t -> (a -> b -> t) -> Binary a b -> t
fromB u f UnitB      = u
fromB u f (Two x y) = f x y

toB :: (t->Bool)->(t->a)->(t->b)->t->Binary a b
toB p f g x | p x    = UnitB
              | otherwise = Two (f x) (g x)
```

This allows to write constructors and destructors much more concisely, for example:

```
cList = fromB [] (:)
dList = toB null head tail
dSet  = toB null head rest
      where rest (x:xs) = filter (/=x) xs
```

We have shown here the (bi)functor definitions and `from/to` functions only for the `Binary` type constructor. A more comprehensive list of base functor definitions is given in Figure 4 in the Appendix C.

3 Abstract Data Types as Bialgebras

We have already indicated that an ADT can be regarded simply as a pair consisting of a constructor (or algebra) $c :: s \rightarrow t$ mapping from some argument type s to a carrier type t and a destructor (or coalgebra) $d :: t \rightarrow g\ t$ mapping from the carrier to a type expression given by the functor g applied to the carrier. Such an algebra/coalgebra pair mapping `to/from` a common type is called a *bialgebra* [13]. Since we do not need any information about the construction of ADT values, it is sufficient to represent the constructor argument type just by a type variable. In contrast, we do require the result type of the destructor to be expressed as a functor expression. Hence, we define the following type for ADTs together with functions for selecting the constructor and destructor:

```
data Functor g => ADT s g t = ADT (s->t) (t->g t)

con (ADT c _) = c
des (ADT _ d) = d
```

We can identify several interesting special cases of ADTs: we call ADTs that have equal argument and result types *symmetric*, and a symmetric ADT with base functor `Binary` is called *binary*. We will several times encounter collection ADTs with elements of type a and with a carrier defined by functor g that have a binary destructor, that is, splitting $(g\ a)$ -values into a and $g\ a$, but whose constructor inserts list of a -values into the carrier. Those ADTs are called *Join-ADTs*.

```
type SymADT g t = ADT (g t) g t
type BinADT a t = SymADT (Binary a) t
type JoinADT a g =
  ADT (Binary [a] (g a)) (Binary a) (g a)
```

Let us now consider some ADT examples. We have already seen that all algebraic data types can be canonically extended to ADTs. For example, for lists, we obtain:

```
list :: BinADT a [a]
list = ADT cList dList
```

The definition for `set` is analogous (use `dSet` instead of `dList`), and a queue ADT can be simply defined by:

```
queue :: BinADT a [a]
queue = ADT cList (toB null last init)
```

Let us also mention some ADT examples that use non-list carriers. For example, priority queues are more efficient when based on heaps instead of lists, and they can be realized by simply using appropriate heap operations. Similarly, an array ADT can be defined on the basis of a finite map implementation: arrays are constructed by an empty array and a function that adds an (index,value)-pair (i,x) to an array a , accumulating multiple index entries by a function f . The array is decomposed by successively splitting off the entry for the minimum index. Also based on finite maps, we can define a bag ADT in which bag elements are mapped to the number of their occurrences. The definitions of `pqueue`, `array` and `bag` are shown in Figure 5 in Appendix C.³ Note that `array` and `bag` are actually examples of parameterized ADTs. These are discussed briefly in Appendix A.

ADTs are not restricted to model collection types. Consider, for example, the `Unary` base functor, which is defined as follows:⁴

```
data Unary a = UnitU | One a

fromU :: t -> (a -> t) -> Unary a -> t
fromU u f UnitU      = u
fromU u f (One x)   = f x

toU :: (t -> Bool) -> (t -> a) -> (t -> Unary a)
toU p f x = if p x then UnitU else One (f x)
```

We can now define an ADT for natural numbers in an obvious way:

```
cNat = fromU 0 succ
dNat = toU (==0) pred

nat :: SymADT Unary Int
nat = ADT cNat dNat
```

³A heap and finite map implementation is included in the distribution of the Haskell source files.

⁴Haskell connoisseurs will recognize that `Unary` is essentially the same as `Maybe`. We felt that using `Maybe` would have been a bit confusing. In addition, the chosen names are in line with those of `Binary` and `Ternary`.

```

trans :: (Functor g, Functor h) => (g u -> r) -> ADT s g t -> ADT r h u -> (t -> u)
trans f a b = con b . f . map (trans f a b) . des a

```

Figure 1: Definition of ADT Transformer.

A boolean ADT can be defined by:

```

bool :: BinADT Bool Bool
bool = ADT (fromB False (|))
          (toB' not (\_ -> (True, False)))

```

An important aspect of ADTs is that the clear separation into constructor and destructor gives them a highly modular structure which, in particular, makes it very easy to combine different views of ADTs on the constructor/destructor side. We can, for example, provide a join view of lists (on the constructor side), or we can define versions of natural number ADTs that construct numbers by multiplication or that return the number currently decomposed in addition to its predecessor:

```

jList :: JoinADT a []
jList = ADT (fromB [] (++) ) dList

prod  :: ADT (Binary Int Int) Unary Int
prod  = ADT (fromB 1 (*) ) dNat

rng   :: ADT (Unary Int) (Binary Int) Int
rng   = ADT cNat (toB (==0) id pred)

```

Finally, examples for ADTs with a more complex (than binary) type structure are all kinds of trees. A definition of a binary tree ADT is included in Figure 5.

It seems that ADTs are not as abstract as they should be because the carrier type is always visible. At least this is true when using transparent types, such as built-in lists, as carriers. However, this can be avoided by using abstract types that do not export their internal structure (cf. the `pqueue` ADT). (NB: The notion of ADT in this paper is a specialized one not aiming at hiding representations, rather the focus is on giving predefined shapes that enable the definition of fold operations.)

4 Programming with ADTs

4.1 Fold

It is now quite simple to define a general fold operation for ADTs because these carry their own destructors. Since a destructor is a function of type $t \rightarrow g\ t$, the parameter functions for fold must also be grouped into one function f of type $g\ u \rightarrow u$ for some type u . This can be conveniently achieved by using the “from” function defined for the base functor g . Then the definition of `fold` works as described above: after applying the destructor, `fold` is recursively applied by means of `map`, followed by an application of f :

```

fold :: Functor g => (g u -> u) -> ADT s g t -> t -> u
fold f a = f . map (fold f a) . des a

```

Note that `fold` is essentially the same as a hylomorphism [21, 25]: the destructor of a is used like an anamorphism that builds a value of structure g , and f serves as the argument of a catamorphism that consumes that value. This (and several more) relationships are explained in [10].

As an example we define a function for computing the sum of a list of integers by a fold on the ADT `list`. Since the result type of `dList` is defined by `Binary`, the parameter function for fold can be conveniently denoted with the help of `fromB`:

```

sum :: Num a => [a] -> a
sum = fold (fromB 0 (+)) list

```

This definition looks very similar to the familiar definition of `sum` by `list fold`. The important difference is, however, that essentially the same definition instantly works for sets, too:

```

sumset :: Num a => [a] -> a
sumset = fold (fromB 0 (+)) set

```

Whereas `sum` adds all numbers of a list irrespective of duplicates, `sumset` adds each number only once.

We should say one word of warning here: since general recursion can be used in the definition of fold parameters and ADT con/destructors, the termination of folds and transformers can, in general, *not* be guaranteed.

4.2 Transform

If we consider the related task of multiplying the numbers of a list, we recognize that the required `fold` parameter is already available as the constructor of the ADT `prod`. This leads to the definition of a particular kind of fold operations that use ADT constructors as parameters. Essentially, such a fold describes the transformation of one ADT (called *source ADT*) into another one (called *target ADT*), and we therefore call these folds *ADT transformers*. In this simple form we can use only an ADT as a target whose constructor’s argument type is defined by the same base functor that is used in the result type of the source ADT destructor. This restriction can be easily lifted by adding a parameter function f that maps from the result type structure of the source ADT destructor to the argument type of the target ADT constructor; we call f the *map* of the transformer.⁵

The definition of the ADT transformer is given in Figure 1. We also define as a special case:

```

transit = trans id

```

We have the following obvious relationships between `trans` and `fold`:

```

trans f a b = fold (con b.f) a
transit a b = fold (con b) a

```

(TransFold)

This shows how to express a fold (such as `sum`) by an ADT transformer: simply select a target ADT that has the fold parameter function as its constructor.

With regard to our motivating example, multiplication of a list of numbers $1..n$ realizes the factorial function:

```

fac = transit rng prod

```

Further examples for the use of transformers are (`halves =`

⁵ Actually, f is a natural transformation between the two base functors.

```

ADT cNat (toU (==0) ('div' 2)):
  log2 :: Int -> Int
  log2 = pred . transit halves nat

  any :: (a -> Bool) -> [a] -> Bool
  any p = trans (mapFst p) list bool

  histogram :: Ord a => [a] -> M.FiniteMap a Int
  histogram = trans once list (array (+))
    where once = mapFst (\n->(n,1))

```

ADT transformers are actually hylomorphisms using the source ADT destructor as the unfold argument and the target ADT constructor as the fold argument. We believe that thinking in ADT abstractions is easier than working with hylomorphisms directly and might eventually enjoy a better user acceptance; in a sense, ADTs allow programming with hylomorphisms without being aware of it.

4.3 Calculating Target ADTs

The length of a list can be computed by a transformer from `list` to `nat`. Since the functors do not agree, we must now provide a function to adjust the destructured values to the constructor of `nat`.

```

length = trans p2 list nat
  where p2 UnitB      = UnitU
        p2 (Two _ y) = One y

```

The map of the transformer is sometimes a bit unwieldy and makes function definitions difficult to read. We can always remove it by selecting an appropriate target ADT that agrees with the functor of the source ADT destructor.⁶ In the above example, the desired ADT can be calculated by deriving a solution for `count` in the following equation:

```
trans p2 list nat = transit list count
```

If we unroll the definition of `trans once`, we obtain:

```
length = cNat . p2 . map length . dList
```

Next we can fuse the functions `cNat` and `p2`. Expanding `fromU` in the definition of `cNat` gives:

```

cNat UnitU      = 0
cNat (One x)    = succ x

```

This can be immediately fused with the two cases of `p2` resulting in the following function definition.

```

cCount UnitB      = 0
cCount (Two _ y)  = succ y

```

(Note that `cCount = fromB 0 (_ y->succ y)`.) We thus have:

```

length = cNat . p2 . map length . dList
       = cCount . map length . dList
       = transit list count

```

where `count = ADT cCount dNat`. We can use `count` also to determine the size of a set:

```
card = transit set count
```

It is striking that the definitions of `length` and `card` differ only in their argument ADT. We can easily give a unifying

⁶We can also use predefined, parameterized natural transformations between functors. For example, using `ntBU` (see Figure 4) `length` can be defined as `trans (ntBU (_ y->y)) list nat`.

definition that takes as an additional parameter the ADT to be aggregated:

```

size :: ADT s (Binary a) t -> t -> Int
size a = transit a count

```

This definition of `size` has, in a sense, a higher degree of polymorphism than, say, the function `length`. It is, however, fixed with respect to the type structure of the ADT, so in this respect, it is not as general as polytypic functions [18]. These two kinds of polymorphisms are somewhat orthogonal to each other.

4.4 Non-Binary Recursion

Until now we have programmed only with ADTs having the base functors `Binary` and `Unary`. It is clear that all presented concepts also apply to, say, tree-like structures. For example, we can easily specify a function for a preorder traversal of binary trees.

```

preorder :: Tree a -> [a]
preorder = trans (ntTB id (++) ) tree list

```

Transformers also offer a nice view on divide-and-conquer algorithms. As an example we will define quicksort. First, we provide a “split”-view on lists by defining an ADT that decomposes a non-empty list with head `x` and tail `l` into three lists of all elements that are less, equal, or greater than `x`.

```

fork :: Ord a => ADT (Binary a [a]) (Ternary [a]) [a]
fork = ADT cList (toT null (sel (==)) (sel (<)) (sel (>)))
  where sel f l@(x:_) = filter (flip f x) l

```

All the target ADT has to do now is to simply join the triples of lists delivered by `fork`. We therefore define the ADT `combine` as follows.

```

combine :: ADT (Ternary [a] [a]) (Binary a) [a]
combine = ADT (fromT [] append213) dList
  where append213 y x z = x++y++z

```

Now we can define quicksort by the following transformer.

```

quicksort :: Ord a => [a] -> [a]
quicksort = transit fork combine

```

Maybe the popularity and success of the divide-and-conquer scheme is in part due to its structured use of recursion.

4.5 ADT Streams

The functions `trans` and `transit` allow to map from one ADT directly into another one. By composing two transformers we obtain a function that maneuvers values “via” an intermediate ADT:

```

via :: (Functor g, Functor h, Functor i) =>
      ADT s g t -> ADT (g u) h u -> ADT (h v) i v -> t -> v
via a b c = transit b c . transit a b

```

By composing more than two transformers we can set up a stream of ADTs.

```

stream :: Functor g => [SymADT g t] -> t -> t
stream [a,b]      = transit a b
stream (a:b:as)  = stream (b:as) . transit a b

```

For simplicity we have given a definition for symmetric ADTs; this can be generalized by equipping the ADTs in the list with natural transformations. More serious, however, is the restriction that all ADTs of a stream must have

the same carrier. This results from the restriction that lists in Haskell are homogeneous (which might be lifted in future by the introduction of existential types). In any case, we can circumvent this by using streaming operators for a fixed number of ADTs, such as `via`.

ADT streams are handy for expressing certain algorithms, for example, removing duplicates from a list, reversing a list, or sorting a list:

```
remdup    = via list set list
reverse   = via list queue list
heapsort  = via list pqueue list
bucketsort = via list bag list
```

We also could have used `stream` in the definition of `remdup` and `reverse`, since `set` and `queue` are implemented on the basis of lists. However, specialized stream operators have more general types and are thus less sensitive to re-implementations of ADTs that change their carriers.

5 Laws for Program Manipulation

It has been stressed by Meijer and Hutton [22] that a good advice for obtaining useful relationships is to look at the free theorem [26] for the type of polymorphic functions. Now the free theorem for `fold` is:

Theorem 1 (Free Fold)

$$1 \text{ strict} \wedge 1.f = f'.\text{map } 1 \wedge d'.r = \text{map } r.d \implies 1.\text{fold } f \text{ (ADT } c \text{ } d) = \text{fold } f' \text{ (ADT } c \text{ } d').r \quad \square$$

We obtain as a special case a traditional “fold fusion from left” theorem if we let $r = \text{id}$ and $d = d'$ (then the last premise becomes trivially true). To avoid cluttering theorems with strictness requirements we are restricting to strict functions in the sequel.

Corollary 1 (Fold Left)

$$1.f = f'.\text{map } 1 \implies 1.\text{fold } f \text{ } a = \text{fold } f' \text{ } a \quad \square$$

Here we can give only a taste of the calculational chances offered by the metamorphic programming style. More laws are investigated in [10].

5.1 Fusion

We have already remarked that ADTs are a conservative extension of algebraic data types which means that the laws and program transformation techniques that have already been developed can still be used in the extended framework, although some rules have to be spelled out a bit differently. For example, as an instance of Corollary 1 with $1 = \text{fold } f' \text{ } b$ we obtain a fusion rule for two functions that are both expressed as folds:

Theorem 2 (Fold Fusion)

$$\text{des } b.f = \text{id} \implies \text{fold } f' \text{ } b.\text{fold } f \text{ } a = \text{fold } f' \text{ } a$$

Proof. First we show:

```
fold f' b.f
= f'.map (fold f' b).des b.f      { FoldDef }
= f'.map (fold f' b).id          { precondition }
= f'.map (fold f' b)
```

Now we can apply Corollary 1 to the left side and immediately obtain the right side. \square

Next we record an interesting property of ADTs:

Definition 1 ADT $c \text{ } d$ is *invertible* : $\iff d.c = \text{id}$

In particular, all ADTs that are canonically derived from algebraic data types (such as `list` and `tree`) are invertible.

An important property of invertible ADTs is that they can be safely omitted from ADT streams.

Theorem 3 (Transit Law)

$$b \text{ is invertible} \implies \text{via } a \text{ } b \text{ } c = \text{transit } a \text{ } c$$

Proof.

```
via a b c
= transit b c.transit a b      { ViaDef }
= fold (con c) b.fold (con b) a { TransFold }
= fold (con c) a              { FoldFusion }
= transit a c                  { TransFold } \square
```

By induction, this result also holds for streams of more than 3 ADTs. This fusion rule for ADTs is a natural generalization of the fusion law for algebraic data types, and its importance lies in the fact that all the well-known fusion optimization techniques for algebraic data types can be carried over to the metamorphic programming style.

5.2 Proving Program Properties

We present a simple law that describes how to move predicates along ADT transformers and demonstrate its use in proving the correctness of the heapsort program.

As a corollary of Theorem 2 we know that a property p which holds for values of an ADT a also holds for their transformed counterparts of an invertible ADT b .

Theorem 4 (Property Transition)

$$(b \text{ is invertible} \wedge p \text{ } a = f.\text{fold } g \text{ } a) \implies p \text{ } a = p \text{ } b.\text{transit } a \text{ } b$$

Proof.

```
p b.transit a b
= f.fold g b.fold (con b) a      { Def. p & TransFold }
= f.fold g a                    { FoldFusion }
= p a                            { Def. p } \square
```

The correctness of `heapsort` can be proved in two steps: first, show that the ADT stream neither generates nor forgets elements. This follows from the very same property of the used ADTs: for invertible ADTs, such as `list`, this is obvious, and for the ADT `pqueue`, the property is assumed (or better, has to be proved) for the underlying `Heap` implementation. Second, show that the resulting list is sorted. This can be done using the above theorem. We first define the following “minimum” property:

```
pMin :: Ord a => ADT s (Binary a) t -> t -> Bool
pMin a = fst . fold f a where
  f UnitB      = (True,Nothing)
  f (Two x (b,Nothing)) = (True,Just x)
  f (Two x (b,Just y))  = (x<y && b,Just (min x y))
```

The function f checks whether x is less than or equal the current minimum and updates the current minimum value. Folding an ADT with f yields a pair whose first component is true when the outermost value is not greater than any other value in the ADT.

Since the minimum property is a characterization of priority queues, for all priority queues p the following expression is always true:

pMin pqueue p

Next we have to recognize that `pMin list` describes the property of lists being sorted. This can be seen by induction as follows: an empty or a one-element list is always sorted, and `f` obviously computes `True` and the minimum of the list in both cases. For a list `x:l` (with `l ≠ empty`) we assume by induction that `pMin list` properly determines whether `l` is sorted (in variable `b`) and its minimum `Just y`. The list `x:l` is then marked as sorted only if `b = True`, that is, if `l` is sorted, and if `x ≤ y`. Since this is a correct characterization of the sorted predicate, we can thus say

```
sorted = pMin list
```

Now we can reason as follows. Since `pMin pqueue` is true for all priority queues, it is, in particular, true for any priority queue that is built by a transformation from a list, that is, for all lists `l` the following expression is always true.

```
pMin pqueue (transit list pqueue l)
```

Now we can apply Theorem 4 with `a = pqueue` and `b = list`, and we obtain:

```
pMin list (transit pqueue list (transit list pqueue l))
```

Next we fold the definition of `via` and get:

```
pMin list (via list pqueue list l)
```

Finally, we fold the definitions of `heapsort` and `sorted`, and we know that the following is true for all lists `l`:

```
sorted (heapsort l)
```

5.3 Exploiting Single-Threadedness of ADTs

We can observe that intermediate stream ADTs are used in a single-threaded way, and this means that they can be implemented with imperative data structures without introducing unwanted side-effects. For example, we can use an imperative queue implementation for the intermediate stream ADT in `reverse`, or we can use an imperative array in the stream for `bucket.sort`.

A closer look reveals that – corresponding to the position of an ADT in a transformer or in a stream – there are essentially three different cases that offer different optimization opportunities:

(1) As just described, intermediate stream ADTs can be implemented in a completely imperative way.

(2) Target ADTs of transformers can be, at least internally, constructed imperatively, since all intermediate versions are single-threaded, but the final value must be a functional (or, persistent) data structure [23, 6], since it can be freely shared. This fact can be exploited as follows: first, we can use an imperative data structure during the construction phase and copy it finally into, or use it as an initial value of, a functional data structure that is returned as the result. In some cases the copy can be saved by using specialized constructors that build a representation of a functional data structure without keeping intermediate versions.

A typical example is the version tree implementation of functional arrays [1]: the initial values are kept in an imperative array, and updates to it are recorded in a tree. Now when an array is used as the target of a transformer, we need not start with an empty array and record all entries in the version tree. Instead we can construct the array value imperatively. This means that, for example, the function

`histogram` runs in binary time if the array ADT is implemented in the described way.

(3) A source ADT `a` of a transformer might be shared, but all intermediate versions constructed from `a` are single-threaded. This shows two possible ways of optimization: first, we can work with a copy of `a`. This can then be a plain imperative data structure, which can be freely updated during the decomposition. Instead of copying we can use “quasi”-destructors that actually do not alter the ADT, but rather keep an auxiliary data structure that helps to simulate the intermediate ADT values during the decomposition.

For example, the graph algorithms presented in Appendix B run very efficiently when we take a graph ADT with an array of adjacency lists representation using a destructor that instead of removing nodes from the graph simply remembers removed nodes in an imperative array. Then contexts that are delivered during decomposition have to be cleared from already deleted nodes. In this way, we obtain graph algorithms that are as efficient as their imperative counterparts [8, 7]. The really nice thing is that we achieve all this without the need for using monads (on the algorithmic level); the details of the efficient implementation are completely hidden in the ADT implementations.

These aspects of introducing imperative implementations are discussed in much more detail in [10] where the informal reasoning given here is underpinned by several theorems.

6 Conclusions

We have shown how to pave the way for structured recursive programming with abstract data types. By introducing ADTs as (constructor, destructor)-pairs, ADT folds and transformers can be easily defined using ADT destructors. The proposed programming style can be used without knowing anything about category theory which supports its acceptance.

Requiring the explicit definition of destructors means additional work compared with the approach for algebraic data types. However, we believe the effort pays off, since it offers much freedom in the design of ADTs, in particular, the separation into constructor and destructor provides a high degree of modularity. Moreover, ADT transformers are much more general than folds on algebraic data types, since they can map into types with less structure.

Still, the extended framework offers all the optimization opportunities that have proved useful for algebraic data types, since invertible ADTs (which cover the class of algebraic data types) can be safely fused away from the interior of ADT streams. Moreover, the single-threadedness of ADTs facilitates the introduction of highly efficient imperative ADT implementations without affecting referential transparency.

References

- [1] A. Aasa, S. Holström, and C. Nilsson. An Efficiency Comparison of Some Representations of Purely Functional Arrays. *BIT*, 28:490–503, 1988.
- [2] 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.

- [3] R. S. Bird. Lectures on Constructive Functional Programming. In M. Broy, editor, *Constructive Methods in Computer Science*, NATO ASI Series, Vol. 55, pp. 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. *2nd Fuji Int. Workshop on Functional and Logic Programming*, pp. 71–90, 1996.
- [6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.
- [7] M. Erwig. Fully Persistent Graphs – Which One to Choose? *9th Int. Workshop on Implementation of Functional Languages*, LNCS 1467, pp. 123–140, 1997.
- [8] M. Erwig. Functional Programming with Graphs. *2nd ACM Int. Conf. on Functional Programming*, pp. 52–65, 1997.
- [9] M. Erwig. Categorical Programming with Abstract Data Types. *7th Int. Conf. on Algebraic Methodology and Software Technology*, LNCS 1548, pp. 406–421, 1998.
- [10] M. Erwig. The Categorical Imperative – Or: How to Hide Your State Monads. *10th Int. Workshop on Implementation of Functional Languages*, pp. 1–25, 1998.
- [11] L. Fegaras and T. Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions. *23rd ACM Symp. on Principles of Programming Languages*, pp. 284–294, 1996.
- [12] M. M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, 1992.
- [13] M. M. Fokkinga. Datatype Laws without Signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.
- [14] J. Gibbons. An Initial Algebra Approach to Directed Acyclic Graphs. *Mathematics of Program Construction*, LNCS 947, pp. 282–303, 1995.
- [15] J. Gibbons and G. Jones. The Under-Appreciated Unfold. *3rd ACM Int. Conf. on Functional Programming*, pp. 273–279, 1998.
- [16] A. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. *Conf. on Functional Programming and Computer Architecture*, pp. 223–232, 1993.
- [17] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling Calculation Eliminates Multiple Data Traversals. *2nd ACM Int. Conf. on Functional Programming*, pp. 164–175, 1997.
- [18] P. Jansson and J. Jeuring. PolyP – A Polytypic Programming Language Extension. *24th ACM Symp. on Principles of Programming Languages*, pp. 470–482, 1997.
- [19] G. Malcolm. Homomorphisms and Promotability. *Mathematics of Program Construction*, LNCS 375, pp. 335–347, 1989.
- [20] L. Meertens. Algorithmics – Towards Programming as a Mathematical Activity. *CWI Symp. on Mathematics and Computer Science*, pp. 289–334, 1986.
- [21] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. *Conf. on Functional Programming and Computer Architecture*, pp. 124–144, 1991.
- [22] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. *Conf. on Functional Programming and Computer Architecture*, pp. 324–333, 1995.
- [23] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.
- [24] T. Sheard and L. Fegaras. A Fold for all Seasons. *Conf. on Functional Programming and Computer Architecture*, pp. 233–242, 1993.
- [25] A. Takano and E. Meijer. Shortcut Deforestation in Calculational Form. *Conf. on Functional Programming and Computer Architecture*, pp. 306–313, 1995.
- [26] P. Wadler. Theorems for Free! *Conf. on Functional Programming and Computer Architecture*, pp. 347–359, 1989.

Appendix A. Parameterized ADTs

When programming with transformers one is frequently faced with the need to define several variants of an ADT just for having a particular destructor (or constructor) behavior available. In some cases it is therefore convenient to define a generic ADT from which the desired instances can be obtained by simply providing appropriate function parameters. This also helps to compare different ADT instances and the functions defined by transformers on them.

Consider, for example, a binary ADT for integer pairs whose destructor is parameterized by the termination predicate p and by a function f that transforms integer pairs.

```
type Int2 = (Int,Int)
```

```
nat2 :: (Int2->Bool)->(Int2->Int2)->ADT () (Binary Int) Int2
nat2 p f = ADT (\_->(0,0)) (toB p fst f)
```

With `nat2` we can express many numeric operations (function product “><” and tupling “/\”) are defined together with other auxiliary functions in Figure 3 in Appendix C.)

```
minus = uncurry (-)
eq0   = (==0).snd
eq0'  = (==0).fst
lt0'  = (<0).fst

mult  = transit (nat2 eq0 (id >< pred)) summ
power = transit (nat2 eq0 (id >< pred)) prod
fac n = transit (nat2 eq0 (pred >< pred)) prod (n,n)
mod   = transit (nat2 lt0' (minus /\ snd)) final
gcd   = transit (nat2 eq0' (mod' /\ fst)) final
      where mod' = fromJust . mod . swap
```

The ADT `final` keeps the last of a sequence of values delivered by the decomposition of a binary ADT, see Figure 5 in Appendix C.

Appendix B. An Advanced Example: Graph ADTs

To define a graph ADT in the technical sense of this paper, we need an inductive definition of graphs. We have introduced and motivated such a compositional view in [8]: a graph is either empty, or it is constructed by adding a node

together with edges to its predecessors and successors. Let `Node` be a type of node values (which is here just a synonym for `Int`), and let `Graph a b` be the type of graphs with node (edge) labels of type `a` (`b`). A node context is a labeled node together with a list of successors and a list of predecessors (paired with the corresponding edge labels):

```
type Adj b      = [(b,Node)]
type Context a b = (Adj b,Node,a,Adj b)
```

Then we have the following two graph constructors:

```
empty :: Graph a b
embed :: Context a b -> Graph a b -> Graph a b
```

Note that `embed` yields a runtime error if either the node to be inserted is already present in the graph or if any of the predecessor or successor nodes does not exist in the graph. Both constructors can be combined as follows.

```
type LinGraph a b = Binary (Context a b) (Graph a b)
```

```
cGraph :: LinGraph a b -> Graph a b
cGraph = fromB empty embed
```

Unordered Graph Decomposition

A graph can be decomposed by removing a node together with its context, that is, its incident edges. In this section we will use a destructor `matchAny` that selects and removes an arbitrary node (for example, the smallest one). Below we shall encounter a more versatile destructor.

```
matchAny :: Graph a b -> (Context a b, Graph a b)
```

Now we can easily define an ADT for graphs (`isEmpty` checks whether a graph contains any nodes).

```
graph :: BinADT (Context a b) (Graph a b)
graph = ADT cGraph (toB' isEmpty matchAny)
```

To take an example for a transformer into graphs, consider the following function that builds graphs from a list of contexts:

```
buildGraph :: [Context a b] -> Graph a b
buildGraph = transit list graph
```

With `buildGraph` we can construct, for instance, a cycle of three nodes (of type `Graph Char String`):

```
buildGraph [([("ca",3]),1,'a',[("ab",2)]),
            ([],2,'b',[("bc",3)]),
            ([],3,'c',[])]
```

With transformers out of graphs we can compute, for example, a list of a graph's nodes, the number of its edges, or node membership (`q2 (_,x,_) = x`, see Figure 3):

```
nodes :: Graph a b -> [Node]
nodes = trans (mapFst q2) graph list

noEdges :: Graph a b -> Int
noEdges = trans (mapFst size) graph summ
  where size (p,_,_,s) = length p+length s

member :: Node -> Graph a b -> Bool
member v = trans (mapFst ((v==).q2)) graph bool
```

The function `gmap` is a graph “endo”-transformer that allows to map functions to graph contexts.

```
gmap :: (Context a b -> Context c d)
      -> Graph a b -> Graph c d
gmap f = trans (mapFst f) graph graph
```

With `gmap` we can, for example, map functions to all node labels, and we can even define graph reversal:

```
mapNodes :: (a -> a') -> Graph a b -> Graph a' b
mapNodes f = gmap (lab f)
  where lab f (p,v,l,s) = (p,v,f l,s)
```

```
grev :: Graph a b -> Graph a b
grev = gmap swap
  where swap (p,v,l,s) = (s,v,l,p)
```

The expressiveness of graph transformers that do not have control over the order of node decompositions is a bit limited. We will consider a more powerful graph ADT next.

A Graph Library in 6 Lines

The keys to more sophisticated graph algorithms are (i) to use a graph ADT that is based on a destructor for matching particular nodes, and (ii) to enable the matching of nodes in a specific order. The first requirement is met by a function `match` that takes an additional `Node`-parameter and tries to decompose the context of this node from the graph. In contrast to `matchAny`, this operation can fail on non-empty graphs if the node is not contained in the graph. Therefore, `match` is defined to return a “Maybe” context-value.⁷

```
type MContext a b = Maybe (Context a b)
type Decomp a b   = (MContext a b, Graph a b)
```

```
match :: Node -> Graph a b -> Decomp a b
```

We are faced, however, with one problem: the decomposition of the graph at one particular node yields, as part of a context value, *several* new nodes. Now which one of these nodes should be used in the next decomposition, and what should be done with the remaining nodes? An answer to both of these questions is to use another ADT as a buffer which is constructed and destructed alongside with the graph transformation.⁸ nodes yielded by a graph decomposition are inserted into the buffer, and the node to be used in the next `match` decomposition is destructed from the buffer. The use of buffers also provides a solution to the second requirement, since the buffer ADT implicitly defines a specific ordering among the nodes.

Hence, we shall define a graph ADT that is parameterized, in the first place, by the buffer ADT to be used. We then need two further “interfacing” functions `f` and `h`, which specify how to obtain the next node from the decomposed buffer ADT value, respectively, what part of the currently decomposed context to insert into the buffer. Function `f` might simply be the identity if only nodes are stored in the buffer, but it might also be a more complex function when the buffer carries additional information. Likewise, `h` might simply extract the predecessors or successors from the context, but it might also combine them with other values that could be needed to control the buffer behavior. In any case,

⁷With this definition `match` can return a graph value even if matching itself fails. This is important for the definition of graph transformers below.

⁸An alternative proposal is to use an initially fixed sequence of nodes. However, it seems that this does not have many applications, since the order of visiting nodes is typically determined dynamically during the exploration of the graph.

```

bufGraph :: (JoinADT c f) -> (c -> Node) -> (c -> Context a b -> [c]) ->
  ADT () (Binary (MContext a b)) (f c, Graph a b)
bufGraph (ADT c d) f h = ADT (\_ -> (c UnitB, empty)) explore
  where explore (b,g) = case d b of UnitB          -> UnitB
                                Two x b' | isEmpty g -> UnitB
                                          | otherwise -> Two ctx (c (Two s b'), g')
                                          where (ctx, g') = match (f x) g
                                                  s         = maybe [] (h x) ctx

```

Figure 2: Parametric Graph ADT.

`h` delivers a list of values, and so the buffer should be a Join-ADT.

The definition of the parameterized graph ADT is given in Figure 2: the function `bufGraph` constructs an ADT whose carrier consists of (buffer, graph)-pairs. Since `bufGraph` is used for decomposing graphs, we give only a “dummy” constructor – the interesting part is the destructor `explore` which works as follows. First, the buffer `b` is decomposed, and if it cannot provide any value, the whole decomposition stops, which also happens when the graph `g` is empty. Otherwise, we compute with `f` from the decomposed buffer value `x` the node to be matched next and use this node to decompose the graph. This yields a remaining graph `g'` and a context-value `ctx`, which forms the first part of the result of `explore`. The second part is the new carrier-value, that is, a pair consisting of a buffer and a graph, which is simply `g'`. The new buffer-value is the remaining buffer `b'` into which a list of values `s` (essentially obtained from the current context) is inserted: if the last graph decomposition failed, that is, if `ctx = Nothing`, then `s` is `[]`. Otherwise, `s` is given by `h x ctx`. (Giving `h` access to the value `x` allows information from the buffer to be carried over to its next version, an example is given below.)

Before we can use `bufGraph` to specify graph algorithms we have to define appropriate target and buffer ADTs. If we are interested in obtaining a list of visited nodes, we could try to simply use a list ADT, but we have to care about the fact that contexts are `Maybe`-values. Therefore, we employ the following ADT combinator that makes an ADT “maybeable”:

```

maybeView :: Functor g => ADT (Binary a t) g t ->
  ADT (Binary (Maybe a) t) g t
maybeView (ADT c d) = ADT c' d
  where c' UnitB          = c UnitB
        c' (Two Nothing y) = y
        c' (Two (Just x) y) = c (Two x y)

```

Now we can use the ADT `mList = maybeView list` as the target for graph transformers. As buffers we can use ADTs we have already defined earlier, such a queue, but we have to equip them with a join view on the constructor side. Again, we define an ADT combinator:

```

joinView :: Functor g => ADT (Binary a t) g t ->
  ADT (Binary [a] t) g t
joinView (ADT c d) = ADT c' d
  where c' UnitB          = c UnitB
        c' (Two xs y) = foldr c'' y xs
          where c'' x y = c (Two x y)

```

Hence, we can derive the following buffer ADTs:

```

jStack = joinView list
jQueue = joinView queue
jPQueue = joinView pqueue

```

Next we can define a function representing a class of graph traversals (the definition is just for saving space):

```
expl buf = trans (mapFst (map q2)) buf mList
```

Now we can specify graph algorithms. We start by depth-first and breadth-first search:

```
sucs _ (_,_,_,s) = map snd s
```

```
dfs v g = expl (bufGraph jStack id sucs) ([v],g)
bfs v g = expl (bufGraph jQueue id sucs) ([v],g)

```

With the above definition, `dfs` is defined to start at one specific node, which means that it cannot, in general, reach the whole graph. This can be easily generalized by using a list of all nodes as a start value for the buffer.

We can even define Dijkstra’s shortest path algorithm as an instance of `bufGraph` (note that node costs must come first in the priority queue):

```
labSucs (y,_) (_,_,_,s) = [(y+1,v) | (1,v) <- s]
```

```
sp v g = expl (bufGraph jPQueue snd labSucs)
  (H.unit (0,v),g)

```

This example demonstrates the need for the parameter `f` and for the access of decomposed buffer values by `h`.

Note that indexed decomposition as offered by `bufGraph` can also be reasonably defined and used for arrays.

Appendix C. ADT Reference

```

infixr 8 /\
infixr 8 ><
(f /\ g) x      = (f x, g x)
(f >< g) (x,y)  = (f x, g y)

swap (x,y) = (y,x)
q2 (_,x,_,_) = x

f ‘o‘ g = curry (f . (uncurry g))

fromJust (Just x) = x
fromMaybe _ (Just x) = x
fromMaybe x Nothing = x

```

Figure 3: Auxiliary Functions.

```

data I a      = I a
data Unary a  = UnitU | One a
data Binary a b = UnitB | Two a b
data Ternary a b = UnitT | Three a b b

instance Functor I where
  map f (I x) = I (f x)

instance Functor Unary where
  map f UnitU = UnitU
  map f (One x) = One (f x)

instance Functor (Binary a) where
  map f UnitB = UnitB
  map f (Two x y) = Two x (f y)

instance Functor (Ternary a) where
  map f UnitT = UnitT
  map f (Three x y z) = Three x (f y) (f z)

instance BiFunctor Binary where
  map2 f g UnitB = UnitB
  map2 f g (Two x y) = Two (f x) (g y)

instance BiFunctor Ternary where
  map2 f g UnitT = UnitT
  map2 f g (Three x y z) = Three (f x) (g y) (g y)

fromI :: (a -> t) -> I a -> t
fromI f (I x) = f x

toI :: (t -> a) -> t -> I a
toI f x = I (f x)

fromU :: t -> (a -> t) -> Unary a -> t
fromU u f UnitU = u
fromU u f (One x) = f x

toU :: (t -> Bool) -> (t -> a) -> (t -> Unary a)
toU p f x = if p x then UnitU else One (f x)

fromB :: t -> (a -> b -> t) -> Binary a b -> t
fromB u f UnitB = u
fromB u f (Two x y) = f x y

toB :: (t -> Bool) -> (t -> a) -> (t -> b) -> t -> Binary a b
toB p f g x = if p x then UnitB else Two (f x) (g x)

toB' :: (t -> Bool) -> (t -> (a,b)) -> t -> Binary a b
toB' p f x = if p x then UnitB else Two y z where (y,z) = f x

fromT :: t -> (a -> b -> b -> t) -> Ternary a b -> t
fromT u f UnitT = u
fromT u f (Three x y z) = f x y z

toT :: (t -> Bool) -> (t -> a) -> (t -> b) -> (t -> b) -> t -> Ternary a b
toT p f g h x = if p x then UnitT else Three (f x) (g x) (h x)

ntBU :: (a -> b -> c) -> Binary a b -> Unary c
ntBU f UnitB = UnitU
ntBU f (Two x y) = One (f x y)

ntTB :: (a -> c) -> (b -> b -> d) -> Ternary a b -> Binary c d
ntTB f g UnitT = UnitB
ntTB f g (Three x y z) = Two (f x) (g y z)

```

Figure 4: Definitions for Base Functors.

```

nat    :: SymADT Unary Int
nat    = ADT cNat dNat

count  :: ADT (Binary a Int) Unary Int
count  = ADT (fromB 0 (\x y->succ y)) dNat

summ   :: ADT (Binary Int Int) Unary Int
summ   = ADT (fromB 0 (+)) dNat

prod   :: ADT (Binary Int Int) Unary Int
prod   = ADT (fromB 1 (*)) dNat

rng    :: ADT (Unary Int) (Binary Int) Int
rng    = ADT cNat (toB (==0) id pred)

halves :: SymADT Unary Int
halves = ADT cNat (toU (==0) ('div' 2))

nat2   :: ((Int,Int) -> Bool) -> ((Int,Int) -> (Int,Int)) -> ADT () (Binary Int) (Int,Int)
nat2 p f = ADT (\_>(0,0)) (toB p fst f)

bool   :: BinADT Bool Bool
bool   = ADT (fromB False (||)) (toB' not (\_>(True,False)))

final  :: ADT (Binary a (Maybe a)) I (Maybe a)
final  = ADT (fromB Nothing (Just 'o' fromMaybe)) (toI id)

list   :: BinADT a [a]
list   = ADT cList dList

set    :: Eq a => BinADT a [a]
set    = ADT cList (toB null head rest) where rest (x:xs) = filter (/=x) xs

queue  :: BinADT a [a]
queue  = ADT cList (toB null last init)

pqueue :: Ord a => BinADT a (H.Heap a)
pqueue = ADT (fromB H.empty H.insert) (toB H.isEmpty H.findMin H.delMin)

array  :: Ord i => (a -> a -> a) -> BinADT (i,a) (M.FiniteMap i a)
array f = ADT (fromB M.emptyFM accum) (toB' M.isEmptyFM split)
  where accum (i,x) a = M.accumFM a i f x
        split a = (x,a') where Just (a',x) = M.splitMinFM a

bag    :: Ord a => BinADT a (M.FiniteMap a Int)
bag    = ADT (fromB M.emptyFM add) (toB' M.isEmptyFM split)
  where add x b = M.accumFM b x (+) 1
        split b = (x,b') where Just (b'',(x,c)) = M.splitMinFM b
              b' = if c==1 then b'' else M.addToFM b'' x (c-1)

data Tree a = Leaf | Branch {key::a, left,right::Tree a}
isLeaf Leaf = True
isLeaf _    = False

tree   :: SymADT (Ternary a) (Tree a)
tree   = ADT (fromT Leaf Branch) (toT isLeaf key left right)

graph  :: BinADT (Context a b) (Graph a b)
graph  = ADT (fromB empty embed) (toB' isEmpty matchAny)

```

Figure 5: Example ADTs.