The Categorical Imperative – Or: How to Hide Your State Monads

Martin Erwig

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

Abstract. We demonstrate a systematic way of introducing state monads to improve the efficiency of data structures. When programs are expressed as transformations of abstract data types – which becomes possible by suitable definitions of ADTs and ADT fold operations, we can observe restricted usage patterns for ADTs, and this can be exploited to derive (i) partial data structures correctly implementing the ADTs and (ii) imperative realizations of these partial data structures. With a library of ADTs together with (some of) their imperative implementations, efficient versions of functional programs can be obtained without being concerned or even knowing about state monads. As one example we demonstrate the optimization of a graph ADT resulting in an optimal imperative implementation of depth-first search.

1 Introduction

State monads are essentially used to (i) to make the passing of state through successive function calls more explicit and (ii) to enable the imperative implementation of updates to the state. The latter is possible, since state monads ensure a single-threaded use of the state. Coming from category theory, monads enjoy some useful laws and are mathematically elegant. However, it seems that state monads are not very easy to use – at least for the average functional programmer. Another aspect is that part of the elegance of functional programs is lost when using monads, and since state monads extend as far as one state is needed, they tend to "infect" large parts of functional programs. This is certainly due to the restrictions monads put on state based computations. Nevertheless, state monads are sometimes needed in Haskell[1] to achieve efficient implementations for some algorithms. Now it is a tempting idea to be able to use state monads implicitly without ever seeing them or even knowing about their existence. In other words, we would like to rely on certain parts of a program to be executed very efficiently through the use of imperative features *without* programming these imperative parts (ourselves). Are we promising a heaven on

[1] References in ML are easier to use, and they are also much more flexible. However, in contrast to state monads, they do not guarantee referential transparency.

earth? Of course, the imperative program fragments have to be generated somehow,[2] but the important point is that they need not be merged with the rest of the program, they can reside in separate modules and can be selected by an optimizer when appropriate.

This factorization of imperative program parts becomes possible by a specific programming style in which programs are expressed by transformations of data types. Without knowing the details, ask yourself: what happens by transforming a list into a set and then back into a list? – Yes, this is a function for removing duplicates. Or, transforming a list into a heap and again back into a list? – This is nothing but heapsort. These two examples worked well, since the involved data structures are well understood, and since there is an agreed way of how to build and inspect values of these types. In contrast, it is not so clear what happens when we transform lists into natural numbers. What should that mean at all? It clearly depends on what data type for natural numbers we have in mind, in particular, how natural numbers are built. One view is to count from zero upwards, and taking this data type, the described transformation realizes counting (that is, the `length` function). But if we think of constructing natural numbers by addition, the transformation rather describes the `sum` function (of course, this works only for lists containing numbers). Hence, the meaning of transformations depends on the precise definition of the participating data types. Once these are given, a transformation proceeds in a structured and exactly prescribed way; it defines a fixed recursion pattern on data types, which is well-known for lists under the name `foldr`. We describe this programming style in Sect. 2.

The restricted usage patterns of data types in transformations offer specific techniques for their implementation. For example, data type values are single-threaded within a transformation which suggests imperative implementations. But the usage of data types is even more restricted: beyond being single-threaded, the sequence of possible function applications is also restricted which allows even more liberal implementations. For example, destructed values are not subjected to constructor applications so that implementations can choose data type representation that only support destructors. We describe these issues in some detail in Sect. 3. This prepares for the later introduction of imperative data structures. In Sect. 4 we present an application using a graph ADT, and Sect. 5 then describes the introduction of imperative implementations by state monads. We finish the paper by commenting on related work in Sect. 6 and by giving some conclusions in Sect. 7.

2 Folds for Abstract Data Types

In this section we briefly review our proposal for extending the structured recursion discipline to abstract data types. For a more comprehen-

[2] Presently, we assume that a skilled programmer – one of those who loves to program with state monads – has done this, but we are currently investigating possibilities for automated generation.

sive introduction, see [5]; the formal categorical background is developed in [3]. Both papers and some Haskell source files can be obtained from `http://www.fernuni-hagen.de/inf/pi4/erwig/meta`.

2.1 Representation of Algebraic Data Types

A definition of data type `T` introduces a set of constructors `c1`, ..., `cn` which can be viewed as functions of types `T1 -> T`, ..., `Tn -> T`. The common result type `T` is also sometimes called the *carrier* of the data type. For instance, the list data type (`T` = `[a]`) is defined by the two constructors `[] :: [a]` (with empty argument type) and `(:) :: a -> [a] -> [a]`. It can be thought of being defined by:

```
data [a] = [] | a : [a]
```

In order to define operations (such as `fold`) for arbitrary data types we need a way of encoding data types in Haskell itself. This can be achieved by combining all constructors into one constructor mapping from the separated sum of argument types to the carrier. The union of argument types can be denoted by specific type constructors. For example, the argument type structure for the list data type is captured by the following type constructor.

```
data Linear a b = UnitL | ProdL a b
```

Now `Linear a` applied to `[a]` denotes the proper argument type for the combined list constructor, which can then be defined as:

```
cList :: Linear a [a] -> [a]
cList UnitL       = []
cList (ProdL x l) = x:l
```

As another example consider a data type for natural numbers made up by a zero constant and a successor function. To combine both into one constructor we need another type constructor:

```
data Const a = UnitC | ProdC a
```

Now we can define the constructor using built-in integer numbers as the carrier:

```
cNat :: Const Int -> Int
cNat UnitC     = 0
cNat (ProdC n) = succ n
```

Thus, in general, an algebraic data type can be regarded simply as a function `c :: f t -> t` where `f` is the type constructor denoting the argument type union of the constructor(s) `c`.

2.2 Type Constructors as Functors

As far as constructors and algebraic data types are concerned we can use arbitrary type constructors to describe the argument type structure. However, the definition of ADT fold requires that destructors (the dual of constructors) are defined with result types being expressed by type constructors that offer a `map` function. In Haskell there is a predefined constructor class `Functor` for this (in the following we abbreviate `Functor` by `F` to save some line breaks):

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

Now `Linear` and `Const` are both examples of functors. The corresponding instance declarations are:

```
instance F (Linear a) where
  map f UnitL       = UnitL
  map f (ProdL x y) = ProdL x (f y)

instance F Const where
  map f UnitC     = UnitC
  map f (ProdC x) = ProdC (f x)
```

`Linear` is a binary type constructor, and the generalization of the functor property to the binary case is captured by the constructor class `BiF` that offers a function `map2` for mapping along both type parameters.

```
class BiF 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
```

The instance definition for `Linear` is:

```
instance BiF Linear where
  map2 f g UnitL       = UnitL
  map2 f g (ProdL x y) = ProdL (f x) (g y)
```

The definitions for `cList` and `cNat` have a uniform shape, and in fact, most of the constructor and destructor definitions are of a specific form that can be captured by standardized functions to map from and to functor type constructors. For example, a canonical way of mapping from and to `Linear` types is:

```
fromL :: t -> (a -> b -> t) -> Linear a b -> t
fromL u f UnitL       = u
fromL u f (ProdL x y) = f x y

toL :: (t -> Bool) -> (t -> a) -> (t -> b) -> t -> Linear a b
toL p f g x = if p x then UnitL else ProdL (f x) (g x)
```

This allows us to denote the list constructor much more succinctly:

```
cList = fromL [] (:)
```

Similarly, we can define `fromC` and `toC`.

```
fromC :: t -> (a -> t) -> Const a -> t
fromC u f UnitC     = u
fromC u f (ProdC x) = f x

toC :: (t -> Bool) -> (t -> a) -> (t -> Const a)
toC p f x = if p x then UnitC else ProdC (f x)
```

We can then give the following definition for `cNat`:

```
cNat = fromC 0 succ
```

Applications of `toL` and `toC` follow below.

2.3 Destructors and Abstract Data Types

A *data type destructor* is the dual of a constructor, that is, a function `d :: t -> g t` mapping the carrier to, for example, a union of possible result types. For each algebraic data type `cT :: f t -> t` we can easily define its canonical destructor `dT :: t -> f t` by simply flipping both sides of the definition. For example, the list destructor is defined by:

```
dList :: [a] -> Linear a [a]
dList []    = UnitL
dList (x:l) = ProdL x l
```

Again we can give a shorter version by using `toL`, the dual of `fromL`:

```
dList = toL null head tail
```

The definition of the canonical destructor for `cNat` is similar.

```
dNat = toC (==0) pred
```

Of course, we can also define destructors for non-free data types (that is, equationally constrained data types). For example, a destructor for sets (based on a list carrier) is obtained as follows.

```
dSet :: Eq a => [a] -> Linear a [a]
dSet = toL null head rest
       where rest (x:xs) = filter (/=x) xs
```

Now we can define an abstract data type as a pair of constructor and destructor with a common carrier type. We need no restriction on the argument type of the

constructor, but we require the result type of the destructor to be given as an application of a functor to the carrier type. This is necessary, since the definition of fold uses the function `map` to fold recursive occurrences of `t`-values.

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

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

Occasionally, we will use the following type abbreviations:

```
type SymADT g t = ADT (g t) g t
type LinADT a t = SymADT (Linear a) t
```

We can now define `list`, `set`, and `nat` ADTs simply by:

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

set :: LinADT a [a]
set =  ADT cList dSet

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

We are, of course, not constrained to symmetric ADTs. We can define many ADT variants by exchanging either the constructor or the destructor. Concerning natural numbers, for example, we can use multiplication as a binary constructor or a function that additionally splits off the number being decomposed as a destructor:

```
prod :: ADT (Linear Int Int) Const Int
prod =  ADT (fromL 1 (*)) dNat

rng  :: ADT (Const Int) (Linear Int) Int
rng  =  ADT cNat (toL (==0) id pred)
```

Many more examples can be found in [5].

2.4 ADT Folds and Transformers

Fold operations on algebraic data types are typically defined with the help of pattern matching: applied to a value `v`, fold determines the outermost constructor of `v` and applies an appropriate parameter function (that conforms to the type of the disclosed constructor). Formally, a fold is defined as a homomorphism from the argument type to some result type, and this definition works only if the result type is a quotient of the argument type (because otherwise the homomorphism would not be uniquely defined). In other words, fold cannot map

to less constrained structures. This is the reason that, for example, counting the elements of a set *cannot* be expressed as a fold operation. This restriction can be lifted if fold is not based on pattern matching, but on explicitly defined data type destructors.

Folding an ADT value of type `t` with a parameter function `f` then works as follows: first, the ADT destructor is applied, yielding a value `v` of type `g t`. Intuitively, one part of `v` contains values that are taken from (or split off) the ADT, and the other part represents the recursive occurrence(s) of `t`-values. The recursive part is then folded, followed by an application of `f` to the result and the non-recursive part of `v`. The recursive folding step is realized by using the function `map` that is defined for the functor `g`, and we thus see the need for at least the type of ADT destructors being expressed by a functor expression.

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

We observe that `fold`'s parameter function must map from the functor of the ADT to the result type. For example, a function that multiplies all numbers in a list can be written as a fold:

```
mult :: Num a => [a] -> a
mult = fold (fromL 1 (*)) list
```

In this example it is striking that the parameter function of fold is nothing but a constructor of an ADT, namely the ADT `prod`. This special case occurs very often and is important enough to introduce an own definition for it. We call this kind of fold an *ADT transformer*. In order to be able to transform an ADT with a result type functor `g` into an ADT whose argument type does not match `g` (that is, whose argument type cannot be expressed by an application of `g`) we include a parameter function that can be used to adjust the two type structures.

```
trans :: (F g,F 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
```

In an expression `trans f a b` we call `a` the *source ADT* and `b` the *target ADT*; `f` is called the *map* of the transformer. When the type structures of the source and the target ADT agree, we have `f` = `id`. We abbreviate this case by:

```
transit = trans id
```

We can thus rewrite the above example as:

```
mult = transit list prod
```

To understand the need for the additional parameter of `trans`, consider the task of determining the length of a list. We can express this as a transformer from `list` to `nat`. However, the results delivered by `dList` have not the proper shape for applying `cNat`, and we must provide a map from `Linear` to `Const`:

```
length = trans p2 list nat
         where p2 UnitL       = UnitC
               p2 (ProdL _ y) = ProdC y
```

Note that virtually the same definition works for sets, too, that is, if we simply replace `list` by `set` we obtain a proper function for determining the cardinality of sets. This is because folds and transformers are based on destructors, and the set destructor normalizes the set representation, that is, it removes duplicates, so that each set element is counted exactly once.

By composing two or more transformers we can build streams of ADTs which can be used like filters. The most common case is to compose two transformers:

```
via::(F g,F h,F 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
```

With `via` we can now formulate the examples mentioned in the Introduction.

```
remdup   = via list set list
heapsort = via list pqueue list
```

2.5 Laws

The proposed extension of folds to ADTs is conservative in the sense that laws and optimization techniques developed for algebraic data types are still valid in the extended framework. The purpose of this section is to collect some of the laws that are needed later on in the paper.

We have already observed that `trans` is a special case of `fold` in the sense that the parameter function for `fold` is obtained from an ADT. In a similar way we can generalize `fold` by relaxing the requirement that the destructor must be taken from an ADT. We then arrive at a function definition that is essentially a hylomorphism [13, 17].

```
hylo :: F f => (f b -> b) -> (a -> f a) -> (a -> b)
hylo c d = c . map (hylo c d) . d
```

Now we can also specialize the first parameter, and we obtain a definition for `unfold`.

```
unfold :: (F f,F g) => (t -> f t) -> ADT (f u) g u -> (t -> u)
unfold f a = con a . map (unfold f a) . f
```

`fold` and `unfold` are related in a natural way to `hylo`:

```
        fold f a = hylo f (des a)                (FoldHylo)
      unfold f a = hylo (con a) f                (UnfoldHylo)
unfold (des a) b = fold (con b) a                (UnfoldFold)
```

Since `trans` is actually nothing but a special case of `fold` we also know:

```
trans f a b = hylo (con b.f) (des a)             (TransHylo)
```

A source of many useful fusion laws is the free theorem [18] for the type of `hylo`:

Theorem 1 (FreeHylo).

```
l.c = c'.map l ∧ d'.r = map r.d ⟹
l.hylo c d = hylo c' d'.r
```
□

We can derive two specialized versions from the FreeHylo theorem for fusion from the left and from the right: (1) let `r = id` and `d' = d` (then the second premise of the theorem is trivially true) or (2) let `l = id` and `c' = c` (then the first premise of the theorem is trivially true).

Corollary 1 (HyloLeft).

```
l.c = c'.map l ⟹ l.hylo c d = hylo c' d
```
□

Corollary 2 (HyloRight).

```
d'.r = map r.d ⟹ hylo c d'.r = hylo c d
```
□

Note that, in general, FreeHylo and HyloLeft hold only for strict functions `l`.

There exist many more laws, in particular, we have also free theorems for `fold` and `unfold` and several fusion laws that follow from these, but for brevity we omit them here.

3 Partial Data Structures

When programming with folds or transformers, ADT values are always built or decomposed in one run. In particular, no intermediate version that emerges during construction or destruction is visible from the outside, and we can try to develop efficient constructors and destructors that utilize this fact in some way.

First of all, we have to define what "intermediate values", called *temporaries*, exactly are. For this purpose we consider in the sequel the following two ADTs

```
a :: ADT s g t
a =  ADT ca d

b :: ADT (g u) h u
b =  ADT c db
```

and a function

```
g :: t -> u
g = trans f a b
```

Assume that the application of `g` to the argument `x :: t` yields the result `y :: u`. Then all values of type `t` that are created during the evaluation of `g x` (that is, all `t`-values except the argument `x` itself) are `t`-temporaries. Likewise, all values of type `u` that are created during the evaluation of `g x`, except the result `y`, are `u`-temporaries.

Let us now consider the properties of temporaries in more detail: first, it is clear that each temporary is used exactly once by either a constructor or a destructor, that is, the intermediate ADT values are *single-threaded*. It is well

known that single-threaded data structures can be safely implemented in an imperative way. Moreover, temporaries enjoy restricted access patterns. More specifically, temporaries used in a constructing phase are never subject to an application of a destructor, that is, in the example we will never encounter an expression like `db y'` for any `u`-temporary `y'`. Similarly, a constructor will never be applied to a temporary created within a destruction phase, that is, in the example we will never see an expression `(con e) x'` for any `t`-temporary `x'` (and for a suitable ADT `e`).

This information can be used as follows: whereas the representation of `t`- and `u`-values in general must support application of constructors *and* destructors, this is not the case for `t`-temporaries and `u`-temporaries, that is, a representation for `t`-temporaries must only support destructor application, and a representation for `u`-temporaries has only to account for constructors. We can therefore try to use such specialized representations in the evaluation of functions like `g` to gain efficiency.

We call this kind of data structures supporting only a subset of operations *partial data structures*. Investigating partial data structures leads in quite the opposite direction as functional data structures do: whereas functional data structures have to account for additional requirements [15] (that is, persistent access and different versions) partial data structures can exploit the fact that they are used in a restricted way.

Requirements	low ..... high
	Partial DS $\longleftarrow$ *Imperative DS* $\longrightarrow$ *Functional DS*

Now let `t'` and `u'` be appropriate types for representing the destructor-limited view of `t`, respectively, the constructor-limited view of `u`. To compute `g` based on `t'` or `u'` we need two functions to map into, respectively, out of, the specialized representations:

```
into  :: t  -> t'
outof :: u' -> u
```

and we need modified constructors/destructors:

```
d' :: t' -> g t'
c' :: g u' -> u'
```

There are four possible cases of how and where specialized representations can be used:

1. Neither in the source nor in the target ADT
2. Only in the source ADT
3. Only in the target ADT
4. In both the source and the target ADT

Now the reformulation of `g` depends on the places where specialized representations are to be used. In the first case, no reformulation is possible; we consider the remaining cases in the next three subsections.

3.1 Specialized Representation in Source ADTs

In this case we essentially employ a modified destructor `d'` working on the specialized representation `t'`. If we persisted in using `trans` for expressing `g'`, we would have to define a whole new source ADT with carrier `t'` instead of just introducing a new destructor. This would affect the modularity of our ADT concept, and being forced to invent a new constructor on the new carrier `t'` would be highly inconvenient. We therefore use an extended definition for ADTs that allows to deal with two different (but related) carriers. In addition to constructor and destructor we also need a *carrier map*, that is, a function that maps from the constructor carrier to the carrier used by the destructor.

```
data F g=>ADT2 s t g t' = ADT2 (s -> t) (t -> t') (t' -> g t')

con2 (ADT2 c _ _) = c
cmap (ADT2 _ m _) = m
des2 (ADT2 _ _ d) = d
```

The idea behind the definition of `ADT2` is to construct a value of an ADT based on one carrier `t` while being able to destruct the same value with a destructor working with a possible different carrier `t'`. We can now define a specialized ADT `a'` simply by:

```
a' = ADT2 (con a) into d'
```

Next we need a transformer definition that employs an extended ADT as source:

```
transS :: (F g,F h)=>(g u->r)->ADT2 s t g t'->ADT r h u->(t->u)
transS f (ADT2 _ m d) (ADT c _) = h . m
         where h = c . f . map h . d
```

We observe that the constructor of the source ADT is not needed at all. This fact can be used to give an alternative definition for `transS` which is based on `trans`. This will facilitate the comparison of both kinds of transformers and also sometimes correctness proofs for functions employing specialized ADTs. We therefore introduce first an operator that maps an extended ADT back to an "ordinary" ADT:

```
toADT :: F g => ADT2 s t g t' -> ADT s g t'
toADT (ADT2 c m d) = ADT (m.c) d
```

The constructor `m.c` of the resulting ADT makes, in general, not much sense, but this is not a problem, since we use such a trimmed ADT only for its destructor. We can now give the following modified definition for `transS`:

```
transS f a b = trans f (toADT a) b . cmap a          (TransS)
```

Again, we also use the special case for `f` = `id`:

```
transitS :: (F g,F h)=>ADT2 s t g t' -> ADT (g u) h u -> (t->u)
transitS = transS id
```

With the help of `transitS` we can now define `g` very easily based on `a'`:

```
g' = transitS a' b
```

Now the question is whether the described transformation is correct at all. In other words, under which conditions does `g = g'` hold? All we have actually done in the definition of `g'` is to exchange the destructor, working on `t'`-values that are obtained from `t`-values by an application of `into`. Formally, we have the following relationship (we still assume `a = ADT ca d` and `b = ADT c db`).

Theorem 2 (Destructor Specialization).
`d'.m = map m.d` $\Longrightarrow$
`trans f a b = transS f (ADT2 ca m d') b`

Proof. Let `a' = ADT2 ca m d'`, and `a'' = toADT a' = ADT (m.ca) d'`. In particular, we know `des a'' = des2 a' = d'`. Now we have:

```
transS f a' b = trans f a'' b.m          {TransS}
              = hylo (c.f) d'.m          {TransHylo}
              = hylo (c.f) d             {HyloRight}
              = trans f a b              {TransHylo}        □
```

To take a simple example, consider the ADT `queue` and a transformer from `queue` into `list`:

```
queue :: LinADT a [a]
queue = ADT cList dQueue
        where dQueue = toL null last init

ql = transit queue list
```

The destructor `dQueue` is very inefficient, since it takes elements always from the end of the list representation. This causes `ql` to run in quadratic time. We can do much better if we use an extended ADT with

(1) `t' = t = [a]`
(2) `d' = toL null head tail` (= `dList`)
(3) `into = rev`

This means to construct as well as destruct a queue represented as a list from the front. For this to work we need two different list representations of queues, and the representation used for destruction is just obtained by reversing the list of the construction phase. So we have:

```
ql' = transitS (ADT2 cList rev dList) list
```

We can prove `ql = ql'` by using Theorem 2. We therefore first we have to establish the precondition, that is, we have to show:

Lemma 1. `dList.rev = map rev.dQueue`

Proof. We consider the two cases of empty and non-empty lists. Note below that `map` is the map function for type `Linear` and not for lists.

```
(dList.rev) []          = dList []
                        = UnitL
                        = map rev UnitL
                        = map rev (dQueue [])
                        = (map rev.dQueue) []

(dList.rev) (xs++[x]) = dList (rev (xs++[x]))
                        = dList (x:rev xs)
                        = ProdL x (rev xs)
                        = map rev (ProdL x xs)
                        = map rev (dQueue (xs++[x]))
                        = (map rev.dQueue) (xs++[x])
```

□

We can now apply Theorem 2 and obtain:

Corollary 3. `ql = ql'` □

This example was fairly simple, it has been chosen to demonstrate the ideas without loading too many details. We will deal with a more realistic application in Sect. 4.

3.2 Specialized Representations in Target ADTs

The situation here is similar to that of the preceding section. We can specialize a target ADT `b` by giving a new constructor `c'` and an appropriate carrier map:

```
b' = ADT2 c' outof (des b)
```

A transformer definition that uses an extended ADT as target can be given dually to `transS`:

```
transT :: (F g,F h)=>(g u->r)->ADT s g t->ADT2 r u h u'->(t->u')
transT f (ADT _ d) (ADT2 c m _) = m . h
         where h = c . f . map h . d
```

(We cannot give an alternative definition for `transT` based on `trans`, since in order to define an analog to the function `toADT` we needed something like the inverse of `m` to build from the destructor of the extended ADT of type `u' -> h u'` a dummy function of type `u -> h u`.) But we can still define `transitT`:

```
transitT :: (F g,F h)=>ADT s g t -> ADT2 (g u) u h u' -> (t->u')
transitT = transT id
```

With the help of `transitT` we can now implement the function `g` using a specialized target ADT `b'`:

```
g' = transitT a b'
```

Again, we can ask when `g = g'` does hold, and we have a result similar to Theorem 2 (let `a = ADT ca d` and `b = ADT c db`).

Theorem 3 (Constructor Specialization).
`m.c'` = `c.map m` $\wedge$ `m` *is strict* $\Longrightarrow$
`transit a b` = `transitT a (ADT2 c' m db)`

Proof. Let `b' = ADT2 c' m db`. We know `con b' = c'`. Moreover, let `h = c'.map h.d`. Now we have:

```
transitT a b' = m.h                 { Def. of transT }
              = m.hylo c' d         { Def. of hylo }
              = hylo c d            { HyloLeft }
              = transit a b         { TransHylo }
```
□

To give an example consider the transformer

```
lq = transit list queue
```

where we assume that the `queue` ADT is defined to enqueue at the end of a list and to dequeue from the front:

```
queue = ADT snoc dList
        where snoc = fromL [] (\x q->q++[x])
```

Here `lq` (like `ql`) takes quadratic time, which can be improved to linear time by using an extended ADT with

(1) `t' = t = [a]`
(2) `c' = fromL [] (:)` (= `cList`)
(3) `outof = rev`

Again, this means to construct as well as destruct a queue represented as a list from the front, and the two different list representations are converted by list reversal.

```
lq' = transitT list (ADT2 cList rev dList)
```

We prove that `lq = lq'` with the help of Theorem 3. We first show:

Lemma 2. `rev.cList` = `snoc.map rev`

Proof. Consider the two cases of type `Linear`:

```
(rev.cList) UnitL          = rev []
                           = []
                           = snoc UnitL
                           = (snoc.map rev) UnitL

(rev.cList) (ProdL x xs) = rev (x:xs)
                           = rev xs++[x]
                           = snoc (Linear x (rev xs))
                           = (snoc.map rev) (ProdL x xs)        □
```

We can now apply Theorem 3 and obtain:

Corollary 4. `lq = lq'` □

3.3 Specialized Representations in Source and Target ADTs

Combining the results of the preceding two subsections we can use specialized representations within both the source ADT destructor and the target ADT constructor. For this we need a further kind of transformer:

```
transST::(F g,F h)=>(g u->r)->ADT2 s t g t'->ADT2 r u h u'->(t->u')
transST f (ADT2 _ ma d) (ADT2 c mb _) = mb . h . ma
        where h = c . f . map h . d
```

Again we let

```
transitST = transST id
```

We can also express `transS` and `transT` as special cases of `transST` by canonically extending a target or source ADT with an identity carrier map.

```
ext a = ADT2 (con a) id (des a)

transS f a b = transST f a (ext b)
transT f a b = transST f (ext a) b
```

We can also combine Theorems 2 and 3:

Theorem 4 (Double Specialization).
`mb.c' = c.map mb` $\wedge$ `mb` *is strict* $\wedge$
`d'.ma = map ma.d` $\Longrightarrow$
`transit a b = transitST (ADT2 ca ma d') (ADT2 c' mb db)`

Proof. Let `a' = ADT2 ca ma d'` and `b' = ADT2 c' mb db`. We know `des a' = d'` and `con b' = c'`. Moreover, let `h = c'.map h.d'`. Now we have:

```
transitST a' b' = mb.h.ma                 { Def. of transST }
                = mb.hylo c' d'.ma        { Def. of hylo }
                = hylo c d'.ma            { HyloLeft }
                = hylo c d                { HyloRight }
                = transit a b             { TransHylo }        □
```

4 An Advanced Example

Graphs are among the most complicated (basic) ADTs to deal with. So they provide a good "benchmark" for the expressiveness of the proposed ADT formalism. Moreover, since the efficient treatment of graphs in purely functional languages is not trivial, this is also a good test for the claimed optimization opportunities.

4.1 Defining Graph ADTs

To cast graphs into the proposed ADT formalism we use the inductive graph view presented in [2]: a graph is either empty, or it is constructed by adding a node together with edges to its predecessors and successors. `Graph a b` denotes the type of graphs with node (edge) labels of type `a` (`b`), and a node context is a labeled node together with a list of labeled incoming and outgoing edges.

```
type Node        = Int
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
```

which can be combined as follows.

```
cGraph :: Linear (Context a b) (Graph a b) -> Graph a b
cGraph = fromL empty embed
```

The decomposition of graphs can be defined in quite different ways. A simple solution is to split off an arbitrary node from the graph. However, this limits the number of problems that can be solved by graph transformers. In contrast, decomposition of specific nodes offers control over the decomposition order and provides much flexibility. We therefore use a function `match` that retrieves and removes a particular node from a graph. Since the request for decomposing a specific node might fail, `match` is defined to return `Maybe` contexts:

```
type MContext a b = Maybe (Context a b)

match :: Node -> Graph a b -> (MContext a b,Graph a b)
```

Next we define a graph ADT that is specialized to support depth-first search.

4.2 Depth-First Search

We shall express depth-first search as a transformer from a (stack, graph) ADT to a list ADT. The stack is needed to control the decomposition order of the graph: the context to be taken next from the graph is determined by the top of the stack, and the successors from the last decomposed context are pushed onto the stack to prepare the next decomposition. We implement the stack as a list.

```
type StkGraph a b = ([Node],Graph a b)
```

The graph destructor is now defined as follows: decomposition immediately stops when the stack is exhausted or when the graph to be decomposed is empty. Otherwise, the top of the stack `v` is matched in the graph which yields a remaining graph `g'` and possibly a context value `Just c` (or `Nothing` if `v` is not contained in `g`). The list of successors of `c` (or `[]`) is pushed onto the stack, and the context `c` together with the new (stack, graph) pair is returned as the result.

```
dGraph :: StkGraph a b -> Linear (MContext a b) (StkGraph a b)
dGraph (s,g) = if null s || isEmpty g then UnitL else
                  ProdL c (suc++tail s,g')
                        where (c,g') = match (head s) g
                              suc    = maybe [] (map snd.q4) c
                              q4 (_,_,_,d) = d
```

Since we are here only interested in destructing a graph, we use this destructor together with a dummy constructor to define a graph ADT.

```
graph :: ADT () (Linear (MContext a b)) (StkGraph a b)
graph =  ADT (\_->([],empty)) dGraph
```

Depth-first search is now realized by simply transforming a graph into a list which collects `Just`-contexts and ignores `Nothing`-values. Since we are interested only in the node values, we select the second component of the context tuple. This projection function `nid` and the `Maybe`-variant of the list ADT `mlist` are given below (note that `map` is the map function for the functor `Maybe`).

```
nid :: Linear (MContext a b) g -> Linear (Maybe Node) g
nid = mapFst (map q2)
      where q2 (_,b,_,_) = b

mlist :: ADT (Linear (Maybe a) [a]) (Linear a) [a]
mlist = ADT mList dList
        where mList UnitL               = []
              mList (ProdL Nothing xs)  = xs
              mList (ProdL (Just x) xs) = x:xs
```

Note that `dfsn` needs an initial stack value, and `dfs` uses by default a list of all graph nodes as this initial value (to be able to completely explore even unconnected graphs).

```
dfsn :: ([Node],Graph a b) -> [Node]
dfsn = trans nid graph mlist

dfs :: Graph a b -> [Node]
dfs g = dfsn (nodes g,g)
```

This implementation of `dfs` bears some inefficiency because of the need to construct intermediate graphs, which means to remove decomposed nodes from successor and predecessor lists. Assuming that graphs are represented by balanced search trees mapping nodes to their contexts, `dfs` runs in $O(n + e \log n + nd^2)$ time where n and e denote the number of nodes and edges in the graph and d is the average node degree. The second term represents the cost for searching the predecessor/successor lists for removing edges. After a context has been found, these lists have to be scanned to remove a node, and they contain, on the average, initially d, after the first removal $d-1$, etc. nodes. Thus, the context of each node will be reduced in $d + (d-1) + \ldots + 1$ steps, and the last term represents the total cost resulting from the reorganization of all contexts. Hence, for sparse, respectively, dense, graphs `dfs` needs $O(n \log n)$, respectively, $O(n^3)$, time.

4.3 Graph Destructor Specialization

Next we describe how to improve the efficiency of `dfs` by devising a partial graph destructor. So how can we save the construction of intermediate graphs? The key idea is to delay the node deletion. This means, instead of removing a node from the graph immediately after decomposition we remember decomposed nodes and remove them from returned contexts. We therefore extend the carrier for decomposition by a set recording the already decomposed ("visited") nodes.[3]

```
type StkGraph' a b = (StkGraph a b,Set Node)
```

The node set is initially empty, so we use the following `into` function that just extends a (stack, graph) pair by an empty set:

```
unvisit x = (x,emptySet)
```

The set of nodes is extended within the modified destructor `dGraph'`: whenever the context of a node `v` is requested the first time, `v` is inserted into the set of visited nodes. The improved performance of `dGraph'` results from the use of the function `context` instead of `match`: whereas `context` only has to find a node's context, `match` has to build a representation of the graph from which the matched node has been removed. The function `context` is otherwise very similar to `match`, it can be thought of being implemented by:

[3] For this purpose the discrete interval encoding tree [4] is suited very well, since its performance improves with the density of the stored subsets, and for graph algorithms, such as `dfs`, the set of visited nodes constantly grows until the complete node set is stored.

```
context :: Node -> Graph a b -> Context a b
context v g = case match v g of
                (Just c,_) -> c
                _          -> error ("Matching node: "++show v)
```

Now we can define `dGraph'`:

```
dGraph'::StkGraph' a b->Linear (MContext a b) (StkGraph' a b)
dGraph' (([],_),_)   = UnitL
dGraph' ((v:vs,g),d) =
   if isEmpty g then UnitL else
   if member v d then ProdL Nothing ((vs,g),d) else
      ProdL (Just (p',v,l,s')) ((map snd s'++vs,g),insert v d)
      where (p,_,l,s) = context v g
            s' = filter (\(_,w)->not (member w d)) s
            p' = filter (\(_,w)->not (member w d)) p
```

Hence, we can use an extended graph ADT with:

(1) `t'` = `StkGraph' a b`
(2) `d'` = `dGraph'`
(3) `into` = `unvisit`

We can thus obtain the following optimized implementation for depth-first search.

```
dfsn' = transS nid (ADT2 (\_->([],empty)) unvisit dGraph') mlist
dfs' g = dfsn' (nodes g,g)
```

Since all set insertions take $O(n \log n)$ steps and all set membership tests need $O(e \log n)$ time, `dfs'` runs in $O((n+e) \log n)$ time (even when representing graphs as immutable arrays mapping nodes to their contexts, which is now possible because the graph itself need not be changed). By using Haskell's imperative state transformers this optimization can be improved eventually resulting in an optimal $O(n+e)$ running time, see Sect. 5.1.

5 Using Imperative Data Structures

The use of imperative data structures has two aspects: first, we can use imperative functions as carrier maps in extended source or target ADTs, and we can thus exploit efficient imperative constructors or destructors without affecting referential transparency. Second, we can use completely imperative data structures for stream ADTs.

5.1 Imperative Constructors and Destructors

As an example we consider the specialized graph destructor `dGraph'`. The goal is to implement the node set imperatively, for example, by using an imperative array. In Haskell we have to encapsulate state changing operations within a state thread which roughly works according to the following scheme: open a state thread, create variables, perform all updates, and close the state thread possibly returning a value.

Now it seems that ADT folds and transformers do not harmonize with this way of imperative programming because a fold or transformer produces results incrementally: decompose a value `x` yielding a value `y`, return the nonrecursive part of `y`, and continue folding the recursive part of `y`. Concerning `dGraph'` this means that in each decomposition step contexts are delivered as well as the node set is updated. In contrast, a state transformer does not allow to emit values out of a running computation; a value can be returned at the earliest after all updates have been performed.

So can we use state transformers at all to implement the node set in the graph destructor imperatively? A possible solution is to place a state transformer in the carrier map where it imperatively updates a node array as desired and accumulates the contexts that are computed during the decomposition in a list. This list is returned as the result of the transformer. The destructor of the extended ADT is then simply an ordinary list destructor that just decomposes the list of contexts obtained by the carrier map. The definition of an imperative graph destructor is shown below.

```
dGraphI :: StkGraph a b->STArray s Node Bool->ST s [Context a b]
dGraphI ([],_) _ = return []
dGraphI (v:vs,g) a =
        if isEmpty g then
           return []
        else do {
           b <- readSTArray a v;
           if b then do {
              cs <- dGraphI (vs,g) a;
              return cs }
           else do {
              let {c = context v g; suc = map snd (q4 c)};
              writeSTArray a v True;
              cs <- dGraphI (suc++vs,g) a;
              return (c:cs)
           }
        }
```

If the stack or the graph is empty, an empty list of contexts is returned. Otherwise, the array is inspected at index `v`, the node on top of the stack, and two cases have to be distinguished: (i) If `b` is `True`, that is, if we have already visited

`v` before, we simply continue determining the list of contexts `cs` for the remaining nodes on the stack `vs`, and we return this list as a result. (ii) If we have not seen `v` before, we extract `v`'s context `c` and select the successor nodes of `c`. After that we update the array at index `v` to remember that `v` has now been processed. Then we proceed similarly to case (i): determine the list of contexts for the remaining stack onto which the successors are pushed, and return this list extended by `c`.

We can now define the function `mDfs` which is to be used as a carrier map:

```
mDfs :: StkGraph a b -> [Context a b]
mDfs (s,g) = runST (do {a  <- newSTArray (nodeRange g) False;
                        cs <- dGraphI (s,g) a;
                        return cs})
```

With `do` we create a state thread in which we first allocate a boolean array indexed by the nodes of the graph to be traversed. This array is initialized to `False` indicating that no node is visited yet. Then we determine and return the list of graph contexts with the imperative destructor just described. This state thread is then executed with the language primitive `runST`.

Note that the imperative graph destruction is completely hidden within `mDfs`. We can now define an extended ADT by:

```
graphI :: ADT2 () (StkGraph a b) (Linear (Context a b)) [Context a b]
graphI = ADT2 (\_->([],empty)) mDfs dList
```

Maybe a bit confusing is the fact that the second carrier of `graphI` is just a list of contexts; nothing about graphs is mentioned. This reflects the fact that the essential computation of `graphI` is contained in the carrier map and not in the destructor – the destructor just passes the results.

Finally, we get for depth-first search:

```
dfsnI = transS (mapFst q2) graphI list

dfsI g = dfsnI (nodes g,g)
```

For graphs that are represented as (immutable) arrays of adjacency lists, `dfsI` now runs in linear, that is, $O(n + e)$, time which is asymptotically optimal.

5.2 Completely Imperative Data Structures

When we consider ADT streams, we observe that intermediate ADTs (like `set` in `remdup` and `pqueue` in `heapsort`) are completely single-threaded. Hence, we can employ a completely imperative implementation of those ADTs. This can again be achieved with the help of `ADT2`; the general procedure is as follows. Consider an ADT

```
s :: ADT s g t
s =  ADT c d
```

and an ADT stream

```
f = via a s b
```

If `s` is canonically derived from an algebraic data type, then `s` can be omitted altogether, that is, the well-known fusion law (which is still valid in this extended framework, see [5]) can be applied, and we can directly rewrite `f` as `transit a b`. Otherwise, some real computation takes place in the constructor `c` and/or in the destructor `d`. We have then to reformulate `s` as

```
s' :: ADT2 s t g t'
s' =  ADT2 c' m d'
```

so that `c'` just collects values delivered by `des a` and `d'` only passes values from `t'` to be consumed by `con b`. This means that the actual computation must happen within `m`, and this can be realized by a state monad.

For example, with regard to `remdup` we can define an extended ADT `set'`:

```
set' :: Ix a => ADT2 (Linear a [a]) [a] (Linear a) [a]
set' =  ADT2 cList hash dList
```

where the function `hash` realizes the set semantics by first inserting and then retrieving elements from an `a`-indexed boolean array (thereby ignoring duplicates).

5.3 Fusing Carrier Maps with Constructors/Destructors

It is striking that in `dGraphI` the list of contexts is built only to be later decomposed by the destructor of `graphI`. So we can try to remove this intermediate list by fusing the functions `mDfs` and `dList`. This should be routine work for deforestation [19, 9, 17], but it is complicated here by the fact that fusion must work across state thread boundaries. How this can be done has been described by Launchbury in [11]. Using function product:

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

we can define the `map` function for the `ST` functor by:

```
map f st = (f >< id) . st
```

This means that `f` mapped to a state transformer `st` gives a new state transformer that computes the same state as `st`, but returns a value `f x` instead of `x`. In practice this means to move `f` over sequential updates and to move `f` into `return` statements:

```
map f (do s; ss)      = do s; map f ss
map f (do v <- s; ss) = do v <- s; map f ss
map f (return x)      = return (f x)
```

With `f = mapFst q2` and `s = nodes g` we obtain for `dfsI`:

```
dfsI g
= dfsnI (s,g)
= transS f graphI list (s,g)
= hylo (cList.f) dList (mDfs (s,g))
```

Now we can move the complete `hylo` expression into `mDfs`. In order to achive anything useful we have to move further into `dGraphI`. Since

```
hylo (cList.f) dList []  =  []
```

and since the result of recursive function calls are inductively assumed to be already correct, we only have to change the last `return` statement, that is, return only the node `v` (which is just the result of `q2 c`) instead of the whole context `c`. Altogether we obtain the optimized implementation for depth-first search shown below.

```
dfsnI' ([],_) _ = return []
dfsnI' (v:vs,g) a =
       if isEmpty g then
          return []
       else do {
          b <- readSTArray a v;
          if b then do {
             cs <- dfsnI' (vs,g) a;
             return cs }
          else do {
             let {c = context v g; suc = map snd (q4 c)};
             writeSTArray a v True;
             cs <- dfsnI' (suc++vs,g) a;
             return (v:cs)
          }
       }

dfsI' :: Graph a b -> [Node]
dfsI' g = runST (do {a  <- newSTArray (nodeRange g) False;
                     cs <- dfsnI' (nodes g,g) a;
                     return cs})
```

5.4 A Simple ADT Optimizer

A very simple optimizing strategy is to substitute transformers based on a library of available ADTs with specialized constructors, destructors or completely imperative ADTs. This pressumes that ADTs and their specializations are made known to the optimizer. For example, if somebody has implemented `graphI`, he

or she has to place the code into a specific place and has to announce that this is a destructor-specialized implementation of the ADT `graph` (for example, extend a function $Spec_D$ by $Spec_D(\texttt{graph}) = \texttt{graphI}$). Then an optimizer can exploit specialized ADTs along the following scheme:

$$Opt(\texttt{transit a b}) = \begin{cases} \texttt{transitST a' b'} & \text{if } Spec_D(\texttt{a}) = \texttt{a'} \wedge Spec_C(\texttt{b}) = \texttt{b'} \\ \texttt{transitS a' b} & \text{if } Spec_D(\texttt{a}) = \texttt{a'} \\ \texttt{transitT a b'} & \text{if } Spec_C(\texttt{b}) = \texttt{b'} \\ \texttt{transit a b} & \text{otherwise} \end{cases}$$

$$Opt(\texttt{trans f a b}) = \begin{cases} \texttt{transS f a' b} & \text{if } Spec_D(\texttt{a}) = \texttt{a'} \\ \texttt{trans f a b} & \text{otherwise} \end{cases}$$

Of course, the optimizer works correctly only if the preconditions of Theorems 2, 3, and 4 are satisfied for the known ADTs, and there should be an explicit deforestation pass following the application of *Opt* to eliminate gluing lists, etc.

6 Related Work

The generalization of list fold to regular algebraic data types has been thoroughly investigated [12, 13, 16, 14, 6]. In particular, exploiting fold's fixed recursion pattern for developing optimizers for functional languages has attracted much interest [9, 16, 17, 10]. However, defining folds for abstract data types has been almost neglected so far. Despite proposals for specific types, such as arrays [1] or graphs [8, 2], it is only Fokkinga [7] who attacks the problem from a general point of view. In his approach terms are represented by combinators, and equations are represented by pairs of such combinators. His treatment is done completely in categorical language, and although his proposal generalizes the case of free data types, it still constrains homomorphisms to map to quotients.

There are quite a lot of papers on the theory and use of monads – recently, Phil Wadler gave a nice survey [20] that contains most of the relevant references, but we are not aware of any work that tries to sweep monads under the rug.

7 Conclusions

We have demonstrated how to systematically insert efficient imperative data structures into functional programs. This can be automated provided that (i) programs are written in metamorphic style (that is, by using ADT transformers) and (ii) specialized, imperative ADT versions are supplied and are made known to an optimizer.

The presented programming style, extending the structured recursion discipline to abstract data types, also extends its optimization opportunities from mere fusion to the utilization of imperative data structures and algorithms.

References

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

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

[3] M. Erwig. Categorical Programming with Abstract Data Types. In *7th Int. Conf. on Algebraic Methodology and Software Technology*, 1998. To appear in LNCS.

[4] M. Erwig. Diets for Fat Sets. *Journal of Functional Programming*, 8(6), 1998.

[5] M. Erwig. Metamorphic Programming: Structured Recursion for Abstract Data Types. Technical Report 242, FernUniversität Hagen, 1998.

[6] L. Fegaras and T. Sheard. Revisiting Catamorphisms over Datatypes with Embedded Functions. In *23rd ACM Symp. on Principles of Programming Languages*, pages 284–294, 1996.

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

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

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

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

[11] J. Launchbury. Graph Algorithms with a Functional Flavour. In *1st Int. Spring School on Advanced Functional Programming*, LNCS 925, pages 308–331, 1995.

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

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

[14] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In *Conf. on Functional Programming and Computer Architecture*, pages 324–333, 1995.

[15] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

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

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

[18] P. Wadler. Theorems for Free! In *Conf. on Functional Programming and Computer Architecture*, pages 347–359, 1989.

[19] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73:231–284, 1990.

[20] P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.