An Exercise in Program Transformation Using Factorisation

J. B. Barros

jbb@di.uminho.pt

Techn. Report DI-PURe-04.02.01

2004, February

PURe
Program Understanding and Re-engineering: Calculi and Applications
(Project `POSI/ICHS/44304/2002`)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-04.02.01

An Exercise in Program Transformation Using Factorisation by J. B. Barros

Abstract

This paper uses a well known factoring algorithm as the basis of a program transformation strategy. This algorithm transforms an explicitly recursive definition into the sequential composition of a producer and a consumer of an intermediate (tree-like) structure. We show how some non-basic algorithms can be specified by characterising these intermediate data structures. This can then be used to substitute some or all of the factors by others that produce/consume similar structures. The main difference to other program transformation techniques lies in its non syntactical basis.

1 Introduction

Back in the 70's, one of the best known slogans in programming came from the title of Wirth's famous book *Algorithms + Data Structures = Programs*[12]. The whole idea of separating the two worlds seemed promising and led to a considerable improvement in the quality of the produced software.

Another consequence of this separation was the development of separate refinement calculi for both algorithms and data.

An important thread in this area was started by the famous work by Burstall and Darlington [3] on applying fold/unfold of functional definitions in order to obtain transformed versions of those programs. This process, despite being syntax oriented, was not suited for automation for it subsumed a certain amount of intuition, i.e., the understanding of what a program does and how it does it.

Another important contribution, this one in the imperative paradigm, was the work of Tony Hoare and others on the formalisation of what is now commonly referred as Floyd-Hoare Logic. In that setting, the meaning of a program is expressed based on a predicate over (some part of) the program's internal state. This has led the foundations of say, Carroll Morgan's refinement calculus [9].

More recent work on generic programming, namely the work on the characterisation of different sorting algorithms by means of their internal call tree[1], has shown the advantages of further separating the study of algorithms in their factorisation as producers and/or consumers of tree like structures.

In fact, as will be clear from the example presented in Section 3, the specification of some non-basic algorithms can be made solely by the characterisation of its intermediate data structure.

The term *algorithm* is being used here meaning a strategy to solve some particular task. This is fundamentally different from the concept of function. The later relates only the input with the output, thus forgetting how the output was obtained. In that respect, all sorting algorithms (if correct!) implement the same function. But they may implement different sorting algorithms.

The rest of the paper is divided into three sections. Section 2 introduces the basic concepts that we will need. We have tried to be as informal as possible, concentrating on giving intuitive definitions of such concepts. Here, as in rest of the paper will use Haskell [8] to code the examples. Sections 3 and 4 present two exercises on program transformation that illustrate the proposed method. For the sake of clarity, in Section 3 we give an informal account of the factoring algorithm. Finally, Section 5 points out some future directions of this work.

2 Basic Facts

When writing down the definition of a (functional) program one can follow different strategies.
A large class of programs can be structured by the form of their input. These programs are usually referred as traversals. This is the case of say, the function that computes the length of a list or the sum of all elements of a list of integers. These functions are also known as **catamorphisms** (from Greek $\kappa\alpha\tau\alpha$, downwards). On lists, and in Haskell, there is even a (higher order) function – `foldr` – which captures this type of definitions. The definition of `sum` can be structured as:

– if the input is the empty list then the result is zero
– if it is not empty, given its building blocks (i.e., its head and tail) compute the result using the head and the result of applying the same function to the tail.

```
sum l | null l    = 0
      | otherwise = h + sum t
      where (h:t) = l
```

One other way to structure the definition of a function is by using the form of its output. This is the case of most stream producing algorithms, such as the function that computes the list of the prime factors of a given natural number. These functions are also known as **anamorphisms** (from Greek $\alpha\nu\alpha$, upwards). The definition of this function can be structured as:

– check weather the result is the empty list
– if that does not happen, compute the head of the result and the input necessary to compute the tail.

```
primeF n | n == 1    = []
         | otherwise = first:primeF (n `div` first)
         where first = head [x | x <- [2..], n `mod` x == 0]
```

This classification leaves out the programs that are neither guided by the structure of their input nor of their output. These can be thought as guided by an intermediate (and sometimes called virtual) data structure and so can be factored into the (sequential) composition of: first a

producer of such a structure, and second a traversal of that structure. These functions are also known as **hylomorphisms** (from Greek $\upsilon\lambda o\sigma$, matter).
Take for instance the function that computes the factorial of a natural number.

```
factorial n | n < 1 = 1
            | otherwise = n * (factorial (n-1))
```

This standard definition can be thought as first computing a list of all numbers from that one down-to one, followed by the multiplication of all elements of the list.

```
factorial n = product (n ‘downTo‘ 1)
   where downTo u l | l > u = []
                    | otherwise = u:downTo (u-1) l
         product l | null l = 1
                   | otherwise = (head l) * (product (tail l))
```

The list produced to obtain the factorial of a number is called the intermediate data structure of this function. Note that, even though some authors call it *virtual* data structure, it is quite real: it corresponds to the (inverted) stack of return addresses stored by the interpreter.
The first two classes (catamorphims and anamorphisms) can be seen as particular cases of this third group. In the first case we just consider the building phase as the identity function, whereas in the second case the identity will act as the traversal part.
The examples above are all about lists but the generalisation of these concepts to other inductive types is quite straightforward. There are quite a few places where this generalisation is presented in a uniform way; see, for instance [6], and [4] for an elegant implementation.
Moreover there is a standard procedure to get the second definition of factorial from the first one. To our knowledge, this was first presented in [7] as part of the Hylo Project [10] and has further been implemented in [5] . For the sake of clarity, in the next section we will exemplify this procedure for a particular definition.
This process of factoring definitions thus expliciting the internal (tree-like) data structures is called **forestation**, and has not received a great amount of attention for it introduces the need to build these structures, worsening the overall performance of the programs. More often one wants to perform the opposite task – deforestation – aiming at improving the efficiency of the code[11].

3 Merge sort

According to the WIKIPEDIA (`http://en2.wikipedia.org/`, merge sort is a *particularly good example of the divide and conquer algorithmic paradigm for rearranging lists into a specified order*. Conceptually, merge sort works as follows. If the list to be sorted is longer than one item:

1. divide the unsorted list into two sub-lists of about half the size,
2. sort each of the two sub-lists;
3. merge the two sorted sub-lists back into one sorted list.

This definition is easily converted into the following Haskell code.

```
mergesort l@(a:b:abs) = merge (mergesort h1) (mergesort h2)
      where (h1,h2) = split l
mergesort l = l
```

The function `split` splits a list into two and `merge` merges two sorted lists into one sorted list.
Note however that this definition is not precise enough to ensure that this is indeed merge sort. take for instance the following definition of `split`

```
split (h:t) = ([h],t)
```

In this case the resulting sorting algorithm is no longer merge sort. The first argument of the calls to merge will always be singleton lists and so (as merging a singleton list with another sorted list is sorted insertion) this sorting algorithm is equivalent to *insertion sort*. In order to get merge sort, i.e., an algorithm that runs in $O(n \log n)$ time, we have to further satisfy point 1. above – the two sub-lists should have about the same size. One possible implementation of such behaviour is

```
split [] = ([],[])
split (h:t) = (h:b,a)
   where (a,b) = split t
```

One drawback of this algorithm is that, the most favourable scenario of the input – the case where the list is already sorted, or has big chunks of sorted parts – is not translated into a better performance of the algorithm. As it is, the modification of the algorithm in order to overcome this point seems difficult, if not impossible.
The purpose of this paper is to show how factoring an explicitly recursive definition can be used as a program understanding tool thus helping in tasks such as program transformation. This example is particularly well-suited to apply this transformation for even it's informal description can be structured into an analysis part followed by a synthesis part.

Let us then look at a factored version of the above definition of merge sort. This version can be automatically obtained by one of the implementations referred above. We'll present an informal account of that.
The whole idea is to factor an explicitly recursive definition through the construction and destruction of some intermediate (polynomial) inductive data type. As so, this type should be a solution to the (type) equation $X \cong \mathcal{F}X$ for a functor $\mathcal{F}$ defined as

$$\mathcal{F}X = a_0 + a_1 \times X + a_2 \times X^2 + \cdots + a_n \times X^n$$

Diagrammatically, we want to find functions f and g so that the following diagram commutes.

$$\begin{tikzcd}
\texttt{[a]} \arrow[rr, "f"] \arrow[dr, "\texttt{anaSplit}"] \arrow[dd, "\texttt{mergesort}"'] & & \mathcal{F}\,\texttt{[a]} \arrow[d, "\mathcal{F}\ \texttt{anaSplit}"] \\
& T \arrow[dl, "\texttt{cataMerge}"] & \mathcal{F}\,T \arrow[d, "\mathcal{F}\ \texttt{cataMerge}"] \\
\texttt{[a]} & & \mathcal{F}\,\texttt{[a]} \arrow[ll, "g"]
\end{tikzcd}$$

Note that, as $\mathcal{F}$ is polynomial, each of its components corresponds to one possible recursive behaviour. In the present case there are two such behaviours: either the function (`mergesort`) is not called at all (if the list is either empty or a singleton), or is called twice. This gives us the form of the functor. It will be of the form $\mathcal{F}X = a_0 + a_2 \times X \times X$, and the above diagram is specialised by the following

$$\begin{tikzcd}
\texttt{[a]} \arrow[rr, "f"] \arrow[dr, "\texttt{anaSplit}"] \arrow[dd, "\texttt{mergesort}"'] & & a_0 + a_2 \times \texttt{[a]} \times \texttt{[a]} \arrow[d, "\texttt{id} + \texttt{id} \times \texttt{anaSplit} \times \texttt{anaSplit}"] \\
& T \arrow[dl, "\texttt{cataMerge}"] & a_0 + a_2 \times X \times X \arrow[d, "\texttt{id} + \texttt{id} \times \texttt{cataMerge} \times \texttt{cataMerge}"] \\
\texttt{[a]} & & a_0 + a_2 \times \texttt{[a]} \times \texttt{[a]} \arrow[ll, "g"]
\end{tikzcd}$$

The function f should decide which recursive behaviour to follow and

- for the non-recursive one, should produce an element of type a_0 with which g can compute the desired result;

– for the recursive one, should produce two lists (which will be the actual parameters of the recursive calls) and (possibly) some other element of type a_2 necessary to compute the desired result together with the results of the recursive calls.

With these considerations one can conclude that the intermediate data structure produced by `anaSplit` and traversed by `cataMerge` is a binary leaf tree (where each leaf has a singleton or empty list). These functions can then be defined as

```
data LT a = L [a] | T (LT a) (LT a)

anaSplit l | null l || singleton l = L l
           | otherwise             = T (anaSplit h1) (anaSplit h2)
           where (h1,h2) = split l

cataMerge (L l) = l
cataMerge (T h1 h2) = merge h1 h2
```

The factored version of the merge sort function is just the (sequential) composition of these two.

```
mergesort = cataMerge . anaSplit
```

This transformation has two clear drawbacks:

– the three line Haskell definition of merge sort has been expanded into seven lines of code.
– the factorisation will (at least in an eager language) imply the construction of an intermediate tree, thus worsening its performance.

As pointed out above, this factorisation can be used as a program understanding tool. Some of the points in the informal description given above can now be formalised by characterising the leaf tree which is produced in the first part of the algorithm. The fact that the function `split` produces two sub-lists of *about the same size* mean that each sub-tree will have about the same size, meaning that the tree is **balanced**.
The algorithm can now be described as, for a given list of size n, first produce a balanced leaf-tree with the elements of the list (implying that the depth of the tree is $O(\log n)$). The second part consists of (bottom-up) merging the leafs up-to a unique sorted list. An example of the intermediate steps of this factored version, when applied to the list `[2,4,8,3,1,6,7,5]` can be seen in Figure 1.
If one sees both the construction and the destruction of the tree by levels, one can understand clearly why this is indeed a $O(n \log n)$ sorting algorithm. Each of these steps consists in traversing all n elements of the list.

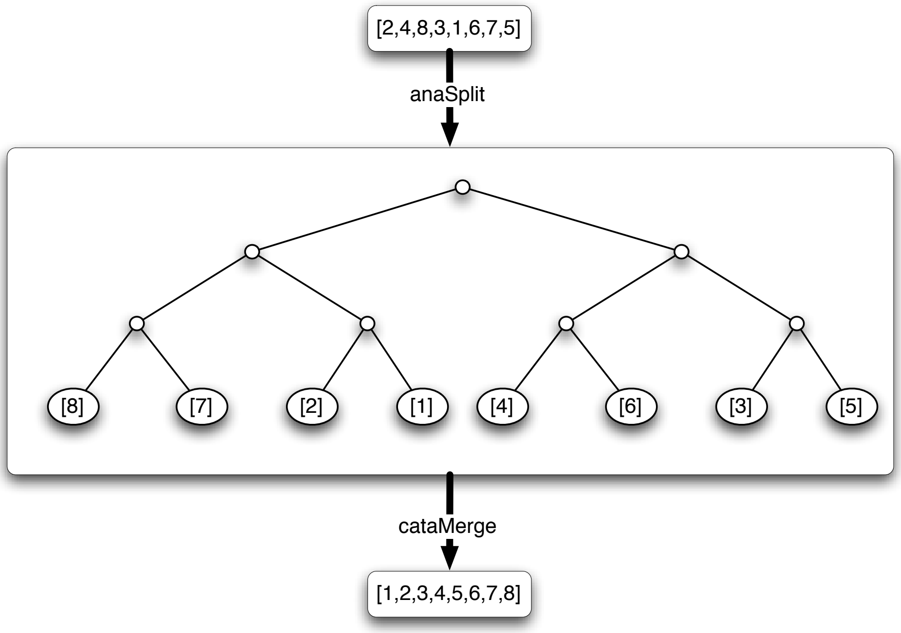

Fig. 1. Original Merge Sort

As there are $\log n$ levels, both the construction and destruction of the tree will be $O(n \log n)$. The sequential composition of two such functions still yields a $O(n \log n)$ function.

One other consequence of this factorisation is that we can now substitute one or both of the factors by other functions that will produce/traverse the same or an equivalent intermediate data structure. We thus transform a program by transforming one of its factors.

We mentioned above that merge sort does not improve its efficiency when the initial list is already sorted or has large chunks of sorted parts. A possible workaround of this problem is to start by pre-processing the list in order to find the already sorted parts of the list. The following function will do exactly that.

```
ascSequences [] = []
ascSequences [x] = [x]
ascSequences (h:t) | h <= hh = ((h:hh:ht):tt
                   | otherwise = [h]:(hh:th):tt
                   where ((hh:th):tt) = ascSequences t
```

This function could be written using the standard Haskell function `groupBy`. We have included this piece of code to show that it is a linear function (in our classification into ana, cata, or hylomorphism, this is clearly a list catamorphism). Note that this function transforms a list into a list of lists and if we are going to use it as a pre-processor to merge sort we will have to slightly change the anamorphism part of the later. This change happens only in the leaf creation part.

```
anaSplit' l | null l      = L []
            | singleton l = L h
            | otherwise   = T (anaSplit' h1) (anaSplit' h2)
           where (h1,h2) = split l
                 [h]     = l
```

Adding up the pieces we can define a new version of merge sort as

```
mergeSort = cataMerge . anaSplit' . ascSequences
```

Figure 2 displays the intermediate structures generated by these factors. As expected, the transformed version will produce a smaller tree, but will have an additional step of grouping the original list.
The question rose by this transformation is how much better (or worse) behaves the resulting function.
Without the explicit factoring of the definition one could be tempted to analyse the performance of this transformed version by expanding the non-singleton leaves into trees. This does not correspond to what happens in practice but could lead us into the misconception that we no longer have a $O(n \log n)$ algorithm. Note that such tree is no longer balanced and therefore we could not reason with that assumption.
However, the same reasoning that was made to the original version of merge sort can still be made about this transformed version.

– The sum of the lengths of all leaves is n (the length of the original list), and this is so for each step of increasing/decreasing one level of the tree.
– The tree has at most $\log n$ levels because it is balanced. As there are fewer nodes in the tree, its depth can be lower than this limit.

There is however an increase in the pre-processing part but, asymptotically, as this pre-processing is linear, it is not relevant. The (sequential) composition of a linear function with a $O(n \log n)$ function yields a $O(n \log n)$ function. This argument can be used to rule out one further transformation – to group the original sequence into (non-consecutive) ascending sub-sequences. Such function is non-linear (quadratic at least on the worst case) thus worsening the overall asymptotic behaviour of merge sort to quadratic.

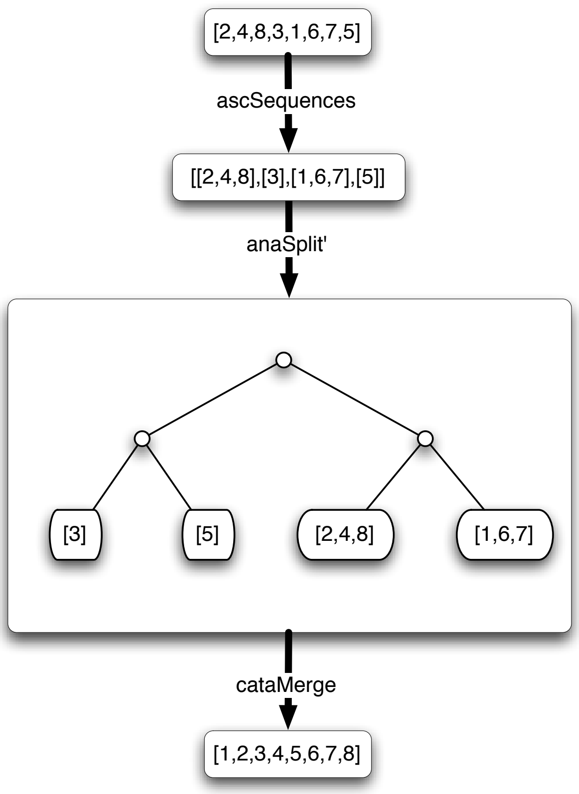

Fig. 2. Transformed Merge Sort

4 Heap sort

Heap sort is another divide and conquer sorting algorithm. Its behaviour on a list with more than one element can be described as:

1. find the minimum of the list and divide the rest into two halves
2. construct the desired result having the minimum as its head, and building the tail by merging the lists corresponding to applying the same method to both halves.

One possible implementation of this algorithm is

```
heapSort l@(f:s:others) = m:(merge (heapSort ll) (heapSort rl))
      where (m,ll,rl) = splitH l
heapSort l = l
```

```
splitH [x] = (x,[],[])
splitH (h:t) | h <= m = (h,m:rl,ll)
             | m < h  = (m,h:rl,ll)
             where (m,ll,rl) = splitH t
```

Applying the factoring algorithm to this definition, the intermediate data structure is a binary tree whose nodes are elements of the original list and the leaves are lists (either empty or singleton).

```
data FT a b = Leaf a | Fork b (FT a b) (FT a b)

heapSort = cataMergeH . anaSplitH

anaSplitH l@(f:s:t) = let (m,l1,l2) = splitH l
                      in Fork m (anaSplitH l1) (anaSplitH l2)
anaSplitH l = Leaf l

cataMergeH (Leaf x) = x
cataMergeH (Fork m e d) = m:(merge (cataMergeH e) (cataMergeH d))
```

Note now that when applied to a (non-empty) list, `splitH` produces a triple `(m,l,r)` where `m:l++r` is a permutation of the original list and

- `m` is the minimum of the list, and
- the lengths of `l` and `r` are equal (up to one).

This means that the function `anaSplitH` produces a binary tree whose root is the minimum element, both its left and right sub-trees have the same number of elements and satisfy these same properties. In other words, the tree produced by this function is a balanced heap whose elements are the elements of the original list.
The first advantage of applying the mentioned factoring algorithm to this definition is that one can easily understand why this is in fact an implementation of heap sort.
An example of the intermediate steps of this factored version, when applied to the list `[2,4,8,3,1,6,7,5]` can be seen in Figure 3.

Both merge sort and heap sort share a common feature: their performance is independent from the initial state of the list. In the previous section we saw how we could easily improve the efficiency of merge sort in the best case without ruining the average performance of the algorithm. This was done by just pre-processing the original list, grouping the contiguous ascending sub-sequences.
This technique could as well be applied to heap sort. We will however present a different method, again based on this factorisation. In this case we will slightly change the intermediate data structure.

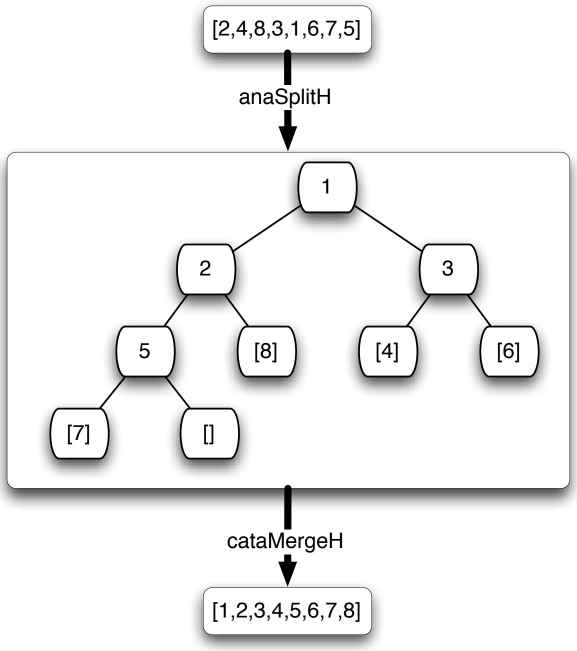

Fig. 3. Original Heap Sort

A heap, as produced by the algorithm above, is a binary tree such that, for every trace `n1`,...,`nk`,`l` we have that the list

`n1:...:nk:l`

is an ordered list.
The transformation that we will present uses a tree whose nodes are lists, and, for every trace `n1`,...,`nk`,`l` we have that the list

`n1++...++nk++l`

is an ordered list.
The catamorphism part of this algorithm is then quite similar to the previous one: we just have to change the `(:)` by concatenation.

```
cataMergeH' (Leaf x) = x
cataMergeH' (Fork m l r) = m ++ (merge (cataMergeH' l) (cataMergeH' r))
```

The split function in the original function computed a triple `(m,l,r)` being `m` the minimum of the list Now will also have to compute a triple, but its first element should be a (sorted) list of elements less or equal to all other elements. A possible definition is then

```
anaSplitH' l@(f:s:t) = let (m,e,d) = splitH' l
                       in Fork m (anaSplitH' e) (anaSplitH' d)
anaSplitH' l = Leaf l

splitH' [x] = ([x],[],[])
splitH' (h:t) | h <= m = (h:m:mm,l1,l2)
              | m < h  = ([m],l3,l4)
   where ((m:mm),l1,l2) = splitH' t
         (l3,l4) = add mm (h:l2,l1)
         add [] p = p
         add (x:xx) p = let (a,b) = add xx p
                        in (x:b,a)
```

Figure 4 displays the intermediate structures generated by these factors. As expected, the transformed version will produce a smaller tree. Note that the modifications that we have made, both on the catamorphism and on the anamorphism, guarantee that we keep the asymptotic behaviour of this algorithm unchanged. The function `splitH'` is still linear on the size of its input, although its worst case is worse that `splitH`.
In the best case, as happened in the transformation of merge sort, we obtain a linear function.
We are now able to perform one last step in transforming the original heap sort algorithm – to put together both the improvements that we made in merge sort (pre-processing the original list) and in heap sort (storing lists of elements in the nodes of the heap). We will only need to redefine the anamorphism of heap sort.

```
splitHH :: (Ord a) => [[a]] -> ([a],[[a]],[[a]])
splitHH [x] = (x,[],[])
splitHH (h:t) = (m',m'':r,l)
  where (m,l,r) = splitHH t
        (m',m'') = combine m h

combine (m:mm) (h:hh) | h <= m = let (a,b) = combine (m:mm) hh
                                 in (h:a,b)
                     | otherwise = let (a,b) = combine mm (h:hh)
                                   in (m:a,b)
combine a b = (a,b)

anaSplitHH l@(f:s:t) = let (m,e,d) = splitHH l
                       in Fork m (anaSplitHH e) (anaSplitHH d)
anaSplitHH [l] = Leaf l
anaSplitHH [] = Leaf []
```

Note again that as `splitHH` is linear on the size of its argument, and both sublists have aproximate lengths, `anaSplitHH` is still a $O(n \log n)$ function.

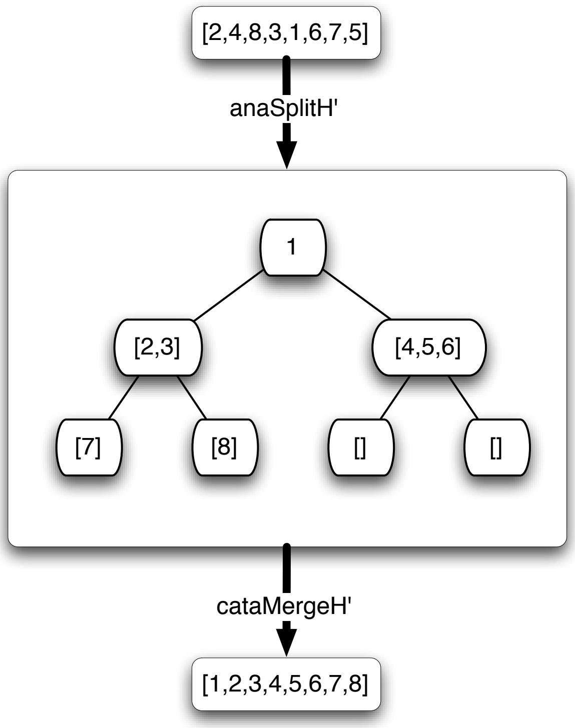

Fig. 4. Transformed Heap Sort

The definition of heap sort becomes:

```
heapsort :: (Ord a) => [a] -> [a]
heapsort = cataMergeH' . anaSplitHH . groupasc
```

Figure 5 displays the intermediate structures generated by these factors when applied to the list `[2,4,8,3,1,6,7,5]` .
Note that some of the work that was previously performed by the catamorphism, namely the merging of the ascending sequences is now performed by the anamorphism.

5 Discussion

We have shown through two small examples how forestation can be used as a program understanding tool, providing insights into the algorithm underlying a particular function definition. This has helped us transform

the definitions of two well known sorting algorithms into similar ones that take advantage of the portions of the original list which are already sorted. We have also shown how performance aspects of these definitions can be taken into account under this perspective.
We have also tried to exemplify how an algorithm (when seen as a strategy to solve a particular task) can be described by the characterisation of the internal data structures. Another such example is quicksort, where the internal data structure is a search tree. Expliciting the construction of this tree one can easily understand why this algorithm has a worst case asymptotic performance of $O(n^2)$ in the case where the original list is sorted in reverse order.
The ideas presented here came, as often happens, from an educational experience. When trying to teach topics in the area of generic programming, namely recursion patterns such as ana, cata and hylomorphisms, one usually tries to express as such, some previously studied functions. This has led to the formalisation of a very simple algorithm to factor explicitly recursive definitions into an hylomorphism, i.e., the composition of a producer followed by a consumer of some tree-like structure. We have found out that this is a successful strategy to teach programming issues and have been using it in different settings, namely in our Logic classes where different procedures (like tableaux, or the Davis-Putnam) are taught (see [2] for a case study in this area).
The method proposed here is not syntax driven and therefore is not intended for automatisation. We feel however that even such syntax oriented methods, like Burstall and Darlington's fold/unfold can be coded in this setting. Going back to the diagrams of page 6, the left-to-right arrow corresponds to the unfold part, whereas the right-to left corresponds to the folding part.

References

1. Lex Augusteijn. Sorting morphisms. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, 1999.
2. L. S. Barbosa, J. B. Barros, and J. J. Almeida. Polytypic recursion patterns. In *ENTCS*. Elsevier, 2000. Proceedings, Simpósio Brasileiro de Linguagens de Programação, UFEP, Recife, May 2000.
3. R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–76, January 1977.
4. Alcino Cunha. Automatic visualization of recursion trees: a case study on generic programming. *Electronic Notes in Theoretical Computer Science*, 86(3), 2003. Selected papers of the 12th International Workshop on Functional and (Constraint) Logic Programming.

5. Alcino Cunha, José Barros, and João Saraiva. Deriving animations from recursive definitions. In *Draft Proceedings of the 14th International Workshop on the Implementation of Functional Languages (IFL'02)*, 2002.
6. Maarten Fokkinga. *Law and Order in Algorithms.* PhD thesis, University of Twente, 1992.
7. Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 73–82. ACM Press, 1996.
8. Simon Peyton Jones and John Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language.* February 1999.
9. Carroll Morgan. *Programming from Specifications.* Prentice-Hall, 1990.
10. Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In *Proceedings of the IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman & Hall, 1997.
11. Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming*, number 300 in LNCS, pages 344–358. Springer-Verlag, 1988.
12. Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

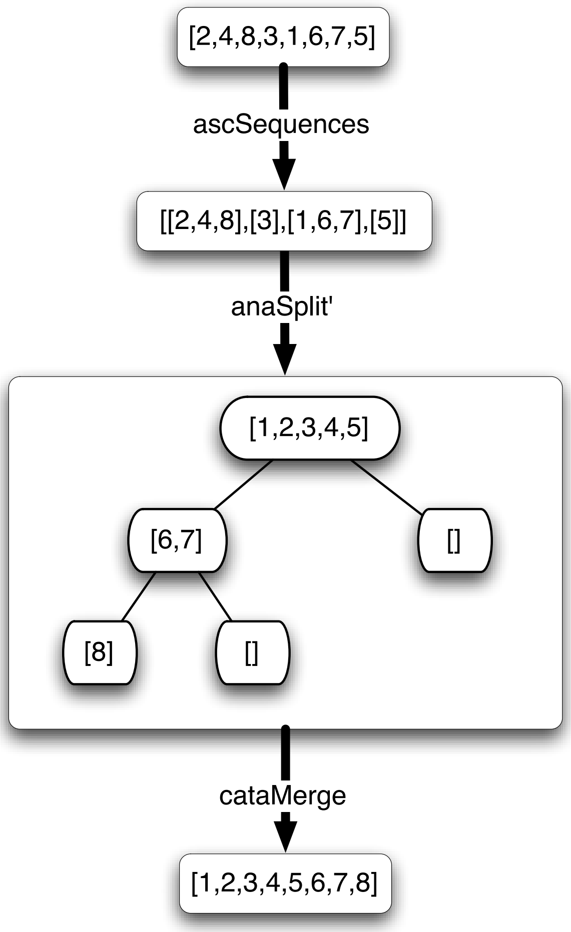

Fig. 5. Transformed Heap Sort