Recursion Patterns and Time-analysis

Manuel Barbosa, Alcino Cunha, and Jorge Sousa Pinto

Departamento de Informática, Universidade do Minho
Campus de Gualtar, 4710 Braga, Portugal
{mbb,mac,jsp}@di.uminho.pt

Abstract. This paper explores some ideas concerning the time-analysis of functional programs defined by instantiating typical recursion patterns such as *folds*, *unfolds*, and *hylomorphisms*. The concepts in this paper are illustrated through a rich set of examples in the Haskell programming language. We concentrate on unfolds and folds (also known as anamorphisms and catamorphisms respectively) of recursively defined types, as well as the more general *hylomorphism* pattern. For the latter, we use as case-studies two famous sorting algorithms, mergesort and quicksort. Even though time analysis is not compositional, we argue that splitting functions to expose the explicit construction of the recursion tree and its later consumption helps with this analysis.

1 Introduction

Over the years, a tradition has been established in the functional programming and mathematics of program construction communities of using *recursion patterns* as a programming tool. The idea is that, since many functions that manipulate recursive types are alike, one can write once and for all a parameterized function that encapsulates their basic structure. Instantiating the parameters then allows to write functions without using explicit recursion.

This approach to programming is also often presented as being a powerful tool for *program understanding*, since it allows to explicitly construct intermediate data-structures whose observation may tell us something about the programs. Another benefit is *modularity*, since new algorithms can be designed by simply plugging functions that construct intermediate structures with others that consume them. We have not, however, come across a study of the implications of this approach for the complexity of the resulting programs and its analysis. In this paper we present a first contribution in this direction.

We give examples of analysing the asymptotic execution time of Haskell functions defined with recursion patterns, and show that a mechanized analysis relating the complexity of the defined functions with the complexity of its genes is difficult to obtain. For hylomorphisms, we show that an analysis based on the intermediate structures may in certain situations be performed in a compositional way, since the worst case of the functions that unfold the recursion tree and then fold over it to obtain the final result is typically related to the shape of those structures.

We also believe that this approach to the analysis of execution time has educational benefits. In particular, we have been teaching execution time analysis of the sorting algorithms presented here (in an imperative setting) to students who have frequented a course on program calculation, and find that acquaintance with intermediate data-structures can be extremely helpful. For algorithms seen as hylomorphisms, half of the analysis work is already done.

Organization of the Paper. Section 2 contains the necessary background material on recursion patterns. In section 3 two well-known sorting algorithms are examined in the light of the previous section, and redefined using appropriate recursion patterns. Sections 4 and 5 investigate the time analysis of recursion patterns, and how this relates to the complexity of the respective genes: first some basic ideas are exposed, then applied to the sorting algorithms of section 3. Finally, we conclude the paper in section 6.

2 Recursion Patterns in a Nutshell

Folds. A classic example of a situation where a recursive pattern can be used is given by any function that iterates an arithmetic operator over a list of numbers. Functions such as these, where the result for the recursive case is given as a function of the head of the list and the result of applying recursively the function to the tail, are called *folds* over lists, and can be written alternatively using the standard Haskell function `foldr`, defined as:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

For instance, the function that sums the elements in a list is written as `sum = foldr (+) 0`.

This idea can be generalized for any regular type, the difference being that the programmer will have to write the appropriate parameterized function that encapsulates the recursion pattern. We exemplify this for two different types of binary trees, respectively leaf-labeled and node-labeled, defined as follows.

```
data LTree a = Leaf a | Fork (LTree a) (LTree a)
data BTree a = Nil | Node a (BTree a) (BTree a)
```

Folds over these trees can be captured by the functions `cataLTree` and `cataBTree`. In order to define these functions, one just has to bear in mind that when processing a node the results of performing recursion on the sub-trees of the node are combined with the contents of the node (if any) to give the final result.

```
cataLTree :: (b -> b -> b) -> (a->b) -> LTree a -> b
cataLTree f g (Leaf x)   = g x
cataLTree f g (Fork l r) = f l' r' where l' = cataLTree f g l
                                         r' = cataLTree f g r

cataBTree :: (a -> b -> b -> b) -> b -> BTree a -> b
cataBTree f z Nil = z
cataBTeee f z (Node x l r) = f x l' r' where l' = cataBTeee f z l
                                             r' = cataBTeee f z r
```

Often the arguments of these functions are called the *genes* of the folds. In the case of binary trees, the genes are pairs of functions, the first of which is used to process nodes, and the second to process leaves (in fact, for node-labeled trees, this second component is a constant since there is no information to process in an empty leaf).

As examples of folds over trees, let us write the height functions for both types of trees.

```
heightLTree = cataLTree (\ l r -> 1 + (max l r)) (\_ -> 0)
heightBTree = cataBTree (\x l r -> 1 + (max l r)) 0
```

Unfolds. In the context of a programming language with non-strict semantics such as Haskell, recursive and co-recursive types coincide. Unfolds stand to corecursive programs as folds stand to recursive programs (they are dual concepts); they are however less widely used by programmers [4]. Consider the Haskell function that constructs (when invoked with 0) a binary tree where each node is labeled with its depth in the tree, stopping at level 100.

```
inflevels100 100 = Nil
inflevels100 n   = Node n (inflevels100 (n+1)) (inflevels100 (n+1))
```

This function has the property that when a node is returned, both its sub-trees are recursively constructed by the same function being defined. This clearly indicates that a corecursion pattern is being used, which can be written once and for all. The following functions capture these patterns for both types of trees we have been using:

```
anaLTree :: (b -> (Either a (b,b))) -> b -> LTree a
anaLTree h x = case (h x) of Left y      -> Leaf y
                             Right (a,b) -> Fork (anaLTree h a) (anaLTree h b)

anaBTree :: (a -> Maybe (b,a,a)) -> a -> BTree b
anaBTree h x = case (h x) of Nothing      -> Nil
                             Just (y,a,b) -> Node y (anaBTree h a) (anaBTree h b)
```

Note that in the latter definition it would have been equivalent to use the type `Either () (b,a,a)` instead of `Maybe (b,a,a)`. We can now write `inflevels100` without using explicit recursion:

```
inflevels100 = anaBTree h where h 100 = Nothing
                                h n   = Just(n,n+1,n+1)
```

The Generic Perspective: Catamorphisms, Anamorphisms, and Paramorphisms. Although this is clearly outside the scope of the present paper, we remark that the names *catamorphism* and *anamorphism* used for folds and unfolds respectively, come from the development of a unified theory capable of handling all the useful inductive types: there is a single definition of a catamorphism, which is parameterized on its domain recursive type, and a single definition of anamorphism, parameterized on its codomain recursive type [5]. Moreover, the properties of catamorphisms and anamorphisms are valid for all instantiations of the base functors and do not need to be proved independently for each new type one defines.

It should also be pointed out that many other recursion patterns have been studied in the same setting; it is easy to see that many interesting recursive functions cannot be captured by catamorphisms. Take for instance the function that inserts a number in a (node-labeled) binary tree where the elements are sorted *inorder*:

```
insert x Nil          = Node x Nil Nil
insert x (Node y l r) = if x<=y then Node y (insert x l) r
                                else Node y l (insert x r)
```

The function `insert x :: BTree Int -> BTree Int` is not a catamorphism since in the recursive case one of the sub-trees must be used unchanged (and not the result of invoking the function on it); in other words we are here using primitive recursion rather than iterating over the tree. Such functions are called *paramorphisms* [6].

Hylomorphisms. Paramorphisms are more powerful than catamorphisms, but still incapable of capturing the behaviour of many functions: it suffices to see that recursion must follow the structural definition of the type. Take the following recursive clause in the definition of the mergesort algorithm, studied in the next section.

```
ms l = let (a,b) = ms_split l in ms_merge (ms a) (ms b)
```

Recursion here is performed twice, on lists obtained somehow from the initial argument – clearly not a structural form of recursion. The good news here is that most useful functions can be written as the composition of a catamorphism and an anamorphism of some inductive type. The anamorphism first builds an intermediate data-structure reflecting the shape of the recursion tree (so in the above case it would take a list and build a binary tree); the catamorphism then iterates over this tree to give the final result. This composite recursion pattern is called a *hylomorphism* [7]. A property of hylomorphisms is that the intermediate data-structures can be eliminated: a function encapsulating this pattern can be defined with explicit recursion, without using catamorphisms and anamorphisms.

The next section presents examples of hylomorphisms of type `[a] -> [a]`, where the intermediate data-structures are binary trees. These examples will hopefully help clarifying the ideas in this section.

3 Sorting Algorithms as Hylomorphisms

To exemplify the factorization of functions as hylomorphisms, in this section we will be looking at two well-known sorting algorithms: *Mergesort* and *Quicksort*. We basically follow [1], and refer the reader to that paper for a full development of these ideas.

Mergesort. This algorithm can be implemented in Haskell as follows:

```
ms []  = []
ms [x] = [x]
ms l   =  ms_merge (ms a) (ms b)   where (a,b) = ms_split l
```

```
ms_split []    = ([],[])
ms_split (h:t) = let (a,b) = ms_split t in (h:b,a)

ms_merge l []            = l
ms_merge [] l            = l
ms_merge (ha:ta) (hb:tb) = if ha < hb  then ha:(ms_merge ta (hb:tb))
                                       else hb:(ms_merge (ha:ta) tb)
```

This implementation follows the standard divide-and-conquer definition of the mergesort algorithm. The unordered list is split into two halves, which are sorted recursively. The resulting ordered lists are then combined to form a complete sorted version of the input list. Note the way in which the splitting of the unordered list is made, due to the impossibility of accessing list elements randomly.

Insight on how to transform a recursive implementation such as this into a hylomorphism comes from analysing its recursion tree. An example of such a tree is shown in figure 1.

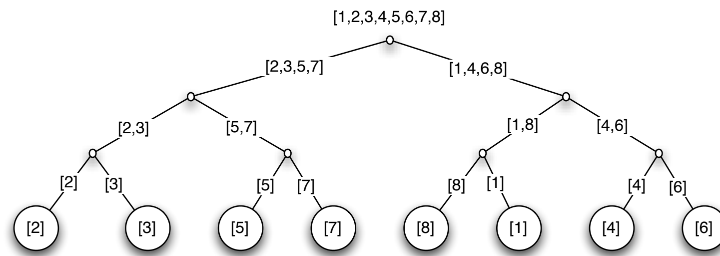

Fig. 1. Mergesort recursion tree for list [2,8,5,4,3,1,7,6], annotated with the result of each recursive call

The recursion tree of this algorithm can be represented as a binary leaf tree, since in mergesort no data is associated with intermediate levels in the recursion tree. All input list elements are included in the split lists that are passed to lower levels in the recursion tree. Similarly, all the data that is required for the combination of the partial solutions are the sorted sub-lists themselves. Recursion stops when the results of splitting are lists with a single element, which correspond to the leaves of the recursion tree.

A direct way of implementing mergesort as a hylomorphism is to use the type `LTree` introduced in section 2 for the intermediate data structure, which will thus be a replica of the recursion tree. A suitable anamorphism will then have to generate this data structure from the input list. The anamorphisms for this type of tree can be generated by the function `anaLTree` of section 2; in particular for the mergesort anamorphism the following function can be used as gene.

```
f [x] = Left x
f lst = Right (ms_split lst)
```

This function generates leaves for the tree whenever a list with one element is encountered, and otherwise it produces a pair of lists that will be used to generate two new tree branches.

Similarly, a suitable catamorphism will have to consume the intermediate data structure and generate the sorted output list. Catamorphisms for this type of tree have pairs of functions as genes, and can be generated by function `cataLTree` that takes two functions as arguments. For the mergesort catamorphism we have:

– The function `ms_merge` defined above may be used as the first argument in `cataLTree`: it takes two sorted lists which are the results of the recursion over the two tree branches, and combines them into a sorted list.
– The second argument is a function that generates an initial result when a leaf is consumed. In the particular case of mergesort, this function must simply construct a list with a single element using the value stored in the leaf. We use the function `wrap = \x -> [x]`.

The complete implementation of mergesort as a hylomorphism is given below:

```
hyloMs = (cataLTree ms_merge wrap)  . (anaLTree f)
```

The reader will have noticed that there is a slight mismatch between the initial definition of mergesort and this hylomorphism, since the latter is not defined on the empty list. This can be easily solved by using a `Maybe` type for the leaves; we will however stick to this partial version for the sake of simplicity.

Quicksort. In order to find out what the catamorphism and anamorphism functions will be like for quicksort, it is first necessary to identify a suitable intermediate data structure, as was done for mergesort. The implementation of quicksort in Haskell is straightforward, following the classic definition of the algorithm:

```
qs []    = []
qs (h:t) = let (a,b) = qs_split h t
           in (qs a) ++ h:(qs b)

qs_split _ []    = ([],[])
qs_split p (h:t) = let (a,b) = qs_split p t in if h<=p then (h:a,b) else (a,h:b)
```

The unsorted input list is split into two smaller lists based on the value stored at the head of the list, which operates as pivot. The lists resulting from the splitting contain the elements which are greater than (resp. smaller than or equal to) the pivot. These lists are then sorted recursively, and the results are combined through concatenation to create a complete sorted output list.

In each quicksort invocation an element of the input list (the pivot) is not passed down to the recursive calls, which means that there will be data associated with each node in the recursion tree. Furthermore, the fact that the stop condition is the empty list means that the leaves of the recursion tree will contain no data. The appropriate type to use for the intermediate data-structure is thus a node-labeled binary tree, as exemplified in figure 2.

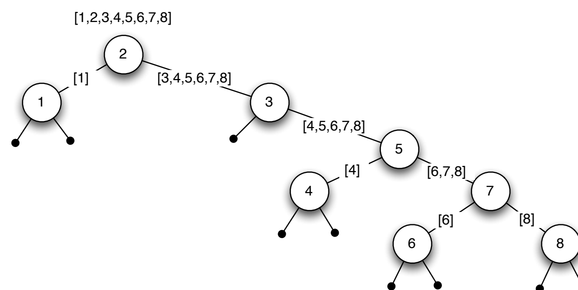

Fig. 2. Quicksort recursion tree for list [2,3,5,7,8,1,4,6]

As was the case for mergesort, the anamorphism for the quicksort algorithm will have to generate such a tree from the unsorted input list. The catamorphism will have to consume this result and generate an ordered list. These functions will now be written by providing the appropriate gene arguments to `anaBTree` and `cataBTree` respectively (see section 2).

The following function can be used as a gene in `anaBTree`. It returns `Nothing` if an empty list is found; otherwise, it selects a pivot and uses it to generate a triplet containing the pivot itself and the two lists resulting from splitting the input list based on the value of the pivot. Note that this triplet contains all the information required to construct a new tree node.

```
f []    = Nothing
f (h:t) = Just (h,l,r)   where (l,r) = qs_split h t
```

As for the catamorphism, the gene function will associate the empty list with the consumption of the empty tree leaves, and will combine the value at each node (the pivot) with the results of recursively processing the two tree branches in the node. A possible complete implementation of quicksort as a hylomorphism is then:

```
hyloQs = (cataBTree g []) . (anaBTree f)
        where g x a b = a ++ (x:b)
```

Notice that, for most recursive definitions, notably the ones presented in this section, the hylomorphisms that implement them can be derived automatically [?].

4 Analysis of Recursion Patterns

In this section we take a first look at the execution time analysis of recursion patterns. Our goal is to evaluate the asymptotic execution time of functions and express it using the standard $\mathcal{O}$, Ω and Θ notations (see for instance [2]), in order to understand the relation between the complexity of functions defined with recursion patterns and the complexity of their genes.

Before doing this however, we consider briefly the problem of compositionality of time analysis. Our claim is that programs can be written as combinations of recursion patterns. In principle, analysing the complexity of smaller components is an easier task than analysing the entire program, which suggests a compositional approach to complexity analysis. This approach is however not straightforward, even for rather simple cases.

Functions can be combined in several ways, such as composition or application. In both cases, it may be rather tricky to assess the asymptotic behaviour of the combined function, given the asymptotic behaviour of its components. Consider for example the following functions that sort a list twice in a row (pointless but useful for showing why complexity analysis is not compositional).

```
sort_twice_1 = merge_sort . insertion_sort
sort_twice_2 = insertion_sort . merge_sort
```

It is well-known that mergesort executes in $\Theta(n \lg n)$, for any configuration of the input list. On the other hand, insertion sort has a best-case behaviour which is in $\Theta(n)$, when the input list is already sorted, and a worst-case behaviour in $\Theta(n^2)$. One might be tempted to guess that the worst-case behaviour of the two functions above is in $\Theta(n^2)$, however this is true only for the first function. In fact, for the second version of the function, insertion sort is always processing an already sorted list, its best-case behaviour input. Thus `sort_twice_2` has worst-case behaviour in $\Theta(n \lg n)$.

This shows that it is only in particular situations that it is possible to apply compositional analysis, and for that we need to keep track of how the input to the functions being analysed affects their behaviour. This will be fundamental for the analysis of hylomorphisms in the next section.

We now turn to the time analysis of folds over lists. Consider the following implementation of a function that reverses a list:

```
reverse = foldr append_end []
append_end x = foldr (:) [x]
```

Let us first analyse the gene of `append_end x`. The constructor `(:)` runs in constant time and always uses its second argument, which is very important for our analysis, since this means that `append_end` performs a full traversal of the argument list. This traversal adds constant cost at each position, and constant cost at the end (the cost of constructing `[x]` from `x`); it thus runs in time $\Theta(n)$.

To see how the use of the second argument is relevant for this simple reasoning, it suffices to consider the following definition of a function that returns the head of a list in constant time (assuming non-strict semantics).

```
head = foldr (\x y -> x) undefined
```

A similar analysis can now be made for the `reverse` function. `append_end`, which also uses its second argument, runs in time $\Theta(i)$, with i the size of this argument (the result of a recursive call). Since this function will be called exactly n times, with i varying from 0 to $n-1$, the execution time of `reverse` can be written as $T(n) = \sum_{i=0}^{n-1} \Theta(i) = \Theta(n^2)$.

This example seems to indicate that this kind of analysis can be mechanized, with the help of a program analyser capable of identifying the use of variables in functions. We will however show in the next section that this will no longer be the case when the intermediate structures are not linear.

5 Analysis of Binary Tree Hylomorphisms: Sorting Algorithms

The execution times of quicksort and mergesort are well-known. Quicksort has a worst-case execution time in $\Theta(n^2)$, and a best-case execution time in $\Theta(n \lg n)$. Its average-case execution time is closer to the latter. For mergesort, the worst-case and best-case execution times are both in $\Theta(n \lg n)$. Although these execution times are usually associated with implementations using imperative languages, where the sorting algorithms operate on random access arrays, they also apply to the functional implementations with lists.

It is fairly simple to perform these analyses using classic techniques such as recursion trees and recurrences. In this section we do exactly this, except that we reach the same results using the hylomorphism versions of the algorithms, i.e, we analyse separately the anamorphism and catamorphism components, and then combine the results.

The points we want to make here are first that by analysing the asymptotic behaviour of the anamorphism and catamorphism functions it is in certain conditions possible to obtain the asymptotic behaviour of the hylomorphism. Moreover, this analysis may even be more convenient than a standard analysis of the initial (i.e. before factorization) sorting algorithms, since the intermediate structures, whose shape is extremely important for the analysis, are now constructed explicitly.

Secondly, when the recursive types involved are trees, it is no longer obvious to analyse recursion patterns mechanically based on the asymptotic behaviour of their genes, due to the role played by the *shape* of the trees on the execution.

Quicksort. Let us first analyse the execution time of the two components in the hylomorphism implementation given in the previous section.

The anamorphism, the function generating the intermediate data structure from the input list, has an execution time which is described by the following recurrence:

$$\begin{cases} T(1) = \Theta(1) \\ T(n) = T(k) + T(n-1-k) + \Theta(n) \end{cases} \tag{1}$$

This recurrence expresses the workload associated with building the tree:

- For empty lists, the constant amount of work of constructing an empty tree.
- For non-empty lists, before constructing the tree node, it is necessary to split the input list into two parts and then build the two corresponding sub-trees. k represents the size of one of the sub-lists, which will depend on the value of the pivot.

The execution time of this function will very much depend on the values of the elements in the input list: a worst-case situation will occur when the input list is already sorted or in reverse order. In this case, the splitting will produce a sub-list with $k = n-1$ elements (the other one will be empty). The resulting tree will grow only in the right branch in the form of a list (see right hand side of figure 3). In this situation the second equation in the recurrence becomes simply

$$T(n) = T(n-1) + \Theta(n) \tag{2}$$

leading to a worst-case execution time in $\Theta(n^2)$. On the other hand, the best-case execution (left hand side of figure 3) will occur when splits are consistently balanced, i.e. when the size of the sub-lists at each recursion level are roughly half the size of the input list.

In this type of situation the recurrence becomes

$$T(n) = T(\lfloor (n-1)/2 \rfloor) + T(\lceil (n-1)/2 \rceil) + \Theta(n) \tag{3}$$

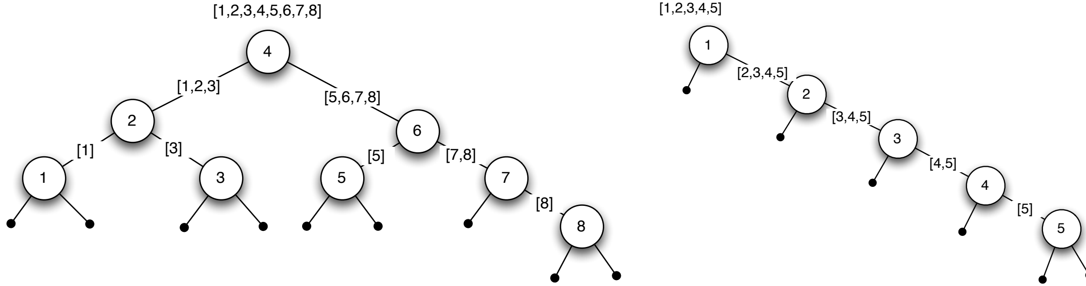

Fig. 3. Quicksort recursion tree for lists [4,2,1,3,6,5,7,8] (left) and [1,2,3,4,5] (right)

which leads to a best-case execution time in $\Theta(n \lg n)$.

Let us now turn our attention to the catamorphism of the quicksort algorithm, that traverses the intermediate data structure and generates an ordered output list. The execution time of the catamorphism can therefore be described by the recurrence

$$\begin{cases} T(1) = \Theta(1) \\ T(n) = T(k) + T(n-1-k) + \Theta(k) \end{cases} \tag{4}$$

where k is the number of elements in the left sub-tree. This recurrence accounts for the work involved in converting the tree into a sorted list:

– For a tree containing only one element, the only thing that is required is to construct a list with one element.
– For trees with more elements, it is necessary to generate the sorted lists containing the elements in both sub-trees and then to combine them into a single sorted list. This last task has an execution time which is linear in the number of elements in the left sub-tree, since it appends the two lists (with the node value in the middle) and the time cost of the append operation is linear in the size of its first argument, the left sub-list.

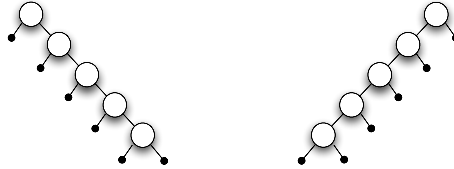

Fig. 4. Best case and worst case input trees for Quicksort catamorphism

The best-case execution time for this function will occur when all the nodes in the input tree contain no left-subtree i.e. when all the nodes are on the right spine of the tree (figure 4). In this case, there is only a constant amount of work to be carried out at each node, since the left sub-tree is empty. Therefore, the execution time will be a linear function of the number of tree elements.

The worst-case is symmetric to the previous one: it occurs when all the nodes in the input tree contain no right sub-tree (see also figure 4). In this case, at each tree level, it is necessary to go through all the elements which have been previously processed in order to append the current element to the end of the sorted list. This leads to a quadratic variation of the execution time of this function with respect to the number of tree elements.

One case still remains to be explored, corresponding to the asymptotic behaviour of the catamorphism for a fairly balanced intermediate tree. In this case, the recurrence for the catamorphism becomes

$$T(n) \approx 2T((n-1)/2) + \Theta((n-1)/2) \tag{5}$$

This points to (note the similarity with recurrence 3) an asymptotic behaviour identical to that of the anamorphism: $\Theta(n \lg n)$. Table 1 summarizes the results of the time analysis of the quicksort anamorphism and catamorphism. It

Input list ordering	Tree Shape	Anamorphism	Catamorphism	Hylomorphism
Decreasing	/	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Increasing	\	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$
Random	△	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$

Table 1. Time analysis of the quicksort hylomorphism

is now straightforward to analyse the time behaviour of the hylomorphism for each of the relevant cases, as shown in the last column of table 1: for each input the asymptotic time of the composition is the sum of the asymptotic times of the component functions, thus the function with the worst performance determines the end-result.

Mergesort. The anamorphism and catamorphism functions for this algorithm operate in a very similar manner to quicksort. The catamorphism behaves almost exactly the same as for quicksort, the difference being that the execution time of its gene is linear with respect to the size of the resulting merged list. This means that its worst-case behaviour is also in $\Theta(n^2)$, but this occurs both for sorted and reversed input lists. The best-case behaviour now occurs when the intermediate tree is balanced, and it is in $\Theta(n \lg n)$. This difference on the catamorphism side does not affect the analysis of the algorithm significantly.

However, there is a very important difference on the anamorphism side, in terms of asymptotic behaviour: the input list splitting process always produces sub-lists of similar sizes. This means that the intermediate tree that is produced is always fairly balanced. In terms of execution time, this places the anamorphism function in $\Theta(n \lg n)$ for all input list configurations, since the total work needed for constructing each tree level is in $\Theta(n)$ (a full traversal of all the sublists obtained from the initial) and the tree has $\lg n$ levels. Table 2 summarises the execution time

Input list ordering	Tree Shape	Anamorphism	Catamorphism	Hylomorphism
Any	△	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$
–	/	–	$\Theta(n^2)$	–
–	\	–	$\Theta(n^2)$	–

Table 2. Time analysis of the mergesort hylomorphism

characteristics of the mergesort anamorphism, catamorphism and hylomorphism for different input list orderings. We observe that despite the fact that the catamorphism function has a worst-case execution time of $\Theta(n^2)$, this has no influence on the asymptotic behaviour of the mergesort hylomorphism. This is because the anamorphism never produces a tree that leads the catamorphism into its worst-case behaviour.

Analysis of Recursion Patterns, Revisited. The examples in this section clearly show that when dealing with trees rather than lists, the conclusions of section 4 regarding the execution time of functions of the form `foldr f c` must be reassessed.

It is clear from both the quicksort and mergesort examples that the complexity of the catamorphisms over binary trees cannot be calculated mechanically from the complexity of the respective genes. The quicksort catamorphism is linear on the length of its first argument and the mergesort catamorphism is linear on the sum of the lengths of its two arguments; in both cases the *shape* of the tree dictates the behaviour of the function.

In what concerns the time-analysis of the hylomorphisms, both examples are surprisingly simple, since they correspond to situations where function composition is easy to analyse: for quicksort, the worst cases coincide, and for mergesort the anamorphism generates a single tree shape, which means that the evaluation context of the catamorphism is such that its worst-case behaviour is never possible.

6 Conclusions and Future Work

For all the examples in this paper it was not relevant whether the underlying semantics was strict or non-strict. Consider however the following function that computes the smallest element in a list.

```
first = head . sort
```

where `sort` is some sorting function (maybe mergesort or quicksort). In the presence of a non-strict semantics (such as in Haskell), `first` may (depending on the sorting algorithm) do much less work than `sort`, and the methods presented in this paper cannot help in analysing such situations. The problem is that the evaluation context of the sorting algorithm has changed, and it is no longer sufficient to analyse the genes and the shape of the data-structures that may lead to worst-case analysis.

David Sands has proposed methods based on *cost functions* [8], i.e., functions that receive the same input as the functions being analysed and return a measure of the execution cost of the function for that input. It would be interesting to see how such methods apply to recursion patterns. On the other hand, the above definition is a classic example of a situation where *program calculation* can be applied; in particular, since sort is either a fold or a hylomorphism, a *fusion* law can be applied to `head`, to convert `first` into a new fold or hylomorphism. Fusion laws allow to *deforest* programs by eliminating intermediate data-structures; a study of the consequences of applying fusion regarding the complexity of the functions involved is also an interesting subject to explore.

Another interesting topic concerns the verification of invariant properties of the intermediate data-structures. It has been made clear in this paper that such properties (notably *balancing* properties) play a major role in the time behaviour of hylomorphisms. Since these structures are constructed by corecursive functions, they cannot be proved by induction [3]. However, other methods may be applicable, allowing to identify the conditions that the genes must verify in order for the constructed structures to meet a particular property. Isolating the construction of the intermediate trees in separate functions may help in proving formally properties of these structures.

Acknowledgments

This research was partially supported by FCT research project Pure POSI/CHS/44304/2002.

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. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
3. J. Gibbons and G. Hutton. Proof methods for structured corecursive programs. In *Proceedings of the 1st Scottish Functional Programming Workshop*, 1999.
4. Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 273–279. ACM Press, 1998.
5. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, October 1990.
6. Lambert Meertens. Paramorphisms. Technical Report RUU-CS-90-4, Utrecht University, Department of Computer Science, January 1990.
7. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.
8. David Sands. A naive time analysis and its theory of cost equivalence. *Journal of Logic and Computation*, 5(4):495–541, 1995.