Sorting morphisms

Lex Augusteijn

Philips Research Laboratories, Eindhoven
Email: `lex@natlab.research.philips.com`

Abstract. Sorting algorithms can be classified in many different ways. The way presented here is by expressing the algorithms as functional programs and to classify them by means of their *recursion patterns*. These patterns on their turn can be classified as the natural recursion patterns that destruct or construct a given data-type, the so called *cata-* and *anamorphisms* respectively. We show that the selection of the recursion pattern can be seen as the major design decision, in most cases leaving no more room for more decisions in the design of the sorting algorithm. It is also shown that the use of alternative data structures may lead to new sorting algorithms.
This presentation also serves as a gentle, light-weight, introduction into the various morphisms.

1 Introduction

In this paper we present several well known sorting algorithms, namely *insertion sort*, *straight selection sort*, *bubble sort*, *quick sort*, *heap sort* and *merge sort* (see e.g. [Knu73, Wir76]) in a non-standard way. We express the sorting algorithms as functional programs that obey a certain pattern of recursion. We show that for each of the sorting algorithms, the recursion patterns forms the major design decision, often leaving no more space for additional decisions to be taken. We make these recursion patterns explicit in the form of higher-order functions, much like the well-known `map` function on lists.
In order to reason about recursion patterns, we need to formalize that notion. Such a formalization is already available, based on a category theoretical modeling of recursive data types as can e.g. be found in [Fok92, Mei92]. In [BdM94] this theory is presented together with its application to many algorithms, including selection sort and quicksort. These algorithms can be understood however only after absorbing the underlying category theory. There is no need to present that theory here. The results that we need can be understood by anyone having some basic knowledge of functional programming, hence we repeat only the main results here. These results show how to each recursive data type a number of morphisms is related, each capturing some pattern of recursion which involve the recursive structure of the data type. Of these morphisms, we use the so called *catamorphism*, *anamorphism*, *hylomorphism* and *paramorphism* on linear lists and binary trees. The value of this approach is not so much in obtaining a nice implementation of some algorithm, but in unraveling its structure.
This presentation gives the opportunity to introduce the various morphisms in a simple way, namely as patterns of recursion that are useful in functional programming, instead

of the usual approach via category theory, which tends to be needlessly intimidating for the average programmer.
In this paper, we assume that all sorting operations transform a list `l` into a list `s` that is a permutation of `l`, such that the elements of `s` are ascending w.r.t. to a total ordering relation `<`. Moreover, we assume the existence of an equivalence relation `==` on the elements, such that for all elements `a`, `b`, either `a<b`, `a==b` or `b<a`.
We express the sorting algorithms in the functional language Gofer [Jon95], which is a dialect of Haskell [HWA+92]. We assume that the reader is familiar with, but not necessarily an expert in, functional programming.
This paper is organized as follows. In section 2 we present the morphisms on the list data type and show that insertion sort, selection sort and bubble sort can be expressed in terms of these morphisms. In section 3 we present the leaf tree data type and show how merge sort can be expressed as a morphism over that data type. In section 4 we present the binary tree data type with its morphisms, which are used to express both quick sort and heap sort. In section 5 paramorphisms on lists are presented, which can be used to express the recursion pattern of several auxiliary functions used by the different sorting algorithms. We show in section 6 that rose trees form a generalization of lists, binary trees and leaf trees. This fact enables a derivation of pairing heap sort and reveals a novel generalization of quick sort. It also opens the door for a taxonomy of algorithms, based on a hierarchy of data structures and on recursion patterns over those data structures. Section 7 presents the conclusions of this paper.

2 Morphisms on lists

The list data type can be described by the following pseudo data type definition in Haskell.

```
data [x] = []
         | x : [x]
```

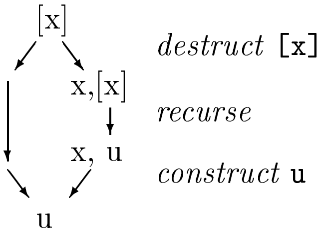
In this section we present three recursion patterns over this data type, and show how *insertion sort*, *selection sort* and *bubble sort* can be expressed by means of these recursion patterns.

2.1 The list catamorphism

A *catamorphism* on a type T is a function of type $T \rightarrow U$ that destruct and object of type T according to the structure of T, calls itself recursively of any components of T that are also of type T and combines this recursive result with the remaining components of T to a U.
A simple example is the function that computes the product of a list of integers:

```
prod []    = 1
prod (x:l) = x * prod l
```

A *list catamorphism* can thus be characterized by two components `(a,f)`, corresponding to the two forms of the list type. The first is a part that maps the *empty list* onto a U. This is just the constant `a` (for `prod` this is `1`). A *non-empty list* can be destructed into a head `h` and a tail `t`. The tail `t` is recursively mapped onto a `u` by `rec t` and then combined with the head by means of the expression `f h (rec t)`, where `f` is the second part of the catamorphism. For `prod` this is `(*)`.
We can present this structure in a diagram where the nodes form the types of the (intermediate) results and the edges the mappings between them.



Recursive functions over lists that have this structure can be written by means of a higher order function of `(a,f)` that captures this recursive patterns. As this recursion pattern is generally called *catamorphism* ($\kappa\alpha\tau\alpha$ means downwards), we call this higher order function `list_cata`. As described above, it returns `a` on the empty list and applies `f` to the head and the recursive call on the tail. the more experienced reader will recognize this function as the well known `foldr`.

```
> type List_cata x u = (u, x->u->u)

> list_cata :: List_cata x u -> [x] -> u
> list_cata (a,f) = cata where
>   cata []    = a
>   cata (x:l) = f x (cata l)
```

With this definition we can rewrite `prod` as follows:

```
> prod = list_cata (1, (*))
```

It can be observed that this catamorphism replaces the empty list by `a` and the non-empty list constructor `(:)` by `f`. This is what a catamorphism does in general: replacing the constructors of a data type by other functions. As a consequence, recursive elements are replaced by the application of the catamorphism to them, i.e. `l` is replaced by `cata l`.

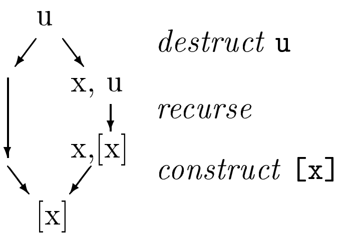
Exercise 1: write the list reversal function as a list catamorphism.

2.2 The list anamorphism

Apart from a recursion pattern that traverses a list, we can specify a converse one that *constructs* a list. A simple example is the function that constructs the list `[n,n-1..1]`.

```
count 0 = []
count n = n : count (n-1)
```

An anamorphism over a type T is a function of type $U \to T$ that destructs the U in some way into a number of components. On the components that are of type U, it applies recursion to convert theme to T's. After that it combines the recursive results with the other components to a T, by means of the *constructor functions* of T.
The structure of this pattern for the list type can be obtained by inverting the catamorphism diagram above.



We need some destructor function of `u` that tells us whether to map `u` onto the empty list, or to destruct it, be means of some function `f` into the head of the resulting list `x` and another `u`. This other `u` can then recursively be mapped onto a list, which forms the tail of the final result.
To this end, we define use the type that represents the disjoint sum of to values.

```
> data Either a b = Left a | Right b
```

Again, this pattern of recursion can be captured in a higher order function. As this recursion pattern is generally called *anamorphism* ($\alpha\nu\alpha$ means upwards), we call this higher order function `list_ana`.

```
> type List_ana u x = u -> Either () (x,u)

> list_ana :: List_ana u x -> u -> [x]
> list_ana a = ana where
>   ana u = case a u of
>           Left _      -> []
>           Right (x,l) -> x : ana l
```

The function `count` can be rewritten as:

```
> count = list_ana destruct_count
> destruct_count 0 = Left ()
> destruct_count n = Right (n,n-1)
```

Exercise 2: write a prime number generator as a list anamorphism.

2.3 The list hylomorphism

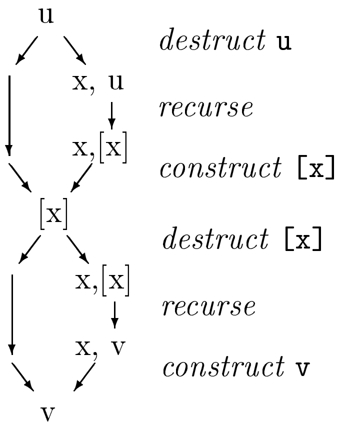
Given a general way to construct a list and to destruct one, we can compose the two operations into a new one, that captures recursion *via* a list. For some philosophical reason, this recursion patterns is called *hylomorphism* ($\acute{\upsilon}\lambda\eta$ means matter).
The list hylomorphism can be defined as

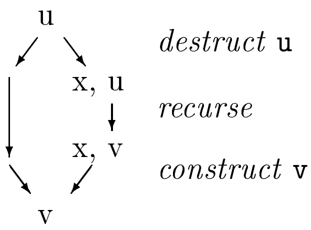
```
list_hylo (a,c) = list_cata c . list_ana a
```

As an example, we could define the factorial function as
`fac = prod .  count`, or more explicitly as:

```
  fac = list_cata (1,(*)) . list_ana destruct_count
```

This straightforward composition forms an intermediate list, which appears to be unnecessary as the following diagrams exhibits.





Instead of construction the list in the middle, we can immediately map the `(x,u)`-pair onto the `(x,v)`-pair by recursively applying the hylomorphism to the `u`.
The implementation is obtained from the `list_ana` by replacing the empty list by `a` and the list constructor `(:)` by `f`.

```
> type List_hylo u x v = (List_ana u x, List_cata x v)

> list_hylo :: List_hylo u x v -> u -> v
> list_hylo (d,(a,f)) = hylo where
>   hylo u = case d u of
>            Left _    -> a
>            Right (x,l) -> f x (hylo l)
```

Applying this to the factorial function, it can be written as:

```
> fac = list_hylo (destruct_count, (1,(*)))
```

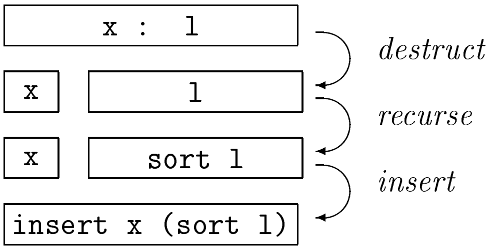
Substitution of `list_hylo` and `destruct_count` then leads to the usual definition of `fac`:

```
fac 0 = 1
fac n = n * fac (n-1)
```

Exercise 3: write the function x^n as a list hylomorphism.

2.4 Insertion sort

We can use the `list_cata` recursion pattern to sort a list. In order to analyze the structure of such a sorting algorithm, we first draw a little diagram.



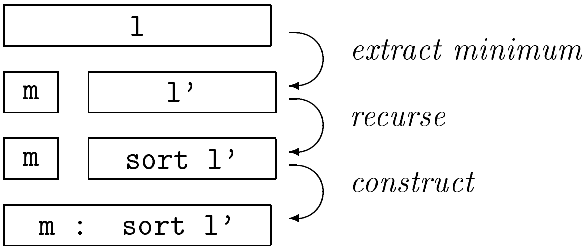
The catamorphism *must* destruct a list `x:l` into a head `x` and a tail `l`, by virtue of its recursive structure. Next it is applied recursively to the tail, which in this case means: sorting it. So we are left with a head `x` and a sorted tail. There is only one way left to construct the full sorted list out of these: insert `x` at the appropriate place in the sorted tail. We see that apart from the recursion pattern, we are left with no more design decision: the resulting algorithm is an insertion sort.

```
> insertion_sort l = list_cata ([],insert) l where
>   insert x [] = [x]
>   insert x (a:l) | x < a     = x:a:l
>                  | otherwise = a : insert x l
```

Observe that `insert x` is a recursive function over lists as well. As it does not correspond to the recursive structures introduced so far, we postpone its treatment until section 5.2.

2.5 Selection sorts

In applying the anamorphism recursion pattern to sorting, we are faced with the following structure.

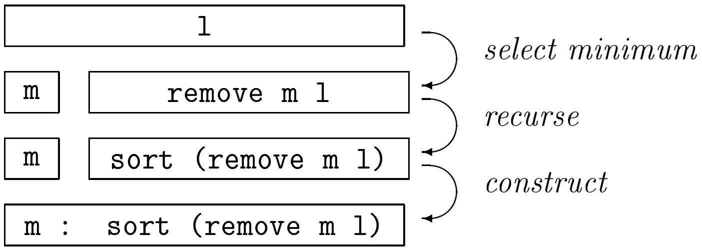


First of all, the unsorted list `l` is mapped onto an element `m` and a remainder `l'`. This remainder is sorted by recursion and serves as the tail of the final result. From this structure, we can deduce that `m` must be the minimum of `l` and `l'` should equal some permutation of `l` with this minimum removed from it. It is the permutation which gives us some additional design freedom. Two ways to exploit this freedom lead to a straight selection sort and a bubble sort respectively. Here we abstract from the way of selection and define the general selection sort as:

```
> selection_sort extract = list_ana select where
>   select [] = Left ()
>   select l  = Right (extract l)
```

2.5.1 Straight selection sort

When we first compute the minimum of `l` and then remove it from `l`, maintaining the order of the remaining elements, we obtain a straight selection sort. It has the following recursive structure.



Its implementation as an anamorphism is as follows.

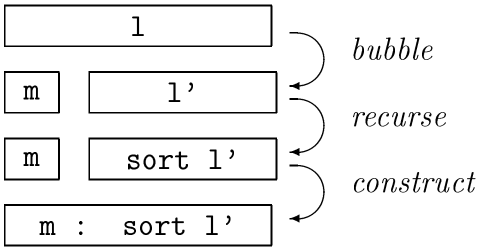
```
> straight_selection_sort l = selection_sort extract l where
>   extract l = (m, remove m l) where m = minimum l

> remove x [] = []
> remove x (y:l) | x == y    = l
>                | otherwise = y : remove x l
```

Observe that `remove x` is a recursive function over lists as well. As it does not correspond to the recursive structures introduced so far, we postpone its treatment until section 5.3, where we also give an implementation of `minimum` as a catamorphism as well.

2.5.2 Bubble sort

Selection sort seems a little too expensive as `select` traverses the list twice, once for obtaining the minimum and once for removing it. We can intertwine these to operations to let the minimum 'bubble' to the head of the list by exchanging elements and then split the minimum and the remainder.




```
> bubble_sort l = selection_sort bubble l where
>   bubble [x]   = (x,[])
>   bubble (x:l) = if x < y then (x,y:m) else (y,x:m) where
>                    (y,m) = bubble l
```

Observe that `bubble` is a recursive function over lists as well. It appears to be a catamorphism, as the following alternative definition shows:

```
> bubble_sort' l = selection_sort bubble l where
>   bubble (x:l) = list_cata ((x,[]),bub) l
>   bub x (y,l)  = if x < y then (x,y:l) else (y,x:l)
```

3 Leaf trees

The sorting algorithms that can be described by list-based recursion patterns all perform linear recursion and as a result behave (at least) quadratically. The $O(n \log n)$ sorting algorithms like quick sort and merge sort use at least two recursive calls per recursion step. In order to express such a recursion pattern we need some binary data structure as a basis for the different morphisms. In this section we concentrate on leaf trees with the elements in their leaves. The next section treats binary trees with the elements at the branches.
One form of binary trees are so-called *leaf-trees*. These trees hold their elements on their leaves. The leaf-tree data type is given by:

```
> data LeafTree x = Leaf x
>                 | Split (LeafTree x) (LeafTree x)
```

3.1 The leaf-tree catamorphism

The structure of the leaf-tree catamorphism is completely analogous to that of the list catamorphism. First destruct the tree, recurse on the sub-trees and combine the recursive results.
An example is the sum of all elements in a leaf tree:

```
  tree_sum Leaf x       = x
  tree_sum (Split l r) = tree_sum l + tree_sum r
```

The leaf-tree catamorphism needs a function on the element, rather than a constant, to construct its non-recursive result. This corresponds to the following diagram, where T stands for `BinTree x`.
The recursion pattern diagram is:

```
     T
   ↙   ↘       destruct T
  x     T,T
  ↓     ↓ ↓    recurse
  x     u,u
   ↘   ↙       construct u
     u
```

Capturing the recursion pattern in a higher order function `leaftree_cata`, gives the following definition (again, just replace the tree constructors `Leaf` and `Split` by other functions, `fl` and `fs` respectively).

```
> type Leaftree_cata x u = (x -> u, u -> u -> u)

> leaftree_cata :: Leaftree_cata x u -> LeafTree x -> u
> leaftree_cata (fl,fs) = cata where
>   cata (Leaf x)    = fl x
>   cata (Split l r) = fs (cata l) (cata r)
```

Using the function `leaftree_cata`, we can define `tree_sum` as:

```
> tree_sum = leaftree_cata (id, (+))
```

3.2 The leaf-tree anamorphism

The structure of the leaf-tree anamorphism is analogous to that of the list anamorphism. First decide by means of a destructor `d` between the tree constructors to be used (`Tip` or `Branch`). This results in an element or in two remaining objects which are recursively mapped onto two trees. Then combine the element or these subtrees.
An example is the construction of a Fibonacci tree:

```
  fib_tree n | n < 2     = Leaf 1
             | otherwise = Branch (fib_tree (n-1)) (fib_tree (n-2))
```

The anamorphism procedure corresponds to the following diagram.

$$\begin{tikzcd}[row sep=small]
 & \mathrm{u} \arrow[dl] \arrow[dr] & & \textit{destruct}\ \mathtt{u} \\
\mathrm{x} \arrow[d] & & \mathrm{u,u} \arrow[d] & \textit{recurse} \\
\mathrm{x} \arrow[dr] & & \mathrm{T,T} \arrow[dl] & \textit{construct}\ \mathtt{T} \\
 & \mathrm{T} & &
\end{tikzcd}$$

Capturing the recursion pattern in a higher order function `leaftree_ana`, gives the following definition.

```
> type Leaftree_ana u x = u -> Either x (u,u)

> leaftree_ana :: Leaftree_ana u x -> u -> LeafTree x
> leaftree_ana d = ana where
>   ana t = case d t of
>           Left l      -> Leaf l
>           Right (l,r) -> Split (ana l) (ana r)
```

Rewriting `fib_tree` with this higher order function gives:

```
> fib_tree = leaftree_ana destruct_fib
> destruct_fib n | n < 2     = Left 1
>                | otherwise = Right (n-1,n-2)
```

3.3 The leaf-tree hylomorphism

As expected, the leaf-tree hylomorphism can be obtained by composing the ana- and the catamorphism.
An example is the Fibonacci function
`fib = tree_sum . fib_tree`, or more explicitly as:

```
  fib = leaftree_cata (id,(+)) . leaftree_ana destruct_fib
```

Again the tree in the middle need not be constructed at all as the following diagram illustrates. We can apply recursion to map the two `u`'s into `v`'s, without constructing the trees.

$$\begin{tikzcd}[row sep=small]
 & \mathrm{u} \arrow[dl] \arrow[dr] & & \textit{destruct}\ \mathtt{u} \\
\mathrm{a} \arrow[d] & & \mathrm{u,u} \arrow[d] & \textit{recurse} \\
\mathrm{a} \arrow[dr] & & \mathrm{v,v} \arrow[dl] & \textit{construct}\ \mathtt{v} \\
 & \mathrm{v} & &
\end{tikzcd}$$

and its implementation with no further comment:

```
> type Leaftree_hylo u x v = (Leaftree_ana u x, Leaftree_cata x v)

> leaftree_hylo :: Leaftree_hylo u x v -> u -> v
> leaftree_hylo (d,(fl,fs)) = hylo where
>   hylo t = case d t of
>            Left l      -> fl l
>            Right (l,r) -> fs (hylo l) (hylo r)
```

Using this definition of `leaftree_hylo` we can define the Fibonacci function as:

```
> fib = leaftree_hylo (destruct_fib, (id,(+)))
```

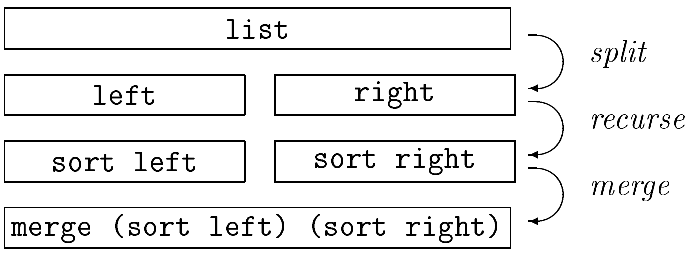
This can of course be simplified by substituting `leaftree_hylo` and `destruct_fib` into:

```
  fib n | n < 2     = 1
        | otherwise = fib (n-1) + fib (n-2)
```

Exercise 4: write the factorial function as a leaf-tree hylomorphism.
Exercise 5: write the function x^n *as a leaftree hylomorphism. What is its complexity? Can you write it as a hylomorphism with* $\mathcal{O}(\log n)$ *complexity?*

3.4 Merge sort

The leaf-tree hylomorphism can be used to sort lists via leaf-trees. The recursion pattern can be depicted as follows.



In the recursive case, the list is split into two sub-list, which are sorted, and then combined. The main choice left here is to make the sub-lists dependent or independent of the elements of the other sub-lists.
When we assume independence, the combination of the recursive results must merge two unrelated sorted lists, and we obtain merge sort.
The choice of two sub-lists which are dependent on each other does not buy us much. If we assume that we can only apply an ordering and an equality relation to the elements, we can't do much more than separating the elements into small and large ones, possibly w.r.t. to the median of the list (which would yield quicksort). We do not pursue this way of sorting any further here.
The implementation of merge sort as an hylomorphism from lists, via leaf-trees, onto lists is given below. The non-recursive case deals with lists of one element. The empty list is treated separately from the hylomorphism.

```
> merge_sort [] = []
> merge_sort l  = leaftree_hylo (select,(single,merge)) l
>  where
>   single x = [x]
>   merge (x:xs) (y:ys) | x < y     = x : merge xs (y:ys)
>                       | otherwise = y : merge (x:xs) ys
>   merge [] m = m
>   merge l [] = l
>   select [x] = Left x
>   select l   = Right (split l)
```

The function `split` splits a list into two sub-list, containing the odd and even elements. We present it here in the form of a list catamorphism.

```
> split = list_cata (([],[]),f) where
>   f x (l,r) = (r,x:l)
```

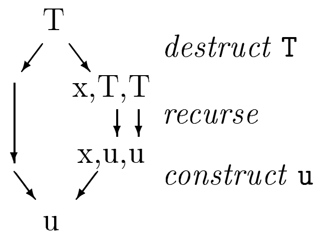
4 Binary trees

Another form of binary trees are trees that hold the values at their *branches* instead of their leaves. This binary tree data type is defined as follows.

```
> data BinTree x = Tip
>                | Branch x (BinTree x) (BinTree x)
```

4.1 The tree catamorphism

The structure of the tree catamorphism should be straight-forward now. First destruct the tree, recurse on the sub-trees and combine the element and the recursive results. This corresponds to the following diagram, where T stands for `BinTree x`.



Capturing the recursion pattern in a higher order function `bintree_cata`, gives the following definition.

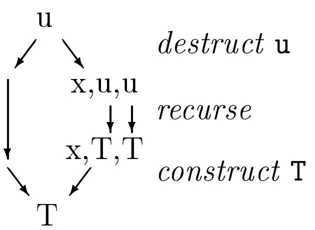
```
> type Bintree_cata x u = (u, x -> u -> u -> u)
```

```
> bintree_cata :: Bintree_cata x u -> BinTree x -> u
> bintree_cata (a,f) = cata where
>   cata Tip            = a
>   cata (Branch x l r) = f x (cata l) (cata r)
```

Observe again that a catamorphism replaces constructors (`Tip` by `a` and `Branch` by `f`) and recursive elements by recursive calls (`l` by `cata l` and `r` by `cata r`).

4.2 The tree anamorphism

The binary tree catamorphism is again obtained by reversing the previous one.



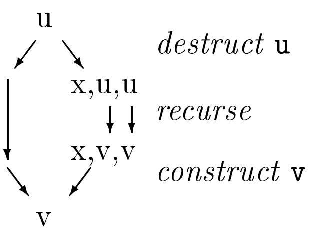
Capturing the recursion pattern in a higher order function `bintree_ana`, gives the following definition.

```
> type Bintree_ana u x = u -> Either () (x,u,u)

> bintree_ana :: Bintree_ana u x -> u -> BinTree x
> bintree_ana d = ana where
>   ana t = case d t of
>           Left _        -> Tip
>           Right (x,l,r) -> Branch x (ana l) (ana r)
```

4.3 The tree hylomorphism

The binary tree hylomorphism should be straightforward now. We present only its diagram



The implementation of the binary tree hylomorphism is obtained from the anamorphism by replacing `Tip` by `a` and the `Branch` constructor by `f`.

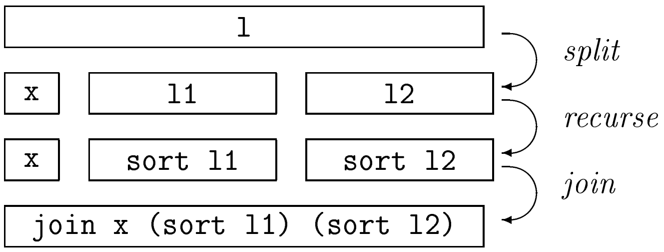
```
> type Bintree_hylo u x v = (Bintree_ana u x, Bintree_cata x v)
```

```
> bintree_hylo :: Bintree_hylo u x v -> u -> v
> bintree_hylo (d,(a,f)) = hylo where
>   hylo t = case d t of
>            Left _       -> a
>            Right (x,l,r) -> f x (hylo l) (hylo r)
```

Exercise 6: write the factorial function as a binary tree hylomorphism.
Exercise 7: write the function x^n as a binary tree hylomorphism. What is its complexity?
Exercise 8: write the towers of hanoi as a binary tree hylomorphism.

4.4 Quicksort

We can apply the binary tree hylomorphism recursion pattern to sorting by sorting a list via binary trees (which are never really constructed of course). The following diagram exhibits the recursion pattern.

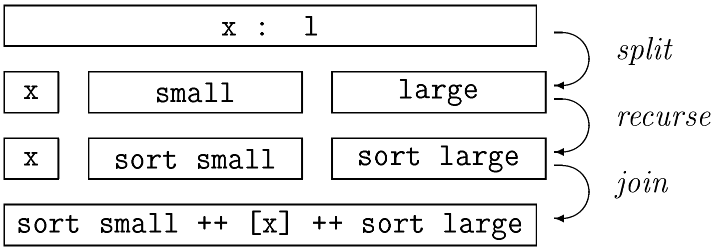


A list `l` is split into an element `x` and two other lists `l1` and `l2`. These lists are sorted by recursion. Next `x` and the sorted sub-lists are joined. We are left with a two design decisions here.

- The choice of `x`. Sensible choices here are the head of the list, using the structure of the list, or the minimum or median of the list, exploiting the ordering relation. If we take the minimum, we obtain a variant of heap sort. A derivation of heap sort is given in section 4.5. For quicksort, we choose to take the head of the list. Taking the median is left to the reader.

- The choice of the two sub-lists. An essential choice here is to make them dependent on the element `x` or not. If not, there seems to be no particular reason to separate `x`. If we do not use the head `x` at all, the algorithm obeys a different recursion pattern, which we treat in section 3.
  The remaining option, making the sub-lists depend on `x`, still leaves some choices open. The most natural one seems to let them consists of the elements that are smaller than, respectively at least `x`, exploiting the ordering relation. This can be done for `x` being either the head or the median of the list, where the latter gives a better balanced algorithm with a superior worst case behavior. We will take the head for simplicity reasons here.

Given the decisions that we take `x` to be head of the list and split the tail into small and large elements w.r.t. to `x`, the only way in which we can combine the sorted sub-lists with `x` is to concatenate them in the proper order.



The final result is an implementation of *quicksort* as a hylomorphism from lists, via trees, onto lists.

```
> quick_sort l = bintree_hylo (split,([],join))  l where
>   split []    = Left ()
>   split (x:l) = Right (x,s,g) where (s,g) = partition (<x) l
>   join x l r  = l ++ x : r
```

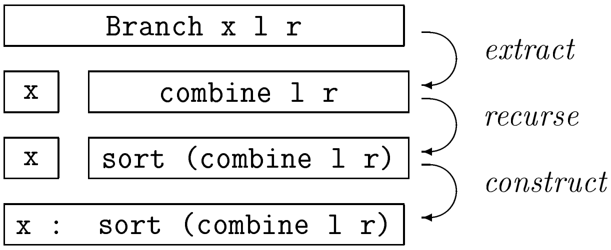
The function `partition` which splits a list into two lists w.r.t. to some predicate `p` appears to be a list catamorphism.
Exercise 9: write the function `partition` *as a list catamorphism.*

4.5 Heap sort

In this section we analyze the recursion pattern of heap sort. This algorithm operates by first constructing a binary tree that has the so called *heap property*, which means that for such a tree `Branch m l r`, `m` is the minimum element in the tree. The two sub-trees `l` and `r` must also have the heap property, but are not related to each other.
After constructing such a heap, the heap is mapped onto a sorted list. Therefore, the defition of heap sort is simply:

```
> heap_sort l = (heap2list . list2heap) l
```

Such a tree is transformed into a sorted list in the following way, where `combine l r` combines two heaps into a new one. It is clearly a list anamorphism.

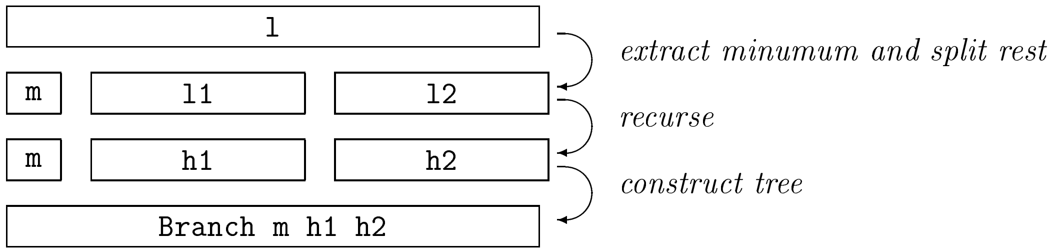


Thus, heapsort can be implemented as below, leaving the function `list2heap`, which transforms an unsorted list into a heap, to be specified. The `b@(Branch x l r)` construction in Gofer matches an argument to the pattern `Branch x l r` as usual, but also binds the argument as a whole to `b`.

```
> heap2list l = list_ana extract l where
>   extract Tip            = Left ()
>   extract (Branch x l r) = Right (x, combine l r)

> combine :: Ord a => BinTree a -> BinTree a -> BinTree a
> combine t Tip   = t
> combine Tip t   = t
> combine b@(Branch x l r) c@(Branch y s t)
>     | x < y     = Branch x l (combine r c)
>     | otherwise = Branch y (combine b s) t
```

Three recursion patterns that could be applicable to the function `list2heap` are a list catamorphism, a tree anamorphism, or a hylomorphism over some additional data type. Let us analyze the recursion pattern of such a function, where we assume that we use `list2heap` recursively in a binary tree pattern (note that this a design decision), more particularly, let us choose the tree anamorphism. They other options work just as well, but for simplicity, we do not persue them here.



The decomposition is a variant of `bubble`: it should not only select the minimum but also split the remainder of the list into two sub-lists of (almost) equal length. This bubbling is once more a list catamorphism.

```
> list2heap l = bintree_ana decompose l where
>   decompose [] = Left ()
>   decompose l  = Right (bubble l)
>   bubble (x:l) = list_cata ((x,[],[]),bub) l
>   bub x (y,l,r)  = if x < y then (x,y:r,l) else (y,x:r,l)
```

Thus, heap sort can be written as the composition of a binary tree anamorphism and a list anamorphism.

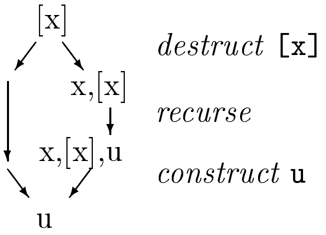
5 Paramorphisms on lists

Several sub-functions, like `insert` and `remove` used above where almost catamorphisms on lists. They deviate by the fact that in the construction of the result, they do not only

use the recursive result of the tail, but also the tail itself. This recursion pattern is known as *paramorphism*, after the Greek word $\pi\alpha\rho\alpha$, which among other things means 'parallel with'.
Paramorphisms can be expressed as catamorphisms by letting the recursive call return its intended result, tupled with its argument. Here, we study them as a separate recursion pattern however.

5.1 The list paramorphism

The list paramorphism follows the recursion patterns of the following diagram.



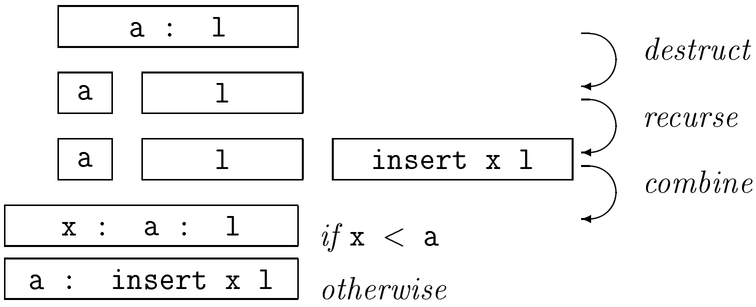
Its implementation is straight-forward, just supply the tail `l` as an additional argument to the constructor function `f`.

```
> type List_para x u = (u, x -> [x] -> u -> u)

> list_para :: List_para x u -> [x] -> u
> list_para (a,f) = para where
>   para []    = a
>   para (x:l) = f x l (para l)
```

5.2 Insert as paramorphism

The insertion operation `insert x` of the insertion sort from section 2.4 can be expressed as a paramorphism. First we analyze its recursive structure.

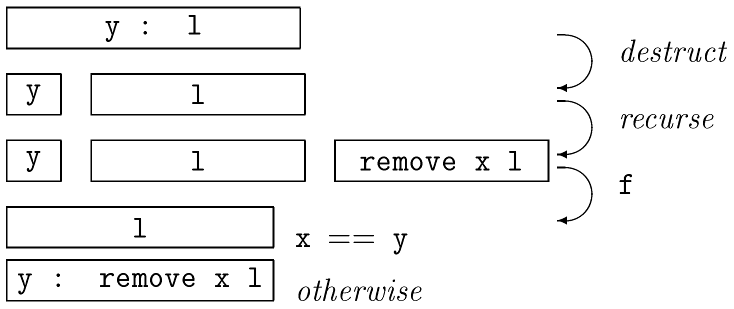


The list `a:l` is split into head `a` and tail `l`. Recursively, `x` is inserted into `l`. Depending on where `x < a`, we need to use the original tail, or the recursive result.
Although it may seem inefficient to construct the recursive result and then optionally throw it away again, laziness comes to help here. If the recursive result is not used, it is simply not computed.

```
> insertion_sort' l = list_cata ([],insert) l where
>   insert x = list_para ([x],combine) where
>     combine a l rec | x < a     = x : a : l
>                     | otherwise = a : rec
```

5.3 Remove as paramorphism

The selection operation `remove x` of the straight selection sort from section 2.5.1 can be expressed as a paramorphism as well. First we analyze its recursive structure.



It destructs the list into the head `y` and tail `l`. It recursively removes `x` from `l`. Next, it chooses between using the non-recursive tail `l` (when `x == y`, it suffices to remove `y`), or, in the other case, to maintain `y` and use the recursive result.

Below, we give the paramorphic version of the straight selection sort algorithm. Observe that `minimum` has been written as a catamorphism.

```
> straight_selection_sort' l = selection_sort extract l where
>   extract l  = (m, remove m l) where m = minimum l
>   minimum (x:l) = list_cata (x,min) l
>   remove x = list_para ([],f) where
>     f y l rec | x == y    = l
>               | otherwise = y : rec
```

6 Generalizing data structures

The previous sections have shown that the use of patterns of recursion gives rise to a classification of sorting algorithms. One can obtain a refined taxonomy of sorting algorithms by introducing yet another level of generalization. This extra level is the generalization of data types. We illustrate this by generalizing the binary tree data type to the rose tree data type. This type is equivalent to the so called *B*-trees in [Knu73], where they are used sor searching.

```
> data RoseTree a = RoseTree [a] [RoseTree a]
```

A binary tree has 1 element and 2 branches, a rose tree n elements and m branches. The empty rose tree is represented by the $m = n = 0$ case. Since quicksort is a hylomorphism on binary trees, a hylomorphism on rose trees is expected to be a generalization of quicksort. The rose tree hylomorphism is given by the following diagram and definition.

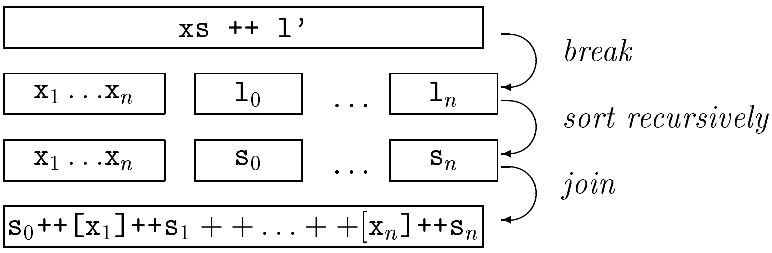
$$\begin{tikzcd}
\mathrm{u} \arrow[d, "\textit{destruct}\ \texttt{u}"] \\
{[\mathrm{x}],[\mathrm{u}]} \arrow[d, "\textit{recurse}"] \\
{[\mathrm{x}],[\mathrm{v}]} \arrow[d, "\textit{construct}\ \texttt{v}"] \\
\mathrm{v}
\end{tikzcd}$$

```
> type Rosetree_cata x v = [x] -> [v] -> v
> type Rosetree_ana u x = u -> ([x],[u])
> type Rosetree_hylo u x v = (Rosetree_ana u x, Rosetree_cata x v)

> rosetree_hylo :: Rosetree_hylo u x v -> u -> v
> rosetree_hylo (f,g) = hylo where
>   hylo t = g x (map hylo l) where (x,l) = f t
```

6.1 Generalizing quicksort

The generalization of quicksort can be obtained by using n pivots, rather than 1. These n pivots are sorted, e.g. by `insertion_sort` for small n, and the remaining elements are grouped into $n+1$ intervals w.r.t. the pivots. I.e. the first internal contains all elements less than the first pivot, the second internal contains all remaining elements less than the second pivot, etc., while the $n+1$-th interval contains all elements not less than the n-th pivot. These intervals are sorted recursively, and the intervals and pivots are joined together. The following diagram illustrates this process.



The implementation is relatively straight-forward after this design. The `l==[]` case needs to be treated specially to ensure termination. First, `l` is split into its first (at most) n elements `xs` and the remainder `l'`. Next `xs` is sorted to obtain `sx`. Then `l'` is split w.r.t. to `sx`.

```
> rose_sort n l = rosetree_hylo (break,join) l where
>   break [] = ([],[])
>   break l  = (sx,split sx l') where
>     (xs,l') = take_drop n l
>     sx = insertion_sort xs
>     split sx l = list_cata ([l],f) sx where
>       f x (a:l) = s:g:l where (s,g) = partition (<x) a
>   join xs []    = xs
>   join xs (s:l) = s++concat (zipWith (:) xs l)

> take_drop 0 l = ([],l)
> take_drop n [] = ([],[])
> take_drop n (x:l) = (x:a,b) where (a,b) = take_drop (n-1) l
```

Experiments show that this algorithm behaves superour to the `quick_sort` function when applied to random list of various sizes. The optimal value of `n` appears to be independent of the length of the list (it equals 3 in this implementation). A decent analysis of the complexity of this algorithm should reveal why this is the case. The split size can be adapted to the length of the list L by replacing `n` by some function of L. It is an open problem which function will give the best behavior.
Since rose trees can be viewed as a generalization of linear lists, binary trees and leaf trees together, the other sorting algorithms generalize as well. E.g. the two-way merge sort becomes a k-way merge sort. We leave this generalization as an exercise to the reader

6.2 Generalizing heap sort

Heap sort can be generalized by means of rose trees as well. The obvious way is to define a heap on rose trees, instead of binary trees, construct the tree from a list and map it onto a sorted list.

The empty heap is represented by `Rosetree [] []`, the non-empty heap by `Rosetree [x] l`, where `x` is the minimum element and `l` a list of heaps. We aim at keeping `l` balanced, that is, let the sub heaps vary in sizes as little as possible, to obtain a good worst-case behavior.

This variant of heap sort is known in the literature as *pairing sort* [FSST86].

```
> pairingSort l = (rose2list . list2rose) l
```

The function `rose2list` is a variant of `heap2list`. Instead of combining two heaps into a new heap, it should combine (meld) a list of heaps into heap. we postpone the treatment of this function `roses_meld`.

```
> rose2list :: (Ord a) => RoseTree a -> [a]
> rose2list = list_ana destruct where
```

```
>   destruct (RoseTree []  ts) = Left ()
>   destruct (RoseTree [a] ts) = Right (a, roses_meld ts)
```

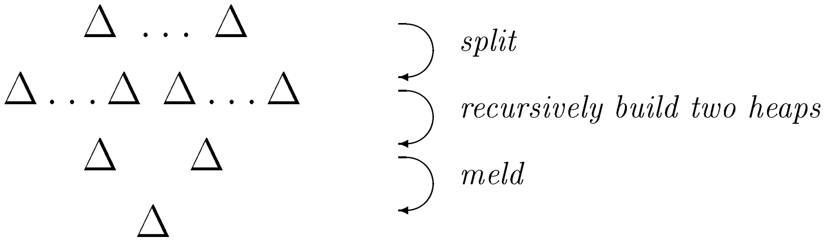
Mapping a list into a heap is simple if we use the postponed function `roses_meld`.

```
> list2rose  :: (Ord a) => [a] -> RoseTree a
> list2rose  = roses_meld . map single where
>   single a = RoseTree [a] []
```

The function `roses_meld` can be designed best by first defining a function that melds two heaps onto a new heap. This is the rose tree variant of the function `combine` from section 4.5.

```
> rose_meld :: Ord a => RoseTree a -> RoseTree a -> RoseTree a
> rose_meld (RoseTree [] _) u = u
> rose_meld t (RoseTree [] _) = t
> rose_meld t@(RoseTree [a] ts) u@(RoseTree [b] us)
>     | a < b     = RoseTree [a] (u:ts)
>     | otherwise = RoseTree [b] (t:us)
```

The function `roses_meld` is now a simple list catamorphism over `rose_meld`. Implementing it that way has one draw-back however: it is not balanced. A list catamorphisms groups the elements together in a linear way. If the combining function is associative, the grouping can take place in a tree-shaped way as well, giving a balanced heap. We call this form of folding a list of values into a single value `treefold` and it is a leaf-tree hylomorphism.




```
> roses_meld :: Ord a => [RoseTree a] -> RoseTree a
> roses_meld = treefold rose_meld no_roses

> treefold :: (a -> a -> a) -> a -> [a] -> a
> treefold f e l = leaftree_hylo (select,(id,f)) l where
>   select []  = Left e
>   select [a] = Left a
>   select l   = Right (split l)

> no_roses = RoseTree [] []
```

7 Conclusions

We have shown that it is possible to express the most well-known sorting algorithms as functional programs, using fixed patterns of recursion. Given such a pattern of recursion there is little or no additional design freedom left. The approach shows that functional programming in general and the study of recursion patterns in particular form a powerful tool in both the characterization and derivation of algorithms. In this paper we studied the three data types linear lists, binary trees and binary leaf trees. Generalizing these data types to rose trees revealed a generalization of quick-sort, which is, as far as the author knows, a novel sorting algorithm. It may well be that other data types, and their corresponding recursion patterns, can be used to derive even more sorting algorithms.
By distinguishing a hierarchy of data structures on the one hand, and different patterns of recursions on the other hand, a taxonomy of algorithms can be constructed. It would be nice to compare this with other techniques for constructing algorithm taxonomies as e.g. presented in [Wat95].
Of course, other algorithms than sorting can be classified by means of their pattern of recursion. See e.g. [Aug93], where similar techniques were used to characterize parsing algorithms.
The question whether the presentation of the algorithms as such is clarified by their expression in terms of morphisms has not been raised yet. When we compare the catamorphic version of insertion sort to the following straight implementation, the latter should be appreciated over the first.

```
insertion_sort []    = []
insertion_sort (x:l) = insert x (insertion_sort l) where
  insert x [] = [x]
  insert x (a:l) | x < a     = x :a:l
                 | otherwise = a : insert x l
```

The value of this approach is not so much in obtaining a nice presentation or implementation of some algorithm, but in unraveling its structure. Especially in the case of heap sort, this approach gives a very good insight in the structure of the algorithm, compared for instance to [Knu73] or [Wir76].
We based the recursion patterns on the natural recursive functions over data types, the catamorphism, anamorphism, hylomorphism and paramorphism. This let to a very systematic derivation of recursion patterns. The category theory that underlies these morphisms was not needed in this presentation. There definition follows so trivially from the data type definition that any functional programmer should be able to define them. A generation of these recursion patterns by a compiler for a functional language is even more desirable: then there is not need at all for a programmer to write them.

Acknowledgements
Above all, I wish to thank Frans Kruseman Aretz for the patient and careful supervision during the writing of my thesis, of which this contribution is a natural continuation. I am also grateful to Doaitse Swierstra and Tanja Vos for stimulating discussions about the

different morphisms, to Erik Meijer for his illuminating thesis and to Herman ter Horst for his refereeing.

References

[Aug93] Lex Augusteijn. *Functional Programming, Program Transformations and Compiler Construction.* PhD thesis, Eindhoven Technical University, October 1993.

[BdM94] R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice-Hall, 1994.

[Fok92] Maarten M. Fokkinga. *Law and order in algorithmics.* PhD thesis, Twente University, 1992.

[FSST86] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[HWA+92] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young. Report on the Programming Language Haskell, A Non-Strict, Purely Functional Language, Version 1.2. *ACM SIGPLAN Notices*, 27(5):Section R, 1992.

[Jon95] Mark P. Jones. *Release notes for Gofer 2.30b.*, September 1995. Included as part of the standard Gofer distribution, `http://www.cs.nott.ac.uk:80/Department/Staff/mpj/`.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973. Sorting and Searching.

[Mei92] Erik Meijer. *Calculating Compilers.* PhD thesis, Utrecht State University, Utrecht, the Netherlands, 1992.

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Eindhoven University of Technology, 1995.

[Wir76] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, 1976.